**标准模板库**

**容器方法 ：**

**非成员函数 :** 位于 **algorithm** 中

**for\_each ()**

功能 ： 将指向的函数应用与容器区间的各个元素 ；

要求 ： 被指向的函数不能修改容器中的值 ；

例如：for\_each（vice.begin() , vice.end() , jin） ;

**random\_shuffle ()**

功能： 随机排列区间内容 ；

要求： 容器允许随机访问 ；

例如： random\_shuffle (vice.begin() , vice.end())

**sort ()**

**版本一 ：**  接受两个区间的迭代器参数 ， 对内置内容使用 < 排序 。

要求 : 容器元素必须定义能够处理该类型对象的 operator < ( )；

例如 ：sort (vice.begin() , vice.end()) ;

**版本二 ：** 接受两个区间的迭代器参数 ， 第三个参数指向要使用的函数指针 ； （可完成降序排列）

例如： sort (vice.begin() , vice.end() , jin ) ;

**copy ()**

用于将一个容器中的内容复制到其他容器 ；

例如： copy （vice.begin() , vice,end() , first.begin()）;

**transform ()**

**版本一 ：** 接受四个参数的函数 ， 前两个参数指定了操作区间 ，第三个参数指出了将结果复制到哪个迭代器位置 ， 最后一个是函数符 指出了对每个函数的操作（一元函数）；

**版本二：** 接受五个参数 ，前两个指出第一个区间 ， 第三个指出了第二个区间的开始位置，第四个参数指出了将结果复制到哪个迭代器位置 ， 最后一个是函数符 指出了对每个函数的操作（二元函数）；

**迭代器 ：**

**特征 ：**

1. 能够对迭代器执行解除引用的操作 ， 以便能够访问他引用的值 ， 即如果 p 是一个迭代器 ， 则应对 \*p　进行定义 ；
2. 应能够将一个迭代器赋给另一个 ，即如果 p 和 q 都是迭代器 ， 则应对 p = q 进行定义 ；
3. 应能够使用迭代器便利容器中的所有元素 ， 这可以通过迭代器 p 定义 ++p 和p++ 来实现 ；
4. 应能够将一个迭代器和另一个迭代器进行比较 ， 看他们是否相等 。 即如果p和q都是迭代器则应对 p==q 和p!= q 进行定义 ；

**迭代器类型：**

1. **输入迭代器 ：**

将容器中的内容读入程序 ， 但不一定支持修改内容 ；

1. **输出迭代器：**

将程序中的内容写入程序 ， 但不一定可读 ；

1. **正向迭代器：**

包含输入，输出的基本内容 ，使用 ++i 或 i++ 来遍历容器 ；

1. **双向迭代器：**

支持双向访问迭代器 **；**

1. **随机访问迭代器**

**概念改进和模型 ：**

**基本迭代器：**

**begin ()：**返回指向容器第一个位置的迭代器 ;

**end ()** ：返回指向容器最后一个位置的迭代器 ；

**iterator** 正向迭代器

// 如： vector<int>::iterator

**预定义迭代器：**

**ostream\_iterator <> 模板 ：** // 输出迭代器

例如： copy (dice.begin () ，dice.end() ，**ostream\_iterator<int , char>(cout ,“ ”)**;

**istream\_iterator <> 模板 ：** //输入迭代器

例如： copy(istream\_iterator<int,char>(cin) , istream\_iterator<int,char>() , dice.begin())

**其他迭代器类型：**

**reverse\_iterator** ：执行递增操作时将导致他被递减 ，

功能 ： 主要用作反向迭代器 ；

**rbegin () :** 返回指向超尾的反向迭代器 ；

**rend () :** 返回指向第一个元素的反向迭代器 ；

**insert\_iterator<>( )** 将数据插入到指定位置处 ；

例如： insert\_iterator <vector<int> >insert\_iter (dice , begin); **back\_insert\_iterator <>( ) :**将元素插入到容器尾 ；

例如： back\_insert\_iterator <vecter<int> > back\_iter (dices) ;

**front\_insert\_iterator<> ( ) ;** 将元素插入到容器头 ；

注意：上面的迭代器主要用于将复制算法转化为插入算法 ；

**容器：**

**容器概念：** 具有名称的通用类别 ； 概念描述所有容器都通用的元素 ， 它是一个概念化的抽象基类—说它是**概念化**是因为容器类并不真正使用继承机制 ， （容器概念指出了所有 STL 容器类都必须满足的一系列要求） ；

**要求：**不能将任何类型的对象存储在容器中 ， 换句话说就是**类型**是可复制构造和可赋值的 ；

**容器类型：** 可用于创建具体容器对象的模板 ；

**类型 ：** deque , list , queue , priority\_queue , stack , vector , map ,multimap , set , multiset , bitset ;

**C++ 新增 :**  forward\_list , unordered\_map ,unordered\_multimap

unordered\_setunordered\_multiset

**基本容器特征 ：**

**X::iterator :** 指向T 的迭代器类型 ；

**X::value\_type ;** 迭代器所指向的容器中存储类型 ；

**size ()**返回容器中元素数目 ；

**swap（）** 交换两个容器中的内容 ;

**push\_back ()** 将元素添加到矢量末尾 （可增加矢量长度） ；

**序列 :**

序列概念增加了迭代器至少是正向迭代器这样的要求 ;要求其元素按严格的线性顺序进行排列，即存在第一个元素和最后一个元素，除第一个元素和最后一个元素以外其他元素前后都有元素（数组，链表）;

**序列容器**

deque , forward\_list , list , queue , priority\_queue , stack , vector

**序列要求 ：**

**X a (n ,t) ：** 声明长度为 n 值为t 的序列 ；

**X (n,t) :** 声明长度为 n 值为t 的匿名序列 ；

**X a(i , j) :** 创建一个a 序列 ，并将他初始化为区间 [i,j)之间的值；

**X (i , j) :（同上）；**

**a.insert (p , t) :** 将 t 插入到 p 的前面 ;

**a.insert (p , n , t) :** 将n 个t 插入到 p 前面 ；

**a.insert （p ，i，j ） :** 将区间[i,j)插入到 p 前面 ；

**a.erase (p) ：** 删除 p 指向的元素 ；

**a.erase (p,q) :** 删除区间[p , q)中的元素 ；

**a.clear () :** 等价与 a.erase (a.begin(),a.end())

**序列可选要求：**

**a.front () :** 返回容器中的第一个值 ； // vector list deque

**a.back () :** 返回容器中的最后一个值 ； // vector list deque

**a.push\_back(t)** :将t 插入到容器最后 ； // vector list deque

**a.push\_front(t):** 将t 插入到容器开始 // list deque

**a.pop\_back ( ) :** 删除容器最后内容// vector list deque

**a.pop\_front () :**删除容器开始值 ； // list deque

**a[n]**： 返回容器中第 n 个元素 // vector deque

**a**.**at(t):** (同上) // vector deque

注意： 他们的区别在于当 n 落在有效范围之外时 ， **a.at(t)** 会引发out\_of\_range 异常 ；

**序列容器类型：**

**vector<> :**

vector 是最简单的序列模型 ， 除非其他类型的特殊优点能够跟好的满足程序的要求 ， 否则应默认使用这种类型 ；

vector **支持随机访问** ， 提供了自动内存管理功能 ， 可以动态的改变 vector 的长度 ， 并随着元素的添加和删除而自动改变内存大小 ，在尾部的添加和删除元素是固定时间的 ， 而在首部时间是线性的；

**支持双向遍历矢量 ；**

除序列外 ， vector 还是可翻转序列的模型（rbegin() , rend() , reverse\_iterator） ；

**deque <>:**

双端队列 ，**在首部插入和删除元素的时间是固定时间** ， **支持随机访问** ，

**支持双向遍历队列**

**list<> :**

双向链表 ， 这说明 list 在任意位置进行增删元素的时间都是固定的

**不支持随机访问 ， 允许双向遍历链表 ；**

除序列和可翻转容器的函数外 ， list 模板类还包含了链表专用的成员函数 ，

**list 成员函数：**

**void merge (list <t ,Alloc> & x ) ：**

将链表 x 和调用链表合并 ， 两个链表必须已经排序 ， 合并后的经过排序的链表保存在调用链表中 ， x 将为空 ；

**void remove (const T & val) :**

从链表中删除 val 的所有实例 ，这个函数的复杂度为线性的 ；

**void remove\_if (**谓词参数**) ：**

该函数将位次参数用于容器中的每个成员， 如果返回 true ;则删除该元素 ；

**void sort () :**

使用 < 运算符对链表进行排序 ；

**void splice (iterator pos , list<T , Alloc> x )** 将链表 x 的内容插入到 pos 前面 ， x 将为空 **void unique ()**

将链表中的相同元素压缩为单个元素（只能将相邻的相同值进行压缩）

**注意：** sort , merge , unique , 方法还各自拥有接受另一个参数的版本 ， 该参数用于指定**用来比较元素的函数 ，** 同样remove() 方法也有一个接受另一个参数的版本 ， 该参数用于指定用来**确定是否删除元素的函数**

**queue<>:**

queue 模板让底层类（默认为deque）, 展示典型的队列接口 ；它**不允许随机访问队列元素** ， 甚至**不允许遍历队列** ；它允许**将元素添加到队尾 ， 从队首删除元素 ， 查看队首和队尾的值 ， 检查元素数目和测试队列是否为空 ；**

**queue 的操作 ：**

**bool empty ()const //** 如果队列为空 ，则返回 ture

**size\_tyoe size ()const //** 返回队列中的数目 ；

**T& front () //** 返回指向队首元素的引用

**T& back () //** 返回指向队尾元素的引用；

**void push(const T&x) //**在队尾插入 x

**void pop () //**  删除队首元素

**priority\_queue<> :**

它支持的操作与 queue 相同。两者之间的主要区别在于 ， 在priority\_queue 中 ， **最大元素被移到队首** ， 内部区别在与，默认的底层类是 vector . 可以修改用于确定哪个元素放在队首的比较方式 ，方法是提供一个可选的构造函数参数 ；

priority\_queue<int>pq1 ;

priority\_queue<int>pq2(greater<int> ) ;

允许使用方法与 stack 基本相同 ；

**stack<> :**

它不仅**不允许随机访问栈元素** ， 甚至**不允许遍历栈**

它允许 **压栈 ， 出栈 ， 查看栈顶元素 ，检查元素数目 ， 测试是否为空 ；**

**Stack 的操作 ：**

**bool empty ()const //** 如果栈为空 ，则返回 ture

**size\_tyoe size ()const //** 返回栈中的数目 ；

**void push(const T&x) //**压栈

**void pop () //**  出栈元素

**T& top () //** 返回指向栈顶元素的引用

**关联容器 ：**

关联容器将值与键联系在一起 ， 并使用键来查找值 ；

关联容器的优点在于 ，它提供了对元素的快速访问 ， 与序列类似 ， 关联容器也允许插入新元素 ， 但不能指定插入位置 ， 原因是**关联容器通常有用于确定数据放置位置的算法 ，以便能够快速检索信息**

关联容器通常是使用某种树实现的 ；

**关联容器： set , multiset , map , multimap ;**

**set<>:**

其值类型与键相同 ， 键是唯一的 ，值就是键 ， 所以不能存储多个值 ；

他是关联集合 ， 可反转 ， 自动排序 ；

**set** 接受两个参数 ，第一个参数指出值类型 ， 第二个参数是可选的，可用于指示用来对键进行排序的比较函数或对象 。 默认情况下将使用 less<> ;

set<string> A <==>set <string , less<string> >A ;

**算法** ：

**set\_union ()** // 取并集

set接受5个参数 ， 前两个接受第一个集合区间 ， 接下来两个接受第二个区间集合 ，最后一个指出将结果复制到什么位置（**迭代器**） ；

如： set\_union (a.begin() , a.end(), b.begin() , b.end() , ostream\_iterator<string , char>(cout , “ ”) );

**set\_intersection ()** //取交集

**set\_difference() //** 去差集（返回位于第一个区间里的 不存在第二个里的元素）

**lower\_bound () // 返回第一个不小于键参的值**

**upper\_bound() // 返回第一个大于键参的值**

**multimap<>**

值与键类型不同 ， 每个键可以与多个值相关联 ；

multimap **<>**包含三个参数 ，**键类型 ， 值类型 ， 用于对键进行排序的比较函数或对象 ；**

如： multimap <int , string> <==> multimap <int , string less<int> > ;

为将信息结合在一起 ， 实际值类型将键类型和数据类型结合为一对 ， 为此**STL**使用模版类pail<class T , class U> 将这两个值存储在一个对象中：**方法如下：**

**multimap <int , string> cold ;**

**pair <nt , string>item(12,”hellow world”) ;**

**cold.insert (item );**

**方法：**

**count () // 将键作为参数 并返回改建的元素数目 ；**

**lower\_bound () //同 set**

**upper\_bound () //同set**

**equal\_range () //将键作为参数 ， 并返回两个迭代器 ， 他们表示的区间与该键匹配**

**函数对象 ：**

函数对象--函数符 ：是可以以**函数方式**与**（）**结合使用任意**对象**。这包括**函数名** 、**指向函数的指针**和**重载了（）运算符的类对象** (即定义了函数 operator () 的类)**;**

**函数符概念 ：**

**生成器** ：不用参数就可以调用的函数符 。

**一元函数** ： 用一个参数就可以调用的函数符 。

**二元函数** ： 用两个参数就可以调用的函数符 。

例如： 提供给 for\_each（）的就是一元函数 ， 因为它一次用于**一个容器元素**

**谓词** ： 返回 bool 值的一元函数 ；

**二元谓词** ： 返回bool 值的二元函数 ；

**预定义的函数符：**（它使得将函数对象用于参数很方便）

|  |  |
| --- | --- |
| **运算符** | **相应的函数符（模板）** |
| **+** | **plus** |
| **-** | **minus** |
| **\*** | **multiplies** |
| **/** | **divides** |
| **%** | **modulus** |
| **==** | **equal\_to** |
| **!=** | **not\_equal\_to** |
| **-** | **negate** |
| **>** | **greater** |
| **<** | **less** |
| **>=** | **greater\_equal** |
| **<=** | **less\_equal** |
| **&&** | **logical\_and** |
| **||** | **logical\_or** |
| **!** | **logical\_not** |

**自适应函数符和函数适配器 ：**

**binder1st :**

使用方式 ： 假设有一个自适应二元函数对象 f2 （）， 则可以创建一个binder1st 对象 ， 该该对象与一个将被用作f2()第一个参数的特定值（val）相关联 ： binder1st (f2,val)fi ;

**bind1st ：** 将简化这种方式 ， 可以这样 bind1st (plus<int> , 12) ;

**binder2nd :**与上类似 ， 只是将参数用于第二个值 ；