2 STL的编程范式

OOP(Object-Oriented Programming)：面向对象 数据和操作在同一个类;OOP企图将datas和methods关联在一起

```
template<class T, class Alloc = alloc>
class list{
    ...
    void sort();
}
```

GP(Generic Programming)：泛型编程 datas和methods分隔开，即algorithm和contains分隔开，通过iterator交互。

```
template<typename _RandomAccessIterator>
inline void sort(_RandomAccessIterator __first, _RandomAccessIterator __end)
```

STL采用GP的原因：

1. Containers和Algorithms团队刻个字闭门造车，Iterators团队沟通。
2. Algorithms通过Iterators确定操作范围，并通过Iterators取用Containers元素。

例子：

有算法（Algorithms）如下：

```
template<class T>
inline const min T&(const T& a, const T& b){
    return b < a ? b : a;
}
```

如果要对一个自定义类进行大小比较，则可以重载<，或者写个Compare函数。这样，算法就有了其通用性，而无需关心容器是什么。

泛化、特化、偏特化

特化即特殊化，即设计者认为对于制定类型，使用特定版本更好。

全特化就是限定死模板实现的具体类型。

偏特化就是如果这个模板有多个类型，那么只限定其中的一部分。

优先级：*全特化类>偏特化类>主版本模板类*

```
//泛化
Template <class type>
Struct __type_traits{typedef __true_type this_dummy_member_must_be_first; };
//特化1
Template < >
Struct __type_traits<int>{typedef __true_type this_dummy_member_must_be_first; };
//特化2
Template < >
Struct __type_traits<double>{typedef __true_type this_dummy_member_must_be_first;
};
//__type_traits<FOO>:: this_dummy_member_must_be_first; 使用的是泛化的内容

//泛化
Template <class T, class Alloc = alloc>
Class vecor{};
//偏特化(个数偏特化，第一个特化，第二个不特化)
Template <class Alloc>
Class vector<bool, Alloc>{};

//泛化
Template <class Iterator>
Struct iterator_traits {};
//偏特化1（范围偏特化，只能是传入指针）
Template <class T>
Struct iterator_traits<T*>{};
//偏特化2
Template <class T>
Struct iterator_traits<const T*>{};
```

为什么list不能使用::sort函数

list底层数据结构为链表，不支持随机访问（random access），所以list这个Containers中，有自带的sort方法。

::sort接口为：

```
sort(_RandomAccessIterator __first, _RandomAccessIterator __last, _Compare __comp)
{
    typedef typename __comp_ref_type<_Compare>::type _Comp_ref;
    _VSTD::__sort<_Comp_ref>(__first, __last, _Comp_ref(__comp));
}
```

list.sort为，可以看到为链表的归并排序：

```
template <class _Tp, class _Alloc>
template <class _Comp>
typename list<_Tp, _Alloc>::iterator
list<_Tp, _Alloc>::__sort(iterator __f1, iterator __e2, size_type __n, _Comp&
__comp)
{
    switch (__n)
    {
    case 0:
    case 1:
        return __f1;
    case 2:
        if (__comp(*--__e2, *__f1))
        {
            __link_pointer __f = __e2.__ptr_;
            base::__unlink_nodes(__f, __f);
            __link_nodes(__f1.__ptr_, __f, __f);
            return __e2;
        }
        return __f1;
    }
    size_type __n2 = __n / 2;
    iterator __e1 = _VSTD::next(__f1, __n2);
    iterator  __r = __f1 = __sort(__f1, __e1, __n2, __comp);
    iterator __f2 = __e1 = __sort(__e1, __e2, __n - __n2, __comp);
    if (__comp(*__f2, *__f1))
    {
        iterator __m2 = _VSTD::next(__f2);
        for (; __m2 != __e2 && __comp(*__m2, *__f1); ++__m2)
            ;
        __link_pointer __f = __f2.__ptr_;
        __link_pointer __l = __m2.__ptr_->__prev_;
        __r = __f2;
        __e1 = __f2 = __m2;
        base::__unlink_nodes(__f, __l);
        __m2 = _VSTD::next(__f1);
        __link_nodes(__f1.__ptr_, __f, __l);
        __f1 = __m2;
    }
    else
        ++__f1;
    while (__f1 != __e1 && __f2 != __e2)
    {
```

```
        if (__comp(*__f2, *__f1))
        {
            iterator __m2 = _VSTD::next(__f2);
            for (; __m2 != __e2 && __comp(*__m2, *__f1); ++__m2)
                ;
            __link_pointer __f = __f2.__ptr_;
            __link_pointer __l = __m2.__ptr_->__prev_;
            if (__e1 == __f2)
                __e1 = __m2;
            __f2 = __m2;
            base::__unlink_nodes(__f, __l);
            __m2 = _VSTD::next(__f1);
            __link_nodes(__f1.__ptr_, __f, __l);
            __f1 = __m2;
        }
        else
            ++__f1;
    }
    return __r;
}
```