C++标准程序库读书笔记

CH5 STL容器

容器的共同能力

容器三个最核心的能力是：

1. 所有容器提供的都是value语意而非reference语意。
2. 总体而言，所有的元素形成一个次序。
3. 一般而言，各项操作并非绝对安全。

所有容器都提供了三个和大小相关的操作函数：

1. size()
2. empty()
3. max_size()

> 返回容器所能容纳的最大元素数量。

比较的定义依据以下三条规则：

1. 比较操作的两端（两个容器）必须属于同一型别。
2. 如果两个容器的所有元素依序相等，那么这两个容器相等。采用operator==检查元素是否相等。
3. 采用字典式顺序比较原则来判断某个容器是否小于另一个容器。

如果两个容器型别相同，而且拷贝后源容器不再被使用，那么可以使用一个简单的优化方法：swap()。

swap()的性能比较优异，因为它只交换容器的内部数据，事实上只交换某些内部指针，时间复杂度是常数，不像实际赋值操作的复杂度为线性。

vector

> vector模塑出一个动态数组，因此，它本身是将元素置于动态数组中加以管理的一个抽象概念。

```
namespace std{
    template <class T,class Allocator = allocator<T> >
    class vector;
}
```

> vector的元素可以是任意型别T，但必须具备assignable和copyable两个性质，第二个template参数可有可无，用来定义内存模型，缺省的内存模型是C++标准程序库提供的allocator。
>
> vector的迭代器是随机存取迭代器，vector支持随机存取。vector是一种有序群集，元素之间总是村咋某种顺序。vector将其元素复制到内部的dynamic array中。
>
> 在末端附加或删除元素时，vector的性能相当好，可是如果在前段或者中部安插或删除元素，***性能就不好***，因为操作点之后的每一个元素都必须移到另一个位置，而每一次移动都得调用assignment操作符。
>
> **vector优异性能的秘诀**，就是配置比其所容纳的元素所需更多的内存。

vector容量重要的原因

- 一旦内存重新配置，和vector相关的所有reference，pointers，iterators都会失效。
- 内存重新配置很耗时间。

避免vector重新分配内存的方法

- 使用reserve()保留适当容量。
- 初始化期间就像构造函数传递附加参数，构造出足够的空间

- 例如代码中就像构造函数传递附加参数，构造出足够的空间。

> 关于性能，以下情况可以预期安插操作和移除操作会比较快些：
>
> 1. 在容器尾部安插或移除元素。
> 2. 容量一开始就够大。
> 3. 安插多个元素时，调用一次当然比调用多次来的快。

安插元素和移除元素都会使作用点之后的各元素的reference，pointers，iterators失效。如果安插操作甚至引发内存重新分配，那么该容器身上的所有reference，pointers，iterators都会失效。

C++标准程序库并未明确要求vector的元素必须分布于连续空间中。

vector的异常处理

对于vector调用的函数抛出异常，C++标准程序库做出如下保证：

- 如果push_back()安插元素时发生异常，该函数不起作用。
- 如果元素的拷贝操作不跑出异常，那么insert()要么成功，要么不生效用。
- pop_back()绝不会抛出任何异常。
- 如果元素拷贝操作不抛出异常，erase()和clear()就不抛出异常。
- swap()不抛出异常。
- 如果元素拷贝操作绝对不会抛出异常，那么所有操作不是成功，就是不起作用。

Deques

deque也采用动态数组来管理元素，提供随机存取。不同的是deque的动态数组头尾都开放，因此可以在头尾两端进行快速安插和删除。

```
namespace std{
    template<class T,class Allocator = allocator<T> >
    class deque;
}
```

与vector相比，deques功能上的不同处在于：

- 两端都能快速安插元素和移除元素。这些操作都可以分期摊还的常数时间内完成。
- 存取元素时，deque的内部结构会多一个间接过程，所以元素的存取和迭代器的动作会稍稍慢一些。
- 迭代器需要在不同区块间跳转，所以必须是特殊的智能指针。
- 在对内存区块有所限制的系统中，deque可以内含更多元素，因为它是用不只一块内存，因此deque的max_size()可能更大。
- deque不支持对容量和内存重分配实际的控制，特别要注意的是，除了头尾两端，在任何地方安插和删除元素，都将导致指向deque元素的任何pointers。references，iterators失效。不过，deque的内存重分配优于vectors，因为其内部的结果显示，deque不必再内存重分配时复制所有元素。
- deque的内存区块不再被使用时，会被释放。deque的内存大小是可缩减的。

以下这几点与vector相似

- 在中间部分安插，移除元素的速度相对较慢，因为所有元素都需移动以腾出或填补空间。
- 迭代器属于随机存取迭代器。

以下情形，最好选用deque

- 需要在两端安插和移除元素。
- 无需引用容器内的元素。
- 要求容器释放不再使用的元素。

deque的各项操作中有以下几点和vector不同：

- deque不提供容量操作。
- deque直接提供函数，用以完成头部元素的安插和删除。

deque中需要注意的地方

deque中需要注意的地方

- 除了at()，没有任何成员函数会检查索引或迭代器是否有效。
- 元素的插入或删除可能导致内存重新分配，所以任何插入或者删除动作都会是所有指向deque元素的pointers，references和iterators失效。唯一的例外是在头部或者尾部插入元素，动作之后，pointers和references仍然有效，iterator依然会失效。

deque异常处理

deque的异常处理 与vector的一样。主要保证以下行为：

- 如果以push_back()或push_back()安插元素时发生异常，则该操作不带来任何效应。
- pop_back()和pop_front()不会抛出任何异常。

Lists

list使用了一个双向链表来管理元素。

```
namespace std{
    template <class T,class Allocator = allocator<T> >
    class list;
}
```

list在以下几个方面与vector和deque存在明显区别

- lists不支持随机存取。在list中随机遍历任意元素，是很缓慢的行为。
- 任何位置上执行元素的安插和移除都非常快。始终都是在常数时间内完成，因为无需移动任何其他元素，实际上内部只是进行了指针操作而已。
- 安插和删除动作并不会uzaocheng指向其他元素的各个pointers，references，iterators失效。
- lists对于异常有着这样的处理方式：要么操作成功，要么什么都不发生。

list所提供的成员函数反映出它和vector以及deque的不同

- 由于不支持随机存取，list既不提供下标操作符，也不提供at()
- list并未提供容量，空间重新分配等操作函数，因为全无必要，每个元素都有自己的内存，在被删除之前一直有效。
- list提供了不少特殊的成员函数，专门用于移动元素。

list的各种操作在异常发生时提供的特殊保证

操作	保证
push_back()	如果不成功，就无任何作用
push_front()	如果不成功，就无任何作用
insert()	如果不成功，就无任何作用
pop_back()	不抛出异常
pop_front()	不抛出异常
erase()	不抛出异常
clear()	不抛出异常
resize()	如果不成功，就无任何作用
remove()	只要元素比较操作不抛出异常，它就不抛出异常
remove_if()	只要判断式不抛出异常，它就不抛出异常
unique()	只要元素比较操作不抛出异常，它就不抛出异常
splice()	不抛出异常
merge()	只要元素比较时不抛出异常，它便保证要么不成功，要么无任何作用
reverse()	不抛出异常
swap()	不抛出异常

Sets Multisets

set和multiset会根据特定的排序准则，自动将元素排序，两者不同之处在于multiset允许重复而set不允许。

```
namespace std{
    template <class T,class Compare = less<T>,class Allocator = allocator<T> >
    class set;
    template <class T,class Compare = less<T>,class Allocator = allocator<T> >
    class multiset;
}
```

可有可无的第二个template参数用来定义排序准则。缺省的准则是less，这是一个仿函数。

所谓的排序准则，必须定义strict weak ordering,其意义如下：

- 必须是反对称的
- 必须是可传递的
- 必须是非自反的

排序准则也可用于相等性检验，如果两个元素都不小于对方，则两个元素为假。

set和multiset通常以平衡二叉树完成。

自动排序的优点与限制

自动排序的优点在于使二叉树于搜寻元素时具有良好性能，其搜寻函数算法具有对数复杂度。

> 自动排序造成set和multiset的一个限制：不能直接改变元素值，因为这样会打乱原本正确的顺序，因此要改变元素值，必须先删除旧元素，再插入新元素。

set提供的接口也反映了这种行为：

- set和multiset不提供用来直接存取元素的任何操作函数。
- 通过迭代器进行元素间接存取，有一个限制：从迭代器的角度来看，元素值是常数。

有两种方式能够定义排序规则

1. 以template参数定义

   > 这种情况下，排序准则就是型别的一部分，因此型别系统确保只有排序准则相同的容器才能被合并。

2. 以构造函数参数定义

   > 这种情况下，同一个型别可以运用不同的排序准则，而排序准则的初始值或者状态也可以不同，如果执行期才获得排序准则，而且需要用到不同的排序准则，可以用这种方式解决。

排序准则也可以用于元素相等性检验工作，当采用缺省排序准则时，两个元素的相等性检验语句如下：

```
if(!(elem1<elem2||elem2<elem1))
```

这样做有三点好处：

- 只需传递一个参数作为排序准则
- 不必针对元素型别提供operator==
- 可以对相等性有截然相反的定义

特殊的搜寻函数

采用set自带的搜寻函数，会获得对数复杂度，而非STL算法的线性复杂度。

迭代器相关函数

set和multiset不提供元素直接存取，所以只能采用迭代器。

> set和multiset不能用只能用于随机存取迭代器的STL算法。也也不能对set和multiset元素调用任何变动性算法。如果要移除set和multiset的元素，你

> 只能使用它们所提供的成员函数。
>
> STL惯例，你必须保证参数有效，迭代器必须指向有效位置，序列起点不能位于终点之后，不能从空容器中删除元素。

安插和移除多个元素时，单一调用比多次调用快得多。

set和multiset安插函数的返回值不相同：

- set提供以下接口：

```
pair<iterator,bool> insert{const value_type& elem};
iterator insert{iterator pos_hint,const value_type& elem};
```

- multiset提供以下接口：

```
iterator insert{const value_type& elem};
iterator insert{iterator pos_hint,const value_type& elem};
```

返回值不同的原因是

> multiset允许元素重复，而set不允许，因此如果将某元素安插至一个set内，而该set内含同值元素，则安插操作将告失败，所以set的返回值型别是以pair组织起来的两个值。

1. pair结构的second成员表示安插是否成功。
2. pair结构的first成员返回新元素的位置，或 返回现存的同值元素的位置。

其他任何情况下，函数都返回新元素的位置。

作用于序列式容器和关联式容器的erase()函数的返回值不同的原因是

1. 序列式容器提供下面的erase()成员函数：

```
iterator erase(iterator pos);
iterator erase(iterator neg,iterator end);
```

> 存在这种差别，完全是为了性能，在关联式容器中搜寻某些元素并返回后继元素可能颇为耗时，因为这种容器的底部是以二叉树完成，所以如果想要编写对所有容器都使用的程序代码，必须忽略返回值。