

Iterators

Object-Oriented Programming with C++

Why need iterators

- Provide a way to visit the elements in order, without knowing the details of the container.
 - *Generalization* of pointers

Why need iterators

- Separate container and algorithms with standard iterator interface functions.
 - The *glue* between algorithms and data structures
 - Without iterators, with N algorithms and M data structures, you need $N*M$ implementations

What are iterators

- One of *design patterns* (Gang of Four):
 - “ Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. ”

Usage

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

Usage

```
vector<int> vecTemp;  
list<double> listTemp;  
  
if (find(vecTemp.begin(), vecTemp.end(), 3) == vecTemp.end())  
    cout << "3 not found in vecTemp" << endl;  
  
if (find(listTemp.begin(), listTemp.end(), 4) == listTemp.end())  
    cout << "4 not found in listTemp" << endl;
```

Requirements

- A *unified interface* used in algorithms
- Work like a *pointer* to the elements in a container
- Have `++` operator to visit elements in order
- Have `*` operator to visit the content of an element

auto_ptr

- An example of overloading operators `*` and `->`

```
template<class T>
class auto_ptr {
private:
    T *pointee;
public:
    /* ... */
    T& operator *() { return *pointee; }
    T* operator ->() { return pointee; }
};
```


List container

```
template<class I>
class List {
public:
    void insert_front();
    void insert_end();
    /* ... */
private:
    ListItem<T> *front;
    ListItem<T> *end;
    long _size;
};
```

```
template<class I>
class ListItem {
public:
    T& val() {
        return _value;
    }
    ListItem* next() {
        return _next;
    }
    /* ... */
private:
    T _value;
    ListItem<T> *_next;
};
```

List iterators

```
template<class I>
class ListIter {
    ListItem<T> *ptr;
public:
    ListIter(ListItem<T> *p=0) : ptr(p) {}
    ListIter<Item>& operator++()
        { ptr = ptr->next(); return *this; }
    bool operator==(const ListIter& i) const
        { return ptr == i.ptr; }
    /* ... */
    T& operator*() { return ptr->val(); }
    T* operator->() { return &(**this); }
};
```

find in List container

- Enabled by `ListIter`:

```
List<int> myList;  
... // insert elements  
  
ListIter<int> begin = myList.begin();  
ListIter<int> end = myList.end();  
ListIter<int> iter;  
  
iter = find(begin, end, 3);  
if (iter == end)  
    cout << "not found" << endl;
```

The associated type

```
// we do NOT know the data type of iter,  
// so we need another variable v to infer T  
template <class I, class I>  
void func_impl(I iter, T& v)  
{  
    T tmp;  
    tmp = *iter;  
    // processing code here  
}
```

The associated type

```
// a wrapper to extract the associated  
// data type T  
template <class I>  
void func(I iter)  
{  
    func_impl(iter, *iter);  
    // processing code here  
}
```

However, we might need more type information that associated to iterators.

Type info. definition

Explicitly define the type info. inside iterators.

```
template <class I>
struct myIter {
    typedef T value_type;
    /* ... */
    T* ptr;
    myIter(T *p = 0) : ptr(p) {}
    T& operator*() {
        return *ptr;
    }
};
```

```
template <class I>
typename I::value_type
func(I iter) {
    return *iter;
}

// code
myIter<int> iter(new int(8));
cout << func(iter);
```

Pitfalls

The problem of the *typedef* trick:

“ It cannot support *pointer-type* iterators, e.g.,
`int*`, `double*`, `Complex*`, which cripples the
STL programming. ”

Use iterator_traits trick

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


Usage of iterator_traits

```
template <class I>
typename iterator_traits<I>::value_type
func(I iter) {
    return *iter;
}

// code
myIter<int> iter(new int(8));
cout << func(iter);
int* p = new int[20]();
cout << func(p); // iterator_traits<int*> ??
```

Template specialization

- Primary template:

```
template<class T1, class T2, int I>  
class A { /* ... */ };
```

Template specialization

- Explicit (full) template specialization:

```
template<>  
class A<int, double, 5> { /* ... */ };
```

- Partial template specialization:

```
template<class T2>  
class A<int, T2, 3> { /* ... */ };
```

Iterator traits

Template specialization with *pointers*:

```
template<class I>
class C
{
public:
    C() {
        cout << "template
            T" << endl;
    }
};
```

```
template<class I>
class C<T*>
{
public:
    C() {
        cout << "template
            T*" << endl;
    }
};
```

Iterator traits

The *traits* technique with template specialization:

```
template<class I>
class iterator_traits
{
public:
    typedef typename
        I::value_type value_type;
    typedef typename
        I::pointer_type pointer_type;
    /* ... */
};
```

```
template<class I>
class iterator_traits<T*>
{
public:
    typedef T value_type;
    typedef T* pointer_type;
    /* ... */
};
```

Iterator traits

The *traits* technique with template specialization:

```
template<class I>
class iterator_traits
{
public:
    typedef typename
        I::value_type value_type;
    typedef typename
        I::pointer_type pointer_type;
    /* ... */
};
```

```
template<class I>
class iterator_traits
<const T*>
{
public:
    typedef T value_type;
    typedef const T*
        pointer_type;
    /* ... */
};
```

Standard traits in STL

```
template<class I>
class iterator_traits
{
public:
    typedef typename I::iterator_category iterator_category;
    typedef typename I::value_type value_type;
    typedef typename I::difference_type difference_type;
    typedef typename I::pointer pointer;
    typedef typename I::reference reference;
    /* ... */
}
```

Standard traits in STL



value_type
difference_type
pointer
reference
iterator_category
...



```
int *  
const int*  
list<int>::iterator  
deque<int>::iterator  
vector<int>::iterator  
MyIter  
...
```

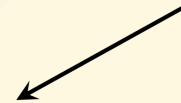
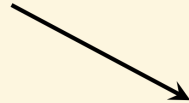

Iterator category (types)

- InputIterator
- OutputIterator
- ForwardIterator
- BidirectionalIterator
- RandomAccessIterator

Iterator category (types)

InputIterator

OutputIterator



ForwardIterator



BidirectionalIterator



RandomAccessIterator

Iterator method: advance

```
template<class InputIterator, class Distance>
void advance_II(InputIterator &i, Distance n)
{
    while (n-- > 0) ++i;
}
```

Iterator method: advance

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

Iterator method: advance

```
template<class RandomAccessIterator, class Distance>
void advance_RAI(RandomAccessIterator &i, Distance n)
{
    i += n;
}
```

Iterator method: advance

But, how to call the correct version of `advance()` *according to the iterator types?*

Use iterator category info.

```
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 {};
```

Iterator method: advance

```
template<class InputIterator, class Distance>
inline void __advance(InputIterator &i,
                      Distance n,
                      input_iterator_tag)
{
    while (n-- > 0) ++i;
}
```


Iterator method: advance

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

Iterator method: advance

```
template<class RandomAccessIterator, class Distance>
inline void __advance(RandomAccessIterator &i,
                      Distance n,
                      random_access_iterator_tag)
{
    i += n;
}
```

Iterator method: advance

Use traits again! Create a temporary object...

```
template<class Iterator, class Distance>
inline void advance(Iterator &i, Distance n)
{
    __advance(i, n,
        iterator_traits<Iterator>::iterator_category());
}
```

Partial specialization for *raw pointers*

```
template <class I>
struct iterator_traits {
    /* ... */
    typedef typename I::iterator_category iterator_category;
};

template <class I>
struct iterator_traits<T*> {
    /* ... */
    typedef random_access_iterator_tag iterator_category;
};
```

Pure transfer

The function version with pure transfer, from `forward_iterator_tag` to `input_iterator_tag`, can be simply *removed* due to inheritance (implicit conversion).

```
template<class ForwardIterator, class Distance>
inline void __advance(ForwardIterator &i, Distance n,
                     forward_iterator_tag)
{
    __advance(i, n, input_iterator_tag());
}
```

Iterator method: distance

```
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;
}
```

Iterator method: distance

```
template<class RandomAccessIterator>
inline iterator_traits<RandomAccessIterator>::difference_type
__distance(RandomAccessIterator first,
           RandomAccessIterator last,
           random_access_iterator_tag)
{
    return last - first;
}
```

Iterator method: distance

The wrapper function

```
template<class Iterator>
inline iterator_traits<Iterator>::difference_type
distance(Iterator first, Iterator last)
{
    return __distance(first, last,
        iterator_traits<Iterator>::iterator_category());
}
```


Iterators

- Container *knows* how to design its own iterator.
- Traits trick extracts type information *embedded* in different iterators, including raw pointers.
- Algorithms are *independent* to containers through the design philosophy of iterators.