Effective STL

基本概念

STL定义：标准容器、iostream库的一部分、函数对象和各种算法。排除标准容器配接器（stack、queue和priority_queue），因为它们缺少对迭代器的支持。同时也排除数组。

STL平台：一个特定的编译器和一个特定的STL实现的组合。

1.慎重选择容器类型

- 标准序列容器：vector、string、deque、list；
- 标准关联容器：set、multiset、map、multimap；
- 非标准序列容器：slist（单向链表）、rope（重型string）；
- 非标准关联容器：hash_set、hash_multiset、hash_map、hash_multimap；
- vector<char>作为string的替代；
- vector作为标准关联容器的替代；
- 几种标准的非STL容器：数组、bitset、valarray、queue、priority_queue，它们不是STL容器。

连续内存容器：把元素存放于一块或多块内存中，每块内存有多个元素。vector、string、deque、rope。
基于节点的容器：修改元素时，元素本身不需要移动，改变的只是指针指向。list、slist。

- 任意位置插入元素？
  -- 序列容器，关联容器不可行；
- 不需要排序元素？
  -- 哈希容器；
- c++标准容器？
  -- 排除哈希容器、slist、rope；
- 随机访问迭代器？

-- vector、deque、string；

- 双向迭代器？
  -- 避免slist、哈希容器；
- 避免元素移动？
  -- 避免连续内存容器；
- 容器中数据布局和C兼容？
  -- vector；
- 追求元素查找速度？
  -- 哈希容器、排序的vector和标准关联容器；
- 容器内部使用了引用计数技术？
  -- 介意则避免string，用vector<char>代替；
- 插入或删除失败，提供回滚能力（事务语义）？
  -- 使用list
- 需要迭代器尽量安全？
  -- 基于节点的容器；
- 随机访问、插入只在末尾且没删除操作？
  -- deque

2.不要试图编写独立于容器类型的代码

意思就是说，不要试图编写适合各种容器类型的代码。因为每一种容器的操作都不一样，不可能能够形成一个好的交集。

- 使用typedef来封装可能要改的容器

```
typedef vector<Widget> WidgetContainer;
typedef WidgetContainer::iterator WCIterator;
```

要想减少在替换容器类型时所修改的代码量，可以把容器隐藏到一个类中，并尽量减少那些与特定容器有关的信息。

```
class CustomerList
{
```

```
private:
    typedef list<Customer> CustomerContainer;
    typedef CustomerContainer::iterator CCIterator;
public:
    ....
};
```

3.确保容器中的对象拷贝正确而且高效

> insert或push_back之类的操作向容器中加入对象时，存入容器中的是所指定对象的拷贝。front或back之类的操作从容器中取出一个对象时，取出的也是容器对象的拷贝。这就是STL的工作方式。
>
> 使拷贝动作高校正确的办法就是让容器包含指针而不是对象，就是说使用Widget*的容器。使用指针带来的问题是内存泄漏，因此可以使用智能指针来做改善。

4.调用empty()而不是size()==0？

> empty()形式对所有的标准容器都是常数时间操作，而对一些list实现，size()耗费线性时间。因为list的内部实现中，为了各操作的效率尽量提高，没有内置size数据。

5.区间成员函数优先于与之对应的单元素成员函数

> assign为区间成员函数，所有标准序列容器都包含。对copy的调用效率会偏低。因为copy是逐个插入，而assign是考虑整体后插入。

- 区间创建

```
container::container(InputIterator begin, InputIterator end);
```

- 区间插入

```
void container::insert(iterator position, InputIterator begin,
InputIterator end);
```

- 区间删除

```
iterator container::erase(iterator begin, iterator end);  //序列容器
void container::erase(iterator begin, iterator end);  //关联容器
```

- 区间赋值

```
void container::assign(InputIterator begin, InputIterator end);
```

6.当心C++编译器的分析机制

```
ifstream dataFile("ints.dat");
list<int> data(istream_iterator<int>(dataFile), istream_iterator<int>(
));
//这段代码不会把data当作一个数据对象，而是把data()当成了一个函数的声明。为了避免这种情况，C++11建议使用"{}"来表示数据的构造。
```

7.如果容器中包含了通过new操作创建的指针，在容器对象析构前将指针delete掉。

> 为了避免内存泄漏，尽量使用智能指针。shared_ptr。

```
//如果不使用智能指针，最大程度避免内存泄漏的方法
struct DeleteObject
{
    template <typename T>
    void operator()(T const* ptr) const
    {
```

```
            delete ptr;
        }
    }

    for_each(xx.begin(), xx.end(), DeleteObject());
```

8.不要创建包含auto_ptr的容器对象

> auto_prt是一种有缺陷的智能指针，当它进行赋值或拷贝操作的时候会存在很多问题，例如右操作数指针会被置空，当再需要调用时会出错。

9.慎重选择删除元素的方法

```
//对于连续内存容器，
//remove()是algorithm库的泛型算法，找到元素并把元素后的所有元素向前移动，最后返回指向最末端的地址；
//erase()删除一个迭代器区域的元素，返回指向下一个元素的指针。
std::Container<int> c;
container.erase(remove(c.begin(),c.end(),199), c.end());
container.erase(remove_if(c.begin(),c.end(),func), c.end());

//对于list的话,效率更高的做法：
c.erase(199);
c.remove_if(...,...,...);
```

```
//remove()对于set、multiset、map、multimap的操作都是错误的。
//对于关联容器，应该调用erase()，返回值是1或者0。
std::map<int, int> container_map;
container_map.earse(111);
//关联容器的条件删除操作
AssocContainer<int> c;
for(auto i = c.begin();i!=c.end())
{
    if(badValue(*i))
        c.erase(i++);    //防止变野指针
```

```
        else ++i;
    }
```

10.了解分配子的约定(allocator)和限制

> 相同类型的allocator<int>是完全等价的。这意味着allocator这类不允许有非静态成员的存在。所以它们共同维护一个静态类的堆。

- new()和allocator::allocate()的区别

```
//分配一个int的对象
new(4);     //输入具体字节数,输出空指针；
//或者
allocator<int>::allocate(1);    //输入对象数，输出T*指针；
```

> 基于节点的容器的内部实现都没有使用allocator，这是因为分配新对象时，分配的是它们的节点，而不是值T，所以有个rebind()函数，重新指定allocator的类型。

11.理解自定义分配子的合理用法

暂时跳过，回头再看。

12.不要对STL容器的线性安全性有不切实际的依赖

> - STL容器中多个线程读是安全的；
> - STL容器中多个线程对不同的容器做写入操作是安全的。

13.vector和string 优先于动态分配的数组

> vector和string能够自己管理内存，取决于它们的构造函数以及析构函数。string通常使用了引用计数的功能。

- 如果多线程环境下，string的引用计数影响效率的话可以采用以下方法：
  1. 尝试库中禁用引用计数；
  2. 寻找或开发另一个不使用引用技术的string；
  3. 考虑使用vector。

14.使用reserve()避免不必要的重新分配

vector和string提供的四个函数解释：

- size():该容器中有多少个元素；
- capacity():该容器已经分配的内存可以容纳多少个元素；
- resize():强迫容器改变到包含n个元素的状态；
- reserve():强迫容器把它的容量变为至少n；

15.注意string实现的多样性

> string的实现方法很多样化，使用sizeof(string)的时候，可能会出现不同的值。

16.把vector和string数据传歌旧的API

```
1.  //对于vector，安全调用的方式有：
2.  if(!v.empty())
```

```
{
    doSomething(&v[0], v.size());
}
//千万不要试图直接使用v.begin(),它返回的是一个迭代器而不是一个指针，只有&*v.begin()才是指针。

//对于string，安全调用的方式有：
doSomething(s.cstr());

//把c数组写入vector
reserve -> copy

//把cstr写入string(list、deque、set)
cstr -> vector<char> ->string
```

17.使用swap()除去多余的容量

```
//先创建一个临时量拷贝v，再交换v的值。对于string也适用。不一定能除去多余的容量，因为容器的内部实现可能会保留一个最小阈值。
vector<int>(myVector).swap(myVector);
//swap()操作不仅交换容器的内容，同时它们的迭代器、指针和引用也会被交换。所以原来指向某个元素的迭代器、指针和引用依然有用。
```

18.避免使用vector

> vector是一个假的容器，它并不真的储存bool。为了节省空间，它储存的是bool的紧凑表示，一个字节可以保存八位，也就是可以保存8个"bool"，而指针和引用所能访问的最小单位是字节。所以vector内部实现中，使用了代理类来代替[]操作符的返回类型。所以bool *pb=&v[0]；会出错。

- 必须使用vector<bool>时的替代：
  1. deque代替，它真正储存bool；
  2. 如果不需要迭代器和动态改变大小，则使用bitset。

19.理解相等(equality)和等价(equivalence)的区别

```
//相等：
operator==(const T& lhs, const T& rhs);
lhs == rhs;

//等价：
opeartor<(const T& lhs, const  T& rhs);
lhs < rhs && rhs < lhs;

//"persephone"与"Persephone"在不区分大小写的set中是find()到，等价的；
//但在泛型算法std::find()中并不相等；
//因此优先使用容器提供的成员函数find()。
```

20.为包含指针的关联容器指定比较类型

```
struct StringPtrLess:public binary_function<const string*, const string
*, bool>
{
    bool operator()(const string *ps1, const string *ps2) const
    {
        return *ps1 < *ps2;
    }
}

//可以为此创建一个通用的模板
struct DereferenceLess
{
    template <typename T>
    bool operator()(T pT1, T pT2) const
    {
        return *pT1 < *pT2;
    }
}
```

21.总是让比较函数在等值情况下返回false

> 如果不让等值的情况下返回false，则关联容器的行为会变得很怪异。例如equal_range(begin(),end(), value)使用的就是comp()函数。

22.不要直接修改set或multiset中的键(key)

> map中的key是const的，而set的key是非const，因为set中的key可能是一个对象，需要修改其中的属性。所以除了set被绑定的key值不要改，其它属性都可以改。

- 如果遇到某些set的实现不能改值，则可以通过const_cast把const属性去掉。千万不能使用static_cast，因为这实际上只是创建了一个临时对象来作修改。
- 如果为了遵循类型安全和可移植性，则把map或者set的值拷贝出来修改后，再把map或set中的旧值删除，添加新的拷贝修改值。

23.考虑用排序的vector代替关联容器

> 节省空间，关联容器的内部数据至少三个指针，vector则省空间很多。
> vector内存中连续，意味着读取会快。4k内存页读取。
> 如果查找和插入删除等操作混合在一起了，vector为了保持有序状态，代价会很大。

24.效率很重要的时候，谨慎选择map::operator与map::insert()

> 单纯的插入:使用m.insert(k, v);
> 更新值:使用m[k] = v;

```
1.  //使用operator[]()时,
```

```
typedef map<int, Widget> IntWidgetMap;
IntWidgetMap m;

m[1] = 1.50;
//实际上等同于
pair<IntWidgeMap::iterator, bool> result =
m.insert(IntWidgetMap::value_type(1, Widget());
result.first->second = 1.50;
//相当于先用insert了键值和一个默认Widget对象，然后再对Widget对象进行赋值。

//更加有效率的做法应该是：
m.insert(IntWidgetMap::value_type(1,1.50));
//所以，当作为添加的功能时，insert()的效率比operator[]要高很多。

//但是当一个等价的键值已经存在map中的时候，operator[]的效率更高：
m[k] = v;
m.insert(IntWidgetMap::value_type(k,v)).first->second = v;
//因为如果使用insert的话，value_type实际上就是pair<int,Widget>类型会有多余的构造函数。

//可以自己设计一个高效插入更新函数：
template <typename MapType, typename KeyArgType, typename ValueArgType>
typename MapType::iterator efficientAddOrUpdate(Maptype& m, const KeyArgType& k, const ValueArgType& v)
{
    typename MapType::iterator lb = m.lower_bound(k);
    if(lb != m.end() && !(m.key_comp()(k, lb->first)))
    {
        lb->second = v;
        return lb;
    }
    else
    {
        typedef typename MapType::value_type MVT;
        return m.insert(MVT(k, v));
    }
}
```

25.熟悉非标准的哈希容器

> 不知道有什么好写的。就是创建一个函数对象作为hash_map的模板实参。

26.iterator优先于const_iterator、reverse_iterator以及const_reverse_iterator

> 隐式转换:
> iterator -> const_iterator
> iterator -> reverse_iterator
> reverse_iterator -> const_reverse_iterator
>
> 显示转换:
> reverse_iterator.base() -> iterator
> const_reverse_iterator.base() -> const_iterator

27.使用distance和advance将容器的const_iterator转换成iterator

> 使用const_cast是不能把迭代器的const属性去掉的，会产生编译错误。因为iterator和const_iterator是完全不同的两种类型。

```
//advance(),distance()位于algorithm库
//使用advance以及distance把const_iterator转换为iterator
typedef vector<int>::iterator IntItr;
typedef vector<int>::const_iterator IntCItr;
vector<int> v;
//现在有一个const_iterator迭代器指向尾部
IntCItr const_itr = v.end()-1;
//使用advance以及distance去掉const属性
IntItr itr = v.begin();
advance(itr, distance(static_cast<IntCItr>(itr), const_itr));
```

28.正确理解reverse_iterator的显示转换base()所生成的iterator的用法

```
//reverse_iterator的base()函数转换为iterator类型的时候，会向后偏移一位，正好对应它们之间的结构。可以这么理解，reverse_iterator.base()转换后的行为与reverse插入的行为是一样的，所以才有这种偏移。
```

29.对于逐个字符的输入考虑使用istreambuf_iterator

> 流的缓存区是一个简单的概念，必须记住。

```
//一个效率不高的插入方法，istream_iterator内部使用的operator>>函数实际上执行了格式化操作：
ifstream inputFile("Data.txt");
inputFile.unsetf(ios::skipws);
string fileData(istream_iterator<char>(inputFile), istream_iterator<char>());
//高效做法，取消了格式化操作：
string
fileData(istreambuf_iterator<char>(inputFile),istreambuf_iterator<char>());
```

30.确保目标区间足够大

> 主要是使用transform算法以及使用插入迭代器以免发生不确定的行为。

```
//正确使用插入迭代器：
vector<int> values;
vector<int> results;
...
results.reserve(results.size()+values.size());
transform(values.begin(),values.end(), back_insert(results),
```

```
transmogrify);
//back_insert,front_insert,insert;在迭代器的前面插入。
```

31.了解各种与排序有关的选择

> 不稳定排序算法:partial_sort,nth_element,sort；
> 稳定排序算法:stable_sort。
>
> 上面的算法都要求随机访问迭代器，所以list专门包含了sort成员函数而且是稳定的。

```
//局部排序算法,放到最前面：
partial_sort(v.begin(),v.begin()+20,v.end(),compareFunc);
//局部符合条件算法，和上面一样，不过不排序：
nth_element(v.begin(),v.begin()+19,v.end(),compareFunc);

//局部找出大于某个值的区间
vector<Widget>::iterator goodEnd = partition(v.begin(),v.end(), hasGood
Quality);
```

32.如果确实需要删除元素，则需要在remove这一类算法之后再调用erase

> remove(),remove_if(),unique()行为一致；
>
> remove()只是把指定元素移到最后端并且把后面的元素向前移动，返回指向元素最后所在的位置的迭代器。所以需要使用erase()进行删除。而list的remove成员函数就真正的删除了元素。
>
> remove()之所以这样设计是因为它接受的是迭代器参数，它推断不出容器的类型，因此它不能直接就删除元素，只能让容器自己调用删除函数。

33.对包含指针的容器使用remove这一类算法时要特别小心

> 如果是智能指针，不必担心内存泄漏。否则必须先用for_each遍历所有指针找出符合条件的指针并且delete指针后把指针置为nullptr。最后再利用erase_remove把nullptr给删除。

```
//mem_fun(在functional库)
class Test
{
public:
    double func(int a){return 3.0;}
}
auto k = mem_fun(&Test::func);
//k(Test*, int);
```

34.了解哪些算法要求使用排序的区间作为参数

> 并非所有的算法都可以应用于任何区间。remove算法要求单向迭代器并且要求这些迭代器能够向容器中的对象赋值。所以remove不能应用于map、set等容器。同理，很多排序算法要求随机访问迭代器，所以对于list的元素不能调用这些算法。

- 要求排序区间的STL算法：
  binary_search、lower_bound、upper_bound、equal_range。(使用了二分查找，承诺对数时间)
  set_union、set_intersection、set_difference、set_symmetric_difference。（承诺线性时间）
  merge、inplace_merge。(归并合并，承诺线性时间)
  includes。(判断一个区间的元素是否都在另一个区间)
- 通常情况下与排序区间一起使用:
  unique(删除每一组连续相等的元素,仅保留第一个)、unique_copy。(unique的行为和remove的一样，并没有真正删除元素。)

35.通过mismatch或lexicographical_compare实现简单的忽略大小写的字符串比较

```
//lexicographical_compare相当于strcmp的一个泛化版本，它支持自定义比较函数。
lexicographical_compare(s1.begin(), s1.end(), s2.begin(),
s2.end(),func);

//如果不考虑移植性,可以：
stricmp(s1.c_str(),s2.c_str());
```

36.理解copy_if算法的正确实现

> STL泛型算法中没有copy_if算法，具体实现该书有讲。暂时跳过。

- 包含"copy"的算法：
  1. copy
  2. replace_copy(旧值换新值)
  3. replace_copy_if
  4. remove_copy
  5. remove_copy_if
  6. uninitialized_copy
  7. copy_backward
  8. reverse_copy
  9. unique_copy
  10. rotate_copy(循环复制)
  11. partial_sort_copy

##37. 使用accumulate或者for_each进行区间统计

> count()返回区间中有多少元素，count_if()返回满足条件的个数。区间中最小值最大值可由min_element、max_element获取。

> accumulate不允许传入的函数对象有副作用，而for_each则允许。所谓副作用其实就是函数对象里面附带了成员数据。accumulate直接返回数值，for_each返回函数对象。

```
 //累计算法accumulate位于<numeric>库,只要求输入迭代器。
 //返回总和:
 accumulate(v.begin(),v.end(),initialValue);

 //通用版本：
 //统计当前字符串总和函数：
 string::size_type stringLengthSum(string::size_type sumSoFar, const string& s)
 {
     return sumSoFar + s.size();
 }

accumulate(ss.begin(),ss.end(),static_cast<string::size_type>(0),stringLengthSum);
```

---

38. 遵循按值传递的原则设计函数子类

> 如果函数对象中具有大量的数据，那么按值传递的函数对象会显得效率很低。如果函数对象中含有虚函数，那么按值传递的函数对象可能会把继承类给隐式转换为基类。为了同时解决这两个问题，可以采用一种很巧妙的方法。把函数对象的真正实现存进一个指针中。

```
//设计真正的实现类
template<typename T>
class BPFCImpl:public unary_function<T, void>
{
private:
    Widget w;
    int x;
    ....
    ....
    int z;  //原来类中大量的数据的存放在这里

    virtual ~BPFCImpl();
    virtual void operator()(const T& val) const;
```

```
14.
15.   friend class BPFC<T>;
16.   }
17.
18.   //用于值传递的类,生成的指针可以用sharted_ptr管理
19.   template <typename T>
20.   class BPFC:public unary_function<T, void>
21.   {
22.   private:
23.       BPFCImpl<T> *pImpl;
24.   public:
25.       void operator()(const T& val) const
26.       {
27.           pImpl->operator()(val);
28.       }
29.   }
```

39. 确保判别式是“纯函数”

> 纯函数的概念就是，返回的结果只和参数值或者常量有关。对于作为判别式的函数对象，有些STL算法可能会先创建函数子的拷贝，然后存放起来以后再用。简单点说就是，有些STL算法会把你传进去的函数对象拷贝后重复调用好几次，最后出现不可预料的结果。因此为了必须存入干净的纯函数，以防出现不必要的行为。

40.若一个类是函数对象，则应该使它可配接

> 像not1(),not2(),bind1st(),bind2nd()这些函数，它们接受的参数类型都是func类，因此不能直接传入一个函数指针，必须先把函数指针通过ptr_fun()函数转换为func类。这个func类包含特殊的类型定义有：argument_type、first_argument_type、second_argument_type、result_type。
>
> 所以自己定义的函数对象如果要适配这些not1()之类的函数的话，必须继承unary_function或者binary_function。

```
//一元函数对象类
template <typename T>
class ChangeValue:public unary_function<T, double>
{
public:
    double operator()(T){return 3.0;}
}

//二元函数对象类
template <typename T>
class CompareValue:public binary_function<T, T, double>
{
public:
    double operator()(T, T){return 3.1;}
}

//传递给unary_function或binary_function的模板实参，如果是非指针类型，需要去掉const和引用属性。
//对于const T&类型：
struct WidgetNameComp: public unary_function<string, void>
{
public:
    void opearator()(const string& s)
    {...}
}
//对于const T*类型：
struct WidgetNameComp: public unary_function<const string*, void>
{
public:
    void opearator()(const string *s)
    {...}
}
```

41.理解ptr_fun、mem_fun和mem_fun_ref的来由

```
//代码解释最为清晰
class MyClass
{
public:
    void foo(){}
}
```

```
vector<MyClass> vmc;
for_each(vmc.begin(), vmc.end(), mem_fun_ref(&MyClass::foo));

vector<MyClass*> vpmc;
for_each(vpmc.begin(), vpmc.end(), mem_fun(&MyClass::foo));
```

42.确保less与operator<具有相同的语义

> 默认情况下对less的调用会直接调用operator<()，所以应该保留这两个函数的一致性。

43.算法调用优先于手写的循环

> 调用算法往往优先于手写循环。算法通常比手写循环效率更高，手写循环更容易出错，算法代码比手写循环更简洁明了。
>
> STL中有70个算法名称，考虑重载的情形，大约有100个不同的函数模板。

44.容器的成员函数优先于同名的算法

> 对于关联容器成员函数的效率远远高于同名算法。list的成员函数remove()是真正的删除了元素，不需要再调用erase()。

45.正确区分count、find、binary_search、lower_bound、upper_bound和equal_range

```
//如果迭代器指定的区间是排序(等价性)的：
```

```
binary_search();
lower_bound();//返回第一个等价元素迭代器或者能插入该元素的指针，所以返回值还要判断是否和查找值等价。
upper_bound();//返回最后一个等价元素后一个的迭代器
equal_range();//更多使用这个代替lower，因为返回一对迭代器相等的话，则没找到该元素。

//如果迭代器是未排序的，只能用以下的(相等性)：
count();
count_if();
find();
find_if();
```

46.考虑使用函数对象而不是函数作为STL算法的参数。

> 使用高级语言编写程序的一个特点，随着抽象程度的提高，所生成的代码效率会降低。
>
> C++的sort()算法效率要高于C的qsort(),因为sort如果传入一个函数对象的话，该函数会内联进sort算法。而qsort()算法传的是函数指针，函数指针会抑制内联函数机制。也就是说qsort会重复调用该函数。

```
//当要取一个迭代器指向的数据类型时，使用：
iterator_traits<xxxiterator>::value_type;
//使用iterator_traits类包装，可以兼顾内置类型int*等。
```

47.避免产生“直写型”代码

> 意思就是说不要把一条语句堆个十来个函数调用，使用算法的时候尽量分层写。

48.总是include正确的头文件

- 几乎所有的STL容器都被声明在同名的头文件中
- 所有泛型算法都被声明在中，除了accumulate、inner_product、adjacent_difference和partial_sum被声明在头文件中。
- 特殊类型的迭代器，包括istream_iterator和istreambuf_iterator被声明在了中。
- 标准的函数对象`less<T>`等和函数对象配接器`not1、bind2nd`等被声明在了

49.学会分析与STL相关的编译器诊断信息

> 不知道该说什么，多看模板错误信息吧

50.熟悉与STL相关的Web网站

- SGI STL:
  http://www.sgi.com/tech/stl/
- STLport:
  http://www.stlport.org/
- Boost:
  http://www.boost.org/

地域性问题

```
//告诉C库，以后的字符操作根据xx语言习惯处理
setlocale(LC_ALL, "de"); //德语

//c++做法
std::locale L("de");

const std::ctype<char>& ct = std::use_facet<std::ctype<char> >(L);
bool result = ct.toupper(c1)<ct.toupper(c2);
//或则可以直接:
```

```
10.   std::toupper(c,L);
```