目录

第11章 Iterators

11.1 Iterator的作用

1. 在算法中使用的统一接口
2. 像指向容器中元素的指针一样工作
3. 通过++操作符顺序访问容器的元素
4. 通过*操作符访问元素的内容

11.1.1 auto_ptr

```
template<class T>
class auto_ptr {
    T *pointee;
public:
    explicit auto_ptr(T *p) { pointee = p;}

    template <class U>
    auto_ptr(const auto_ptr<U> &rhs): pointee(rhs.release) { }

    // 成员函数模板
    template<class U>
    Auto_ptr<T>& operator=(const auto_ptr<U>  &rhs) {
        if (this != &rhs) reset(rhs.release());
        return *this;
    }

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

11.1.2 iter的基础实现：listIter

```
// List的内部实现
template<class T>
class ListItem {
public:
    T value() { return _value; }
```

```
    ListItem *next { return _next};
private:
    T _value;
    ListItem<T> *_next;
};
// List对象
template<class T>
class List {
public:
    void insert_front();
    void insert_end();
private:
    ListItem<T> *front;
    ListItem<T> *end;
    long _size;
}
// List迭代器对象
template<class Item>
class ListIter {
    Item *ptr;
public:
    ListIter(Item *p=0) :ptr(p) {}
    // 要实现基础的这三个操作符
    ListIter<Item> &operator++() { ptr = ptr->next; return *this; }
    Item& operator*() { return *ptr; }
    Item* operator->() { return ptr;}
};
// 使用List迭代器对象
void func(){
    List<int> myList;
    ListIter<ListItem<int> > begin(myList.begin());
    ListIter<ListItem<int> > end(myList.end());
    ListIter<ListItem<int> > iter;

    iter = find(begin, end, 3);
    if (iter == end)
        cout<<“not found”<< endl;
}
```

11.1.3 对 迭代器 及 迭代器指向的类型 进行编程

直接实现：

1. **func**需要使用**iterator**指向的元素的类型
2. 由于**iterator**是一个模板，因此我们不知道其指向的类型是什么
3. 将与指向元素类型有关的代码封装为另一个函数**func_impl**

```
template <class I, class T> // 我们不知道iter指向的对象类型是什么
void func_impl(I iter, T v){   // 所以需要一个额外的变量T v来表示iter指向的对象及其类型
    T tmp;
    tmp = *iter;
    //processing code here
}

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

```

使用**iterator**特征提取器

1. **typename**表示后面的类型，是可以推导出来的
2. 这样就可以直接获得迭代器指向的对象的类型**I::value_type**

```
template <class T>
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;
}
```

11.2 partial specialization偏特化

对同一个模板类C

1. 当传入的类型是指针时，使用下面的模板进行实例化
2. 当传入的类型不是指针时，使用上面的模板进行实例化

```
// 传入的类型不是指针
template<class T>
class C{
public:
    C() {cout<<“template T”<<endl;}
}
// 传入的类型是指针
template<class T>
class C<T*>{
public:
    C() {cout<<“template T*”<<endl;}
}
```

11.2.2 Iterator中的偏特化

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

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

11.2.3 STL中的标准特征提取技术

```
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 differece_type;
    typedef typename I::pointer_type pointer_type;
    typedef typename I::reference reference;
}
```