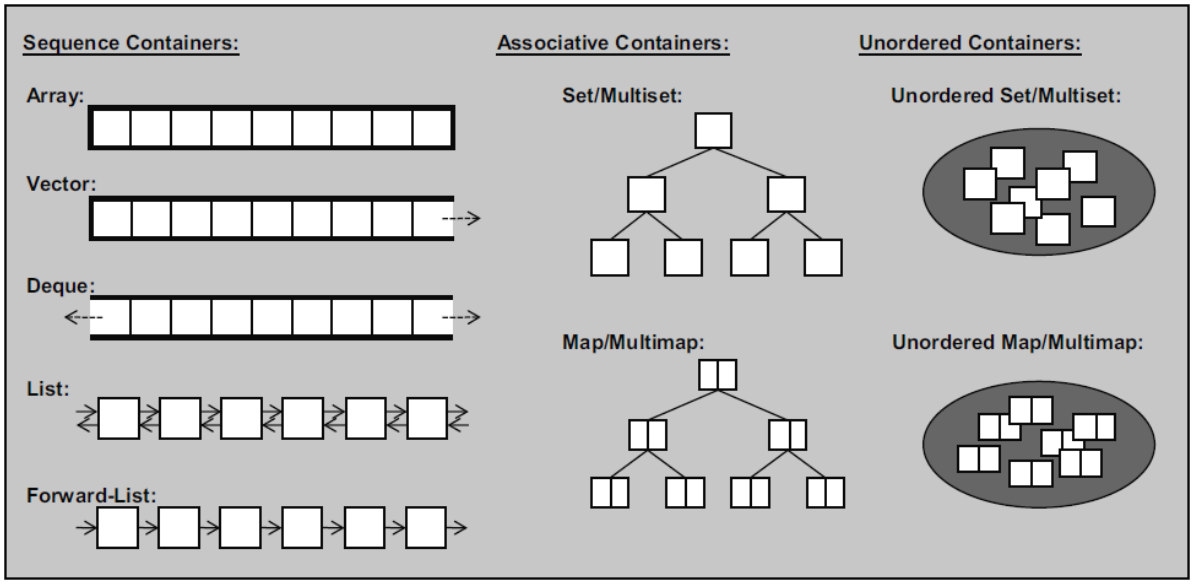
C++容器的特性与适用场景

容器类别

首先放上一张来自《C++标准库》中的图片。



序列式容器（Sequence container）

这是一种有序(ordered)集合，其内每个元素均有确凿的位置----取决于插入时机和地点，与元素值无关。如果你以追加方式对一个集和置入6个元素，他们的排列次序将和置入次序一致。STL提供了5个定义好的序列式容器：array、vector、deque、list和forward_list。

关联式容器(Associative container)

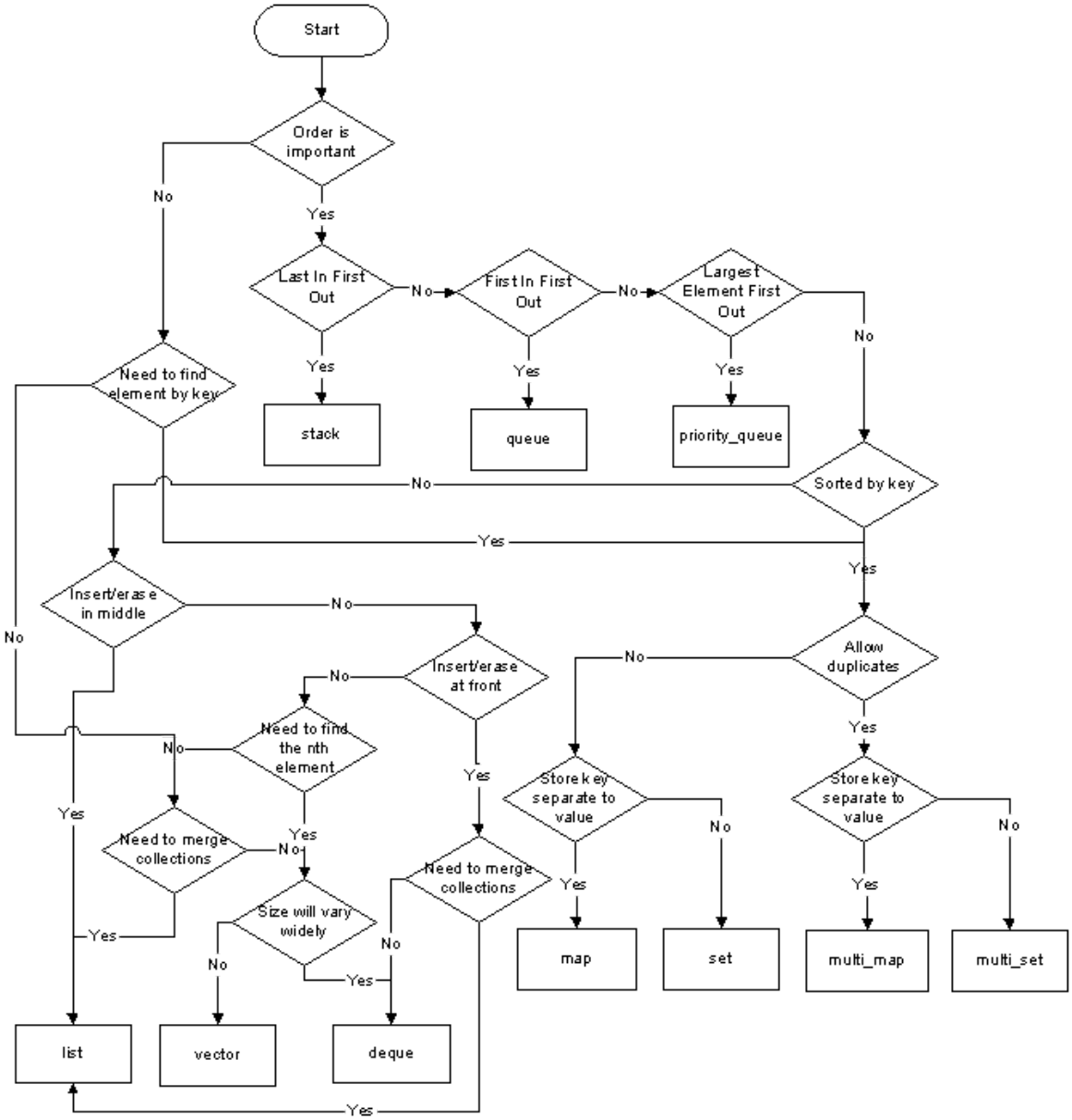
这是一种已排序(sorted)集合，元素位置取决于其value(或key----如果元素是个key/value pair)和给定的某个排序准则。如果将六个元素置入这样的集合中，他们的值将决定他们的次序，和插入次序无关。STL提供了4个关联式容器：set、multiset、map和multimap。

无序容器（Unordered (associative) container）

这是一种无序集合(unordered collection), 其内每个元素的每个位置无关紧要，唯一重要的是某特定元素是否位于此集合内。元素值或其安插顺序，都不影响元素的位置，而且元素的位置有可能在容器生命周期中被改变。如果你放6个元素到这种集合内，它们的次序不明确，并且可能随时间而改变。STL内含4个预定义的无序容器：unordered_set、unordered_multiset、unordered_map和unordered_multimap。

- **Sequence**容器通常被实现为array或linked list
- **Associative**容器通常被实现为binary tree
- **Unordered**容器通常被实现为hash table

各种容器使用时机



- 默认情况下应该使用`std::vector`。`std::vector`的内部构造最简单，并允许随机访问，所以数据的访问十分方便灵活，数据的处理也够快。
- 如果经常要在**序列头部和尾部安插和一处元素**，应该采用`std::deque`。如果你希望元素被移除时，容器能够自动缩减内部用量，那么也该使用`std::deque`。此外，由于`std::vector`通常采用一个内存区块来存放元素，而`std::deque`采用**多个区块**，所以后者可内含更多元素。
- 如果需要经常**在容器中段执行元素安插、移除和移动**，可考虑使用`std::list`。`std::list`提供特殊的成员函数，可在**常量时间内将元素从A容器转移到B容器。**但由于`std::list`**不支持随机访问**，所以如果只知道list的头部却要造访list的中端元素，效能会大打折扣。和所有"以节点为基础"的容器相似，**只要元素仍是容器的一部分，list就不会令指向那些元素的迭代器失效。**`std::vector`则不然，一旦超过其容量，它的所有`iterator`、`pointer`和`reference`失效。至于`std::deque`，当它的大小改变，所有`iterator`、`pointer`和`reference`都会失效。
- 如果你要的容器对异常处理使得"**每次操作若不成功便无任何作用**"，那么应该选用`std::list`(但是不调用其assignment操作符和sort(), 而且如果元素比较过程中会抛出异常，就不要调用merge()、remove()、remove_if()和unique()，或选用associative/unordered容器（但不调用多元素安插动作，而且**如果比较准则的复制/赋值动作可能抛出异常，就不要调用swap()或erase()**））。
- 如果你经常需要根据某个准则**查找元素**，应当使用"依据该准则进行hash"的`std::unordered_set`或`std::multiset`。然而，hash容器内是无序的，所以如果你必须以来元素的次序(order),应该使

用`std::set`或`std::multiset`，他们根据查找准则对元素排序。

- 如果想处理key/value pair，请采用`unordered_map`或`std::unordered_multimap`。如果元素次序很重要，可采用`std::map`或`std::multimap`。
- 如果需要关联式数组(associative array), 应采用unordered map。如果元素次序很重要，可采用`std::map`。
- 如果需要字典结构，应采用`unordered std::multimap`。如果元素次序很重要，可采用`std::multimap`。

	Array	Vector	Deque	List	Forward List	Associative Containers	Unordered Containers
Available since	TR1	C++98	C++98	C++98	C++11	C++98	TR1
Typical internal data structure	Static array	Dynamic array	Array of arrays	Doubly linked list	Singly linked list	Binary tree	Hash table
Element type	Value	Value	Value	Value	Value	Set: value Map: key/value	Set: value Map: key/value
Duplicates allowed	Yes	Yes	Yes	Yes	Yes	Only multiset or multimap	Only multiset or multimap
Iterator category	Random access	Random access	Random access	Bidirectional	Forward	Bidirectional (element/key constant)	Forward (element/key constant)
Growing/shrinking	Never	At one end	At both ends	Everywhere	Everywhere	Everywhere	Everywhere
Random access available	Yes	Yes	Yes	No	No	No	Almost
Search/find elements	Slow	Slow	Slow	Very slow	Very slow	Fast	Very fast
Inserting/removing invalidates iterators	—	On reallocation	Always	Never	Never	Never	On rehashing
Inserting/removing references, pointers	—	On reallocation	Always	Never	Never	Never	Never
Allows memory reservation	—	Yes	No	—	—	—	Yes (buckets)
Frees memory for removed elements	—	Only with `shrink_to_fit()`	Sometimes	Always	Always	Always	Sometimes
Transaction safe (success or no effect)	No	Push/pop at the end	Push/pop at the beginning and the end	All insertions and all erasures	All insertions and all erasures	Single-element insertions and all erasures if comparing doesn't throw	Single-element insertions and all erasures if hashing and comparing don't throw

	Array	Vector	Dequeue	List	Forward List	关联容器	无序容器
可用标准	TR1	C++98	C++98	C++11	C++98	C++98	TR1
数据结构	静态数组	动态数组	数组的数组	双向链表	单向链表	二叉树	哈希表
元素类型	value	value	value	value	value	set: value map: key/value	set: value map: key/value
是否允许重复	是	是	是	是	是	只有`multiset`和`multimap`允许重复	只有`multiset`和`multimap`允许重复

	Array	Vector	Dequeue	List	Forward List	关联容器	无序容器
迭代器类型	随机访问	随机访问	随机访问	双向迭代器	单向迭代器	双向迭代器	单向迭代器
增长/缩小方式	不会增长/缩小	在一端末尾增长	在两端末尾增长	到处增长	到处增长	到处增长	到处增长
是否可以随机访问	是	是	是	否	否	否	差不多
查找元素	慢	慢	慢	非常慢	非常慢	快	非常快
添加和删除操作是否会使迭代器无效	-	在重新申请内存时无效	总是无效	从不	从不	从不	在重新哈希时

	Array	Vector	Dequeue	List	Forward List	关联容器	无序容器
添加和删除操作是否会使引用或指针无效	-	在重新申请内存时无效	总是无效	从不	从不	从不	从不
是否允许保留内存	-	是	否	-	-	-	是
移除元素时释放内存	-	只有在`shrink_to_fit()`时释放内存	有时	总是	总是	总是	又是
事务安全(成功或没有影响)	No	在尾部`push`/`pop`事务安全	在头部和尾部`push`/`pop`安全	所有的插入和擦除	所有的插入和擦除	假如比较函数没有出现异常，那么单个元素的插入和所有擦除操作都是事务安全的	假如比较函数和哈希函数没有出现异常，那么单个元素的插入和所有擦除操作都是事务安全的

容器的共同能力

1. 所有容器提供的都是 "value 语义" 而非 "reference 语义"。容器进行元素的安插动作是，内部实施的是 `copy` 和/或 `move` 动作, 而不是管理元素的 `reference`。 如果不想要复制，那么只能使用 `std::move` 或 保存元素指针(不能使用引用来规避复制)。

2. 元素在容器内有其特定顺序。每一种容器会提供若干返回迭代器的操作函数，这些迭代器可以用来遍历各个元素。如果你在元素之间迭代多次，你会获得相同的次序(不调用增删函数)
3. 一般而言，各项操作并非绝对安全，也就是说他们不会检查每一个可能发生的错误。调用者必须确保传给操作函数的实参符合条件。
4. 都提供如下成员函数

函数	注解
default construct	
copy construct	
destructor	
begin()	
end()	
cbegin()	after C++11
cend()	after C++11
clear()	
swap()	std::array: O(n), 其他容器: O(1)
empty()	empty() 的实现可能比 size() == 0 更有效率，尽可能使用该函数
size()	
max_size()	
empty()	
operator==	
operator!=	
operator<	除了无序容器
operator<=	除了无序容器
operator>	除了无序容器
operator>=	除了无序容器

容器遍历方式

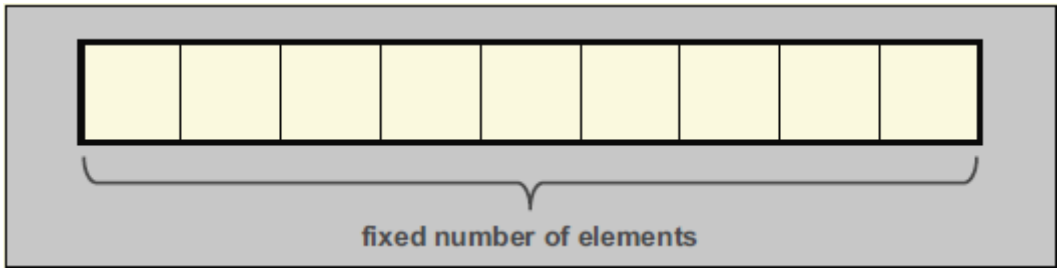
```
/// After C++
for(auto element : container)
{
    element;
}

/// 只对拥有随机访问迭代器的容器使用
for(size_t i = 0; i < container.size(); i++)
{
```

```
    container[i]/container.at(i);
}

/// 所有元素通用
for(auto it = container.begin(); it != container.end(); ++it)
{
    *it;
}
```

size() == 0 与 empty()

在 `C++11` 之前 `std::list::empty()` 函数的时间复杂度可能是 `O(n)` 也可能是 `O(1)`.

`std::array` C++11

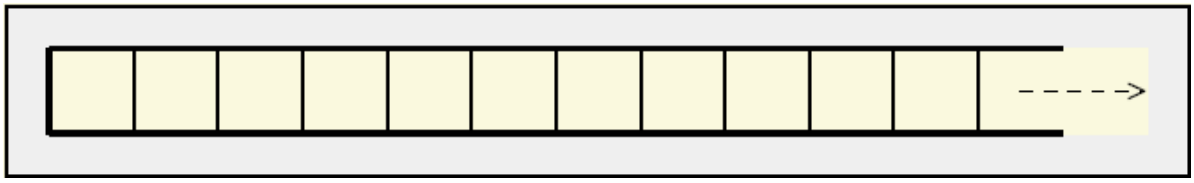


- 随机访问
- 固定大小, 编译期确定
- 大小可为零

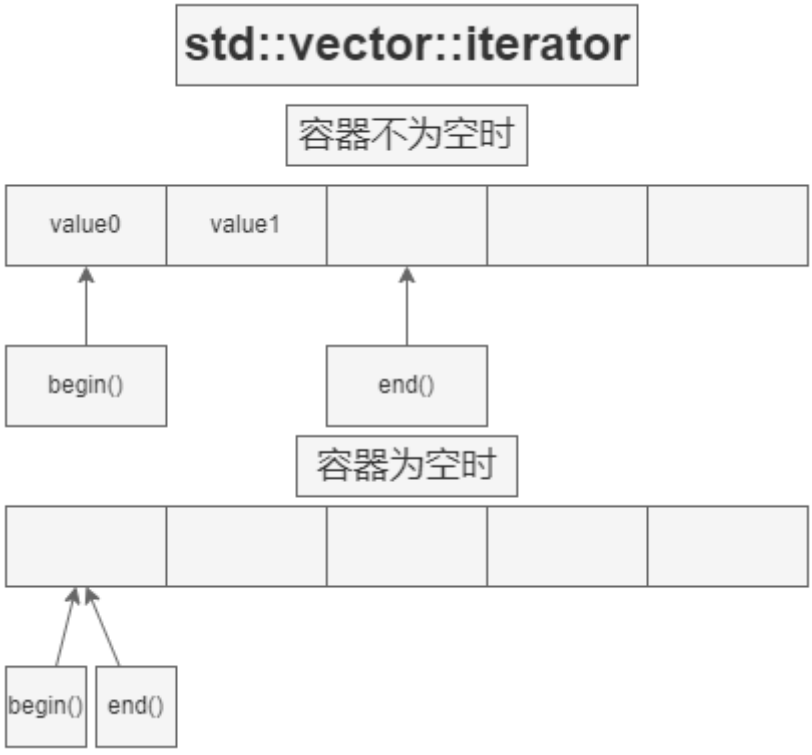
应用场景

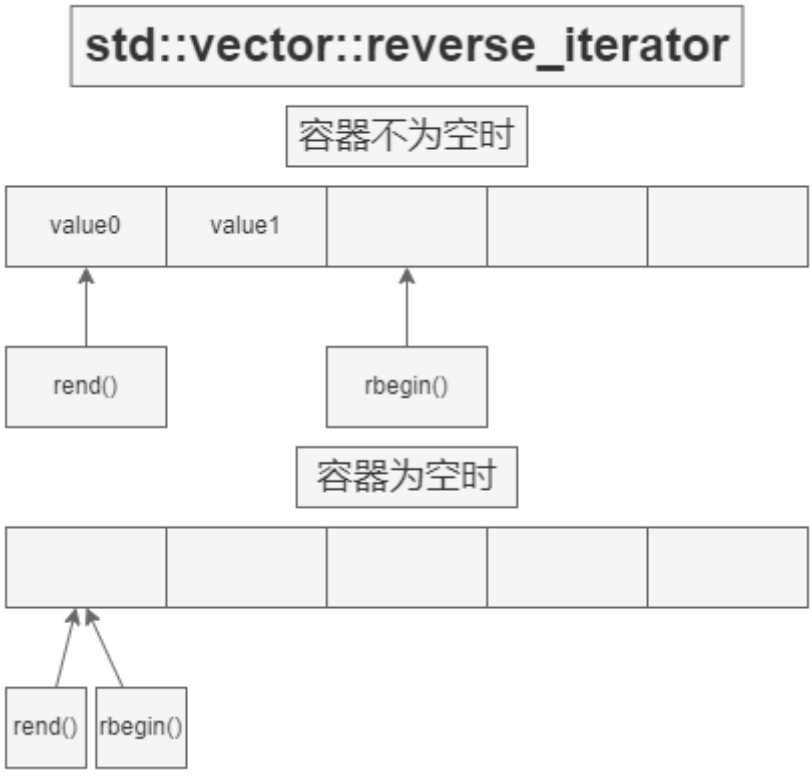
```cpp
/// 作为返回值返回
std::array<int, 5> f()
{
    std::array<int, 5> a;
    return a;
}

/// 作为入参时确定入参大小，不会降级为指针
void f1(const std::array<int, 5>& a)
{
    while(i < a.size()){}
    while(i < 5){}
}

/// 搭配模板灵活使用
template<size_t N>
void f2(const std::array<int, N>& a)
{
    while(i < a.size()){}
}

std::array<int, 6> a;
f2(a);
```

std::vector



迭代器示意





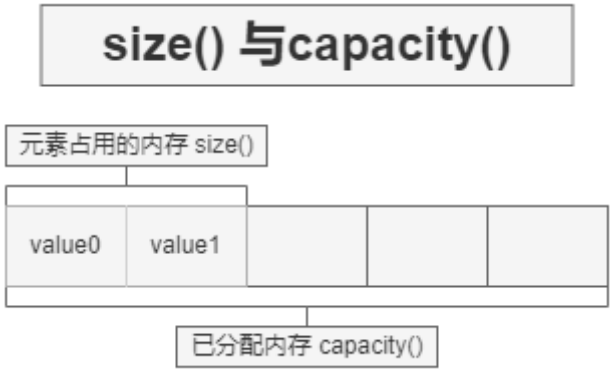
at() 与 operator[]

`at()` 成员函数提供边界检查，超出边界时会抛出异常 `std::out_of_range`

size() 与 capacity()

size() 查看当前有几个元素

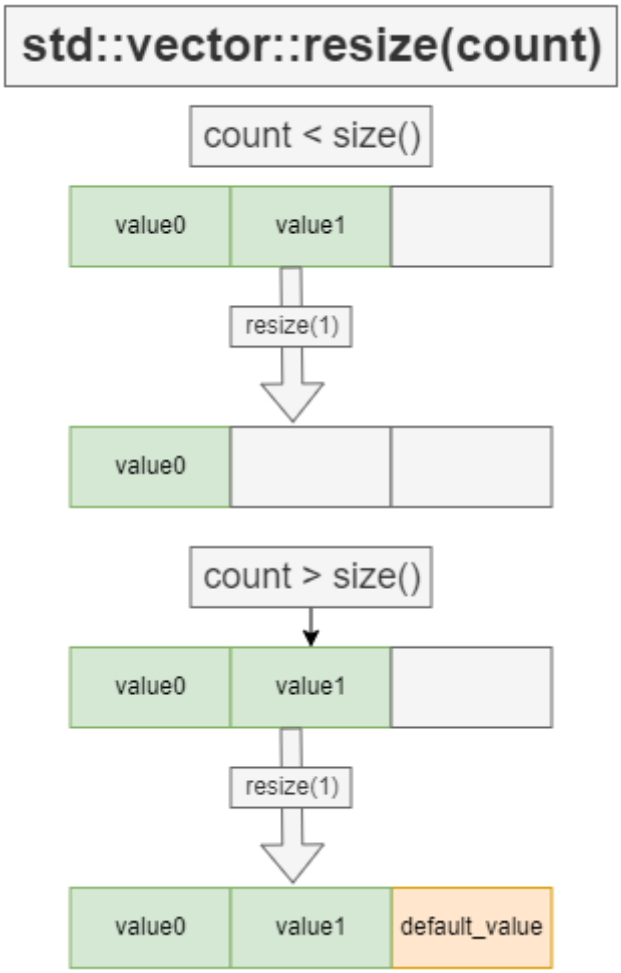
capacity() 查看预分配几个元素的空间



resize() 与 reserve()

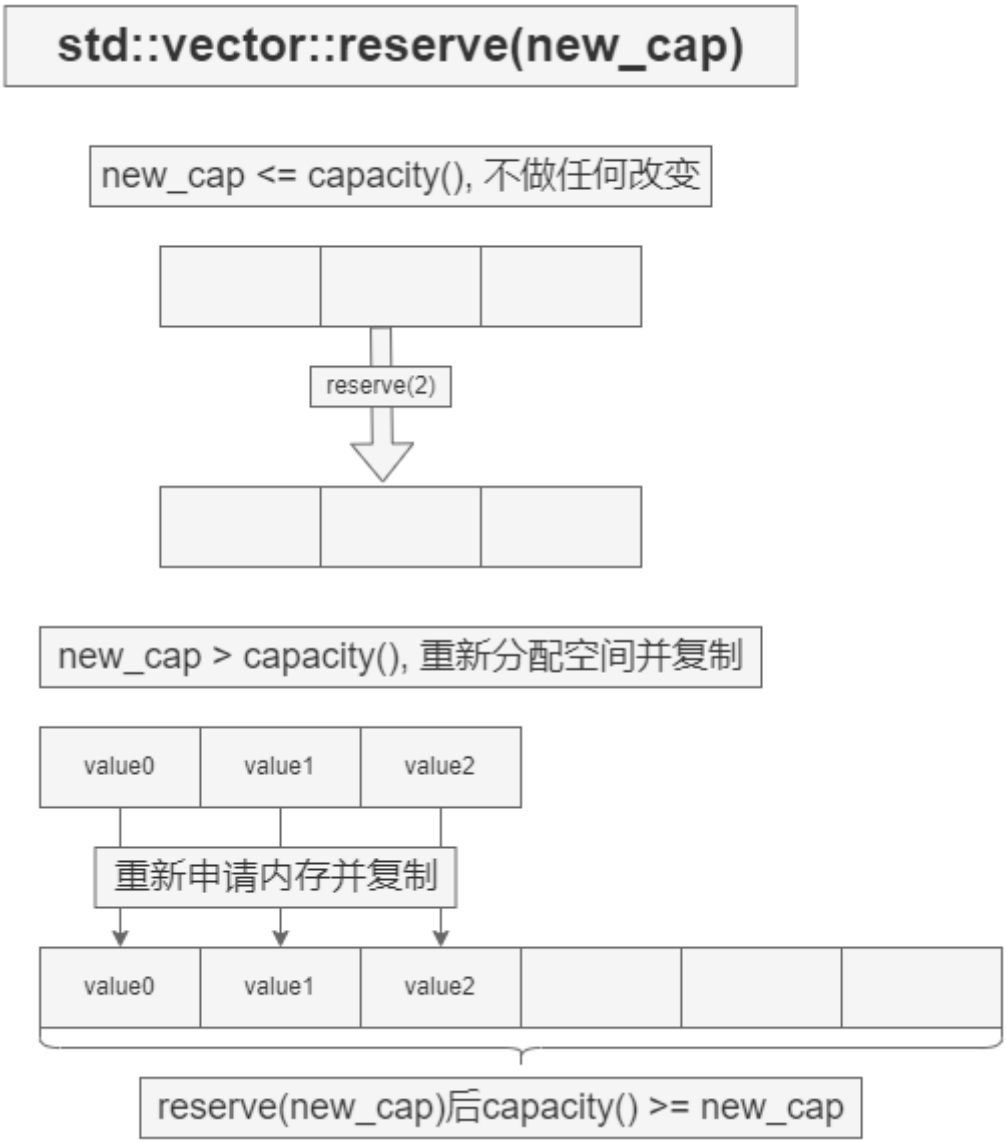
resize(count)

> 重设容器大小以容纳 count 个元素。
>
> 若当前大小大于 count， 则减小容器为其首 count 个元素。
>
> 若当前大小小于 count，
>
> 1. 则后附额外的默认插入的元素
> 2. 则后附额外的 value 的副本



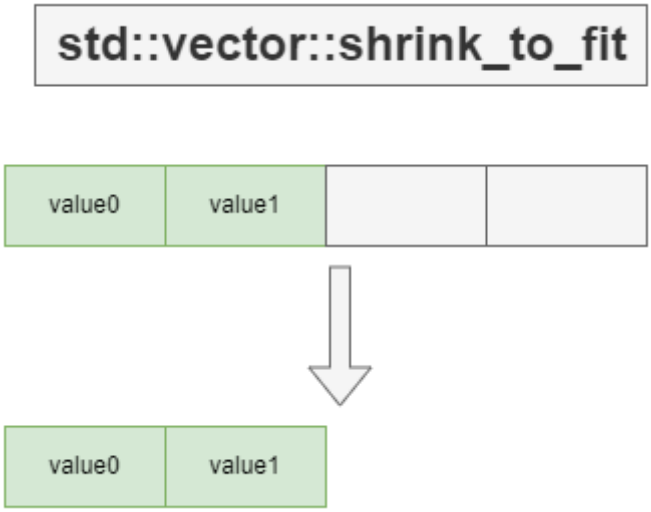
reserve(new_cap)

> 增加 vector 的容量到大于或等于 new_cap 的值。若 new_cap 大于当前的 capacity()，则分配新存储，否则该方法不做任何事。
>
> reserve() 不更改 vector 的 size 。
>
> 若 new_cap 大于 capacity()，则所有迭代器，包含尾后迭代器和所有到元素的引用都被非法化。否则，没有迭代器或引用被非法化。



shrink_to_fit() (C++11)

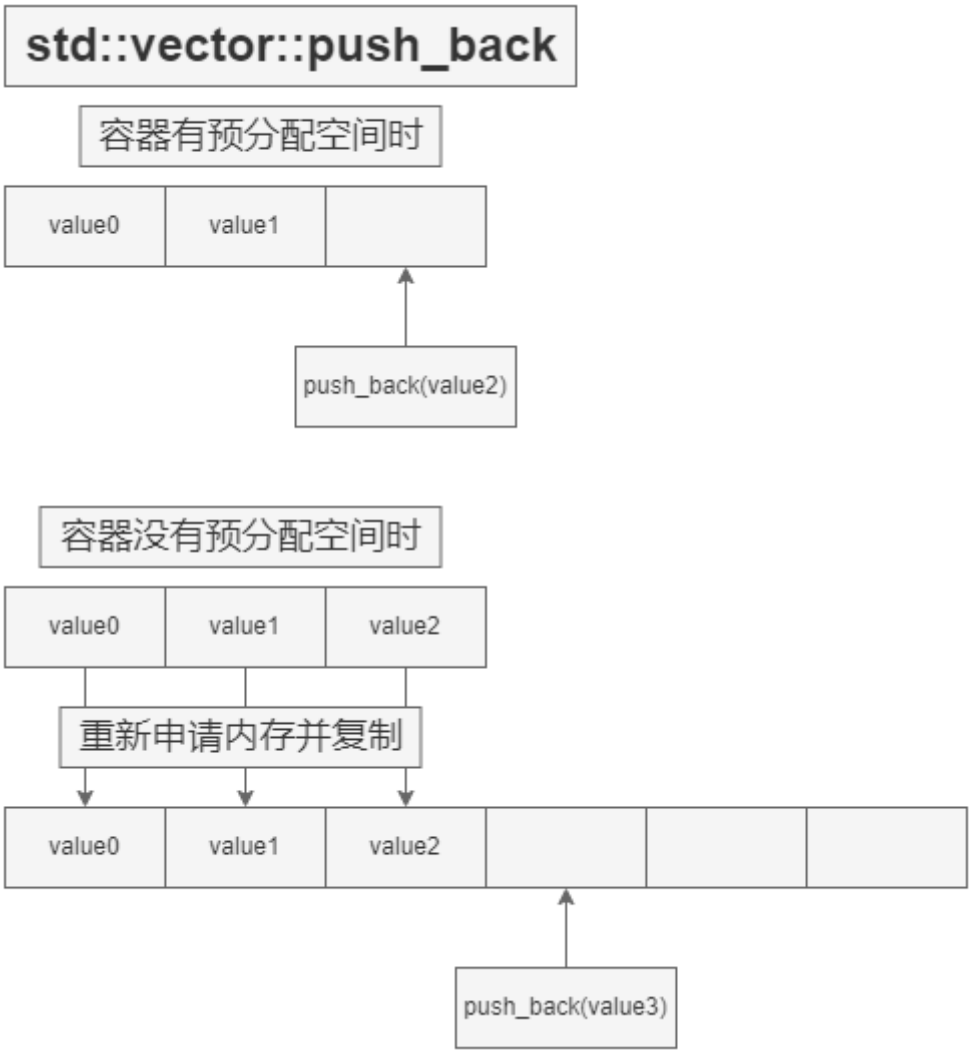
> 请求移除未使用的容量。
>
> 它是减少 capacity() 到 size()非强制性请求。请求是否达成依赖于实现。
>
> 若发生重分配，则所有迭代器，包含尾后迭代器，和所有到元素的引用都被非法化。若不发生重分配，则没有迭代器或引用被非法化。

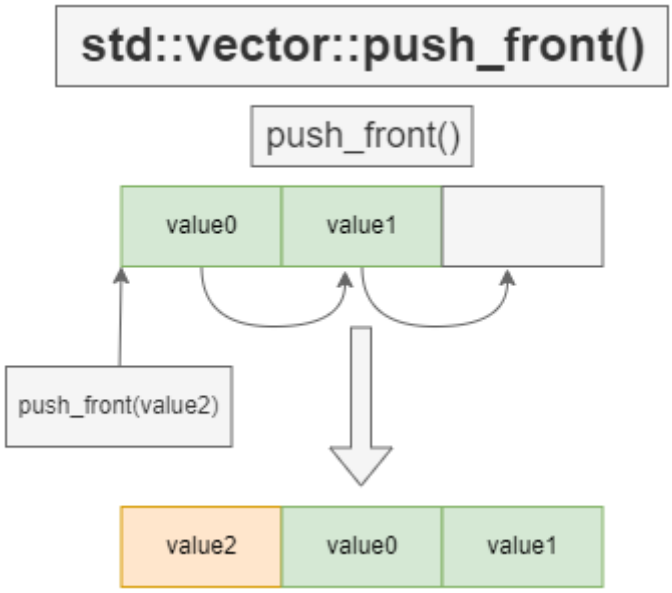


C++11 之前可以使用如下方式，缩减空间

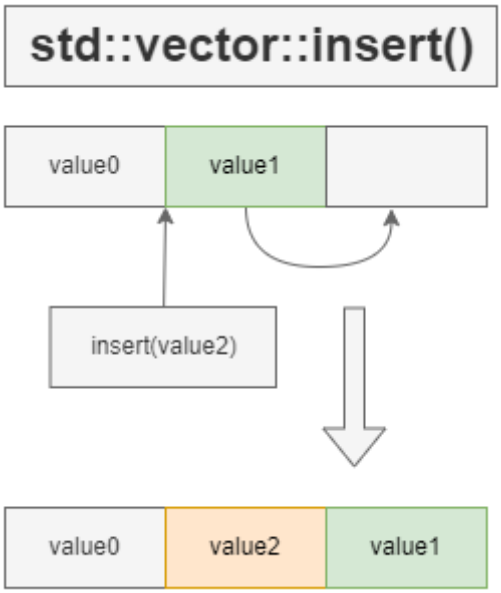
```
template<typename T>
void ShrinkCapacity(const std::vector<T>& v)
{
    std::vector<T> tmp(v);
    v.swap(tmp);
}
```

push_back() 与 push_front()

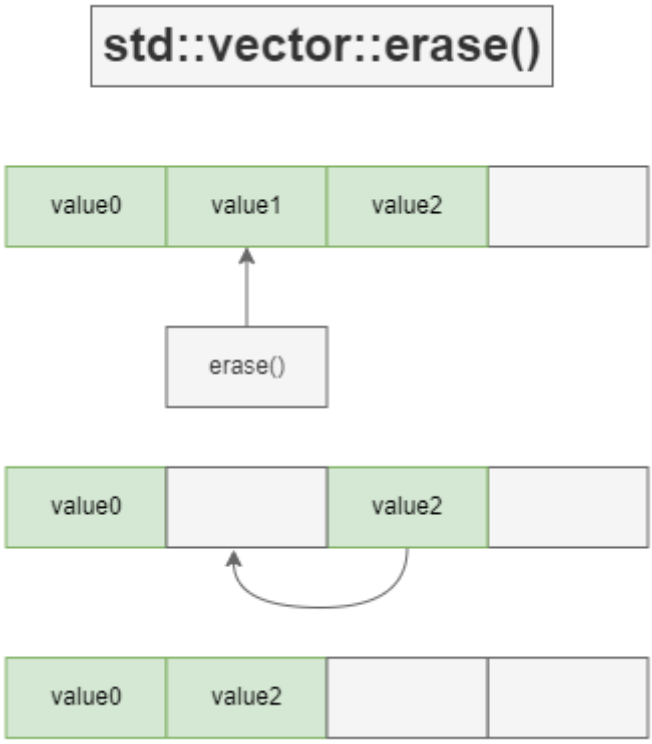




insert()



erase()




```
std::vector<int> vec{1,2,3};
std::vector<int>::iterator it = vec.begin();
while(it != vec.end())
{
    if(condition)
    {
        it = vec.erase(it);
        /// 下面这句话为错误
        /// vec.erase(it++);
    }
}
```

data()

> 返回指向作为元素存储工作的底层数组的指针。指针满足范围 [data(); data() + size()) 始终是合法范围，即使容器为空（该情况下 data() 不可解引用）。

拷贝 `vector` 中的数据到缓冲区

```
unsigned char auc[] = {1,2,3,4,5,6};

/// unsigned char 数组转换为 vector
std::vector<unsigned char> vec(auc, auc+sizeof(auc));

/// vector 转换为 unsigned char 数组
unsigned char* puc = new unsigned char[vec.size()];
memcpy(puc, vec.data(), vec.size());
```

std::vector < bool >

该容器不是 布尔类型的 `std::vector`, 而是有单独的实现，不应看作普通的 `std::vector` 使用。

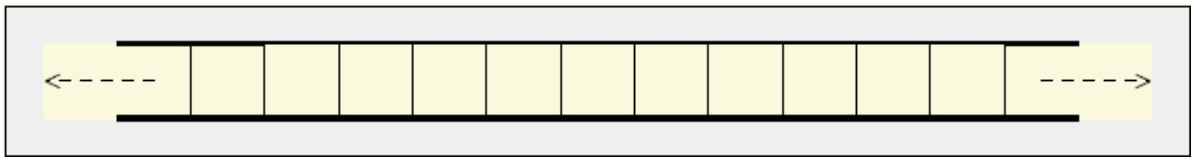
异常处理

1. 如果 `push_back()` 安插元素时发生异常，函数将不会产生效用。
2. 如果元素的 `copy/move` 操作(包括构造函数和赋值运算符)不抛出异常，这意味着 `insert()`、`emplace()`、`emplace_back()`、`push_back()` 要么成功，要么什么也不发生。
3. `pop_back()` 不会抛出任何异常
4. 如果元素的 `copy/move` 的操作(包括构造函数和赋值运算符)不抛出异常, `erase()` 就不抛出异常
5. `swap()` 和 `clear()` 不抛出异常
6. 如果元素的 `copy/move` 操作(包括构造函数和赋值运算符)不抛出异常, 那么所有操作不是成功就是不产生效用。这类元素可能是 POD。

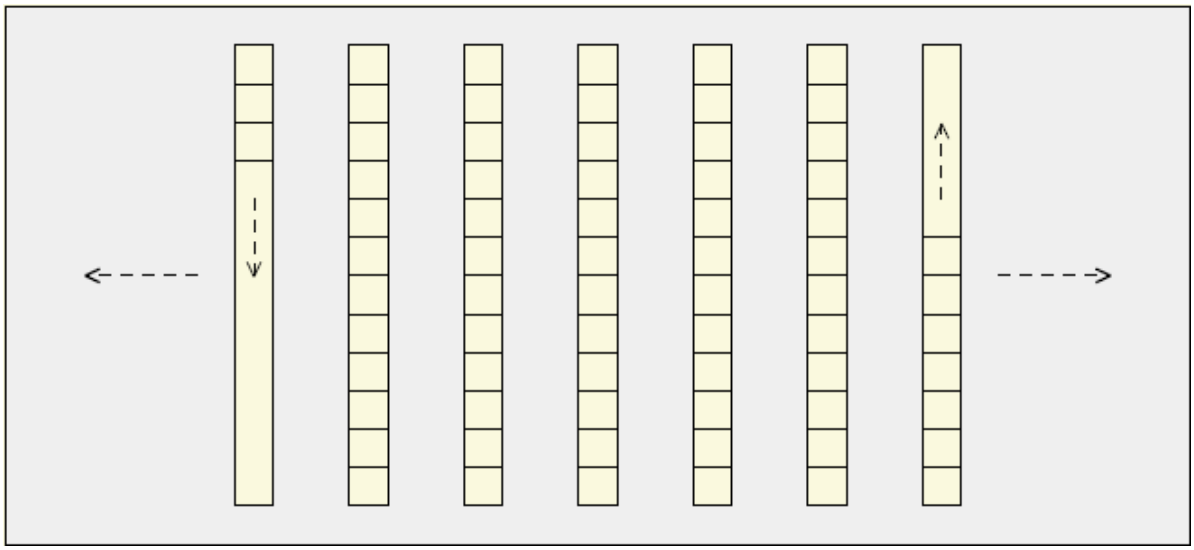
以上所有保证都基于一个条件: 析构函数不得抛出异常。

`std::deque`

提供随机访问，接口与 `std::vector` 几乎一致。可以在首尾快速安插和删除。



通常实现为一组独立的区块，第一区块往一个方向发展，最末的区块往另一个方向发展。



`std::deque` 与 `std::vector` 比较

相同之处

- 在中段安插、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。
- 迭代器属于随机访问迭代器，可以使用 `operator[]` \ `at()`

不同之处

- `std::deque` 可在常量时间内快速在首尾增删元素。 `std::vector` 只能在尾部。
- 访问元素时， `std::deque` 多了一个跳转的过程(在各个区块跳转)
- `std::deque` 的迭代器不是原始指针(因为各个区块之间不连续), 更没有 `data` 的成员函数
- `std::deque` 不支持对容量和内存分配时机的控制。
- `std::deque` 在首尾两端增删元素导致所有元素的迭代器失效(指针和引用仍有效)，其他所有增删操作都会导致所有元素的指针、引用和迭代器失效。
- `std::deque` 的内存分配优于 `std::vector`, `std::deque` 不必在内存分配时复制所有元素。
- `std::deque` 会释放不再使用的内存区块。 `std::deque` 的内存大小是可缩减的, 但要不要这么做，以及如何做，由实现决定。
- `std::deque` 不提供容量操作 `capacity()` 和 `reserve()`
- 在内存区块大小有限制的系统中, `std::deque` 的 `max_size()` 可能比 `std::vector` 的 `max_size()` 要大。 因为 `std::deque` 使用的不止一块内存。

适用场景

- 需要在两端安插和移除元素
- 无须指向容器内的元素
- 要求使用内存会自动缩小

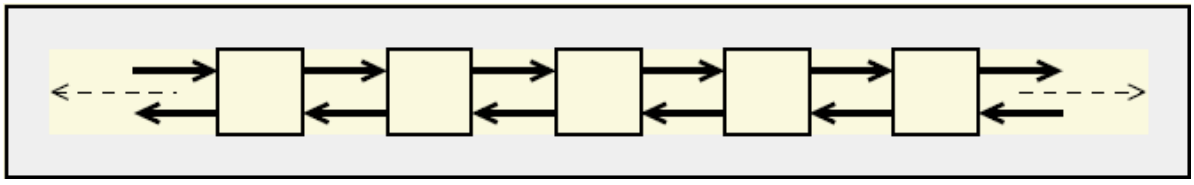
异常处理

原则上 `std::deque` 提供的异常处理和 `std::vector` 提供的一样

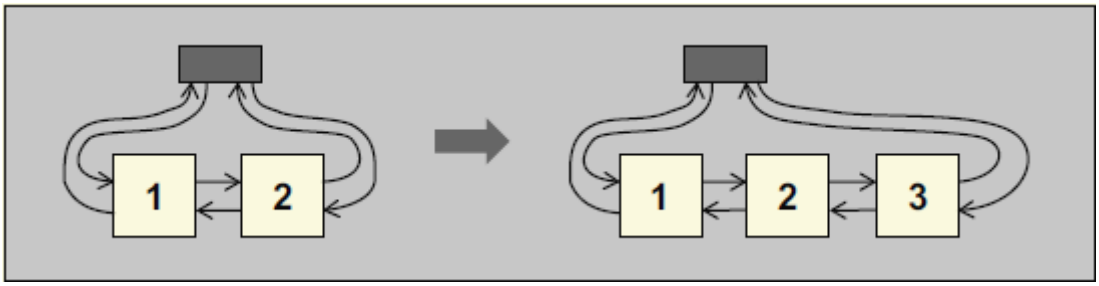
- `push_front()` 和 `push_back()` 安插元素时发生异常，则该操作不带来任何效应。
- `pop_front()` 和 `pop_back()` 不会抛出任何异常。

`std::list`

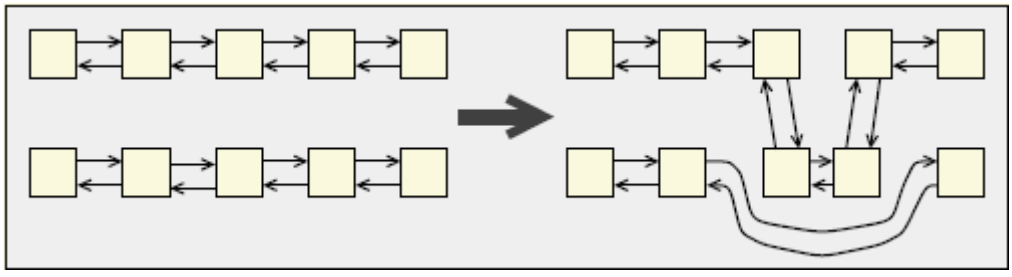
双向链表。



`std::list` 的内部结构完全迥异于 `std::array`、`std::vector`、`std::deque`。 `std::list` 自身提供了两个指针，分别指向第一个元素和最后一个元素，如果操纵对应的指针即可。



成员函数 `splice` 示意



容器特性

- 提供 `front()`、`push_front()`、`pop_front()`、`back()`、`push_back()` 和 `pop_back()` 等操作函数。
- 不提供 `operator[]` 或 `at()`
- 不支持随机访问。O(n)
- 在任何位置插入元素非常快。O(1), 只是改变了指针指向。
- 迭代器永久有效。插入和删除动作并不会造成指向其他元素的指针、引用和迭代器失效。
- 异常安全 `std::list` 的异常处理为: 要么操作成功、要么什么都不发生。
- 事务安全。 只要不调用赋值操作或 `sort()`, 并保证元素相互比较时不抛出异常那么 `std::list` 可以成为事务安全

- 空间最优。没有空间重新分配和预分配内存, 没有冗余内存占用
- 拥有较多的特殊成员函数，相较于 `STL` 中通用的同名函数，更具有效率。如 `merge`、`splice`、`remove`、`reverse`、`unique`、`sort`。

应用场景

特性: 前向迭代器

- 排序

```
std::list<int> list;
list.sort();
/// 错误用法: std::sort(list.begin(), list.end());
```

- 特殊的排序后显示

```
std::list<int> list;
list.push_back(5);
list.push_back(7);
list.push_back(2);
list.push_back(1);
list.push_back(8);
list.push_back(5);

std::vector<std::reference_wrapper<int> > Observer(l.begin(), l.end());

std::sort(Observer.begin(), Observer.end());
```

特性: 迭代器永不失效

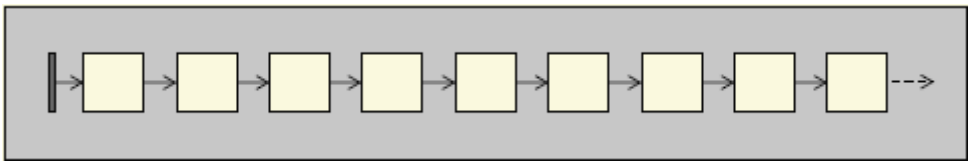
- 双键结构

```
template <class LEFT_KEY, class RIGHT_KEY, class VALUE>
class bimap
{
public:

...

private:
    std::list<VALUE> m_Value;                       ///< 用于存放值
    std::map<LEFT_KEY, iterator> m_LeftKeyMap;     ///< 用于保存左键与值得映射关系的
map
    std::map<RIGHT_KEY, iterator> m_RightKeyMap;  ///< 用于保存右键与值得映射关系的
map
};
```

`std::forward_list` (C++11)

标准描述

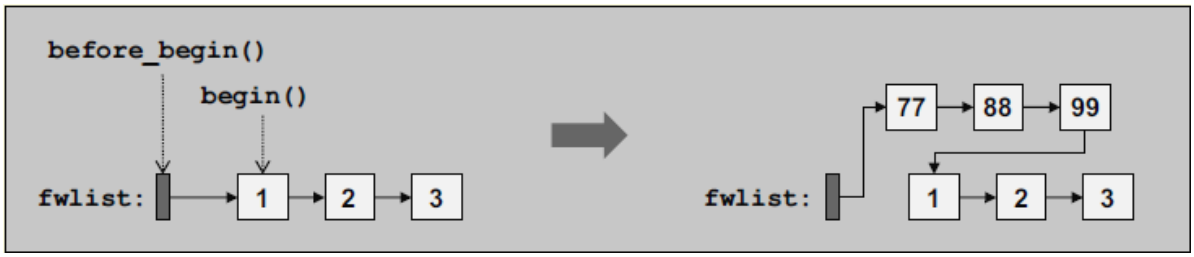
> `std::forward_list` 是支持从容器中的任何位置快速插入和移除元素的容器。不支持快速随机访问。它实现为单链表，且实质上与其在 `C` 中实现相比无任何开销。与 `std::list` 相比，此容器在不需要双向迭代时提供更有效地利用空间的存储。



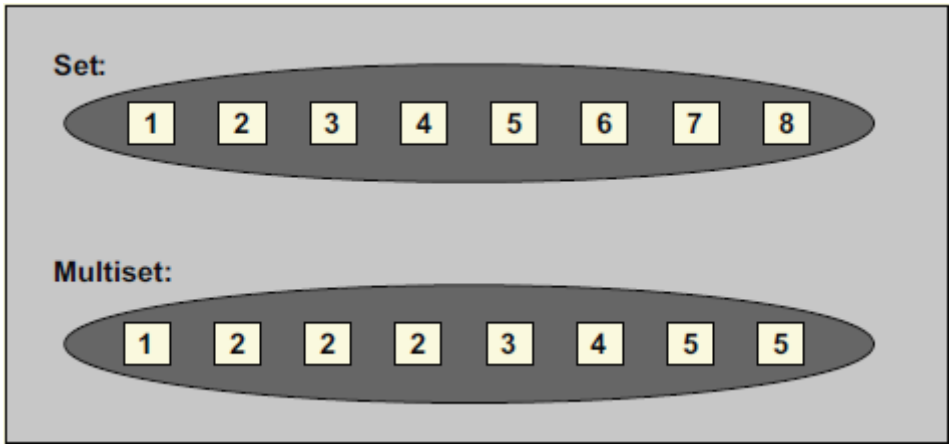
与 `std::list` 比较

- `std::forward_list` 只提供前向迭代器，而不是双向迭代器。没有成员函数 `rbegin()`、`rend()`、`crbegin()` 和 `crend()`
- `std::forward_list` 不提供成员函数 `size()`。
- `std::forward_list` 没有指向最末元素的指针。所以没有成员函数如 `back()`、`push_back()` 和 `pop_back()`
- 对于所有令元素被安插在或删除于的某特定位置上的成员函数，`std::forward_list` 都提供特殊版本。原因是你必须传递第一个被处理元素的前一位置，前向迭代器不能回头。
- `insert_after()` 代替 `insert()`，也额外提供 `before_begin()` 和 `cbefore_begin()`

在起始处安插元素

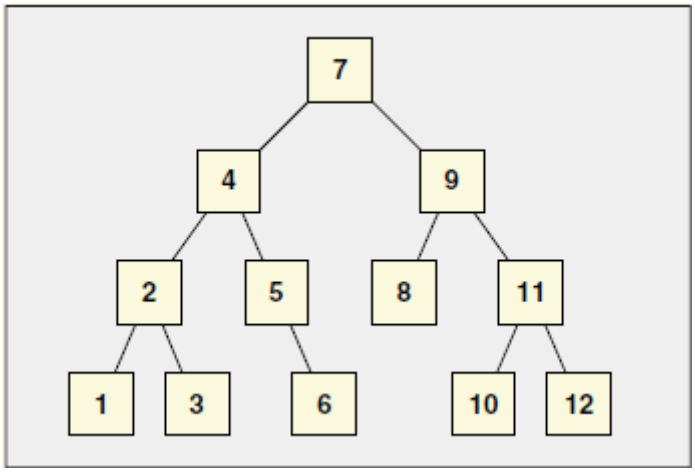


`std::set` 和 `std::multiset`



`std::set` 和 `std::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 `std::multiset` 允许元素重复而 `std::set` 不允许。

如果没有传入某个排序准则，就采用默认准则 `std::less` 以 `operator<` 对元素进行比较。



排序准则符合: 严格弱序

详细定义

> 1. 必须是**非对称的**（antisymmetric）。
>
>   对`operator<`而言， 如果x < y为true， 则y < x为false。
>
>   对判断式(predicate) `op()`而言，如果op(x, y)为true，则op(y, x)为false。
> 2. 必须是**可传递的**（transitive）。
>
> 对`operator<`而言，如果x < y 为true且y < z为true， 则x < z 为false。
>
> 对判断式(predicate) `op()`而言，如果op(x, y)为true且op(y, z)为tru，则op(x, z)为true。
>
> 3. 必须是**非自反的**（irreflexive）
>
>   对`operator<`而言，x < x 永远是false
>
>   对判断式(predicate) `op()`而言，op(x, x)永远是false。
> 4. 必须有**等效传递性**（transitivity of equivalence）

> 对`operator<`而言，假如 !(a<b) && !(b<a) 为true且 !(b<c) && !(c<b) 为 true
> 那么!(a<c) && !(c<a) 也为true.
> 对判断式(predicate) `op()`而言， 假如 op(a,b), op(b,a), op(b,c), 和op(c,b) 都为false, 那么op(a,c) and op(c,a) 也为false.

简单的来说就是a<b返回true，a=b和a>b返回false。

定制排序规则 `operator<`

```
class CALL_INFO_C
{
public:
    int x;
    std::string y;

    bool operator<(const CALL_INFO_C& stOther) const
    {
        return x < stOther.x || (x == stOther.x && y < stOther.y);
    }
};

std::set<CALL_INFO_C> set;
```

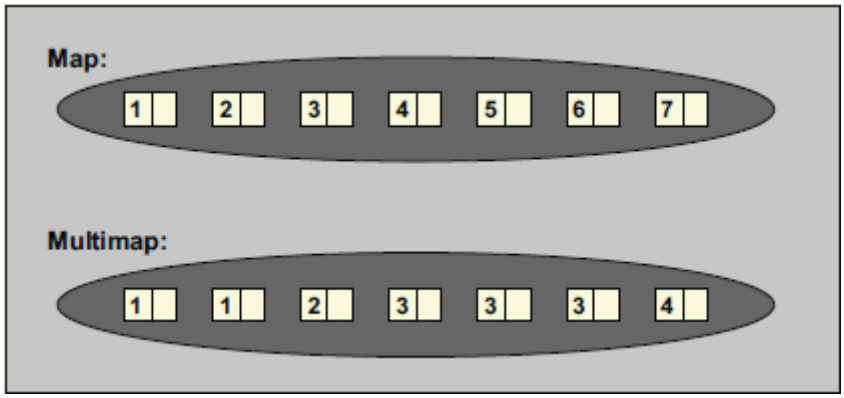
其中 `std::multiset` 的等效元素的次序是随机但稳定的。(C++11以后标准保证新插入的元素，会被放在等效元素群的末尾)

`std::set` 和 `std::multiset` 的能力

- 通常以平衡二叉树完成。
- 自动排序的主要优点在于令二叉树于查找元素时拥有良好的性能。其查找函数具有 `O(logn)` 的时间复杂度。
- 不能随意改变元素值，因为这会打乱原本正确的顺序。
- 如果要改变元素值，必须先删除旧元素，再插入新元素。
- 不提供任何操作函数可以直接访问底层元素
- 通过迭代器进行元素间接访问，有一个限制: 从迭代器的角度看，元素值是常量.(例如不能使用: `std::remove()`)
- 其迭代器是双向迭代器(不能使用 `std::sort()`)

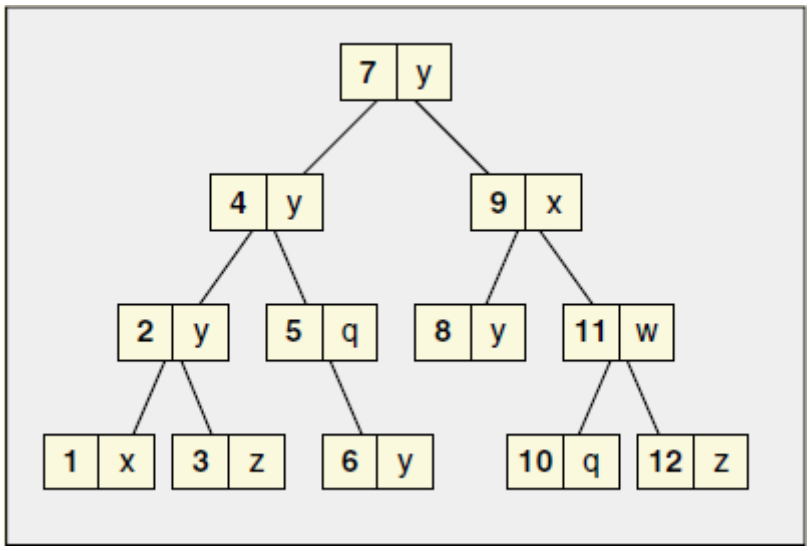
`std::map` 和 `std::multimap`

`std::map` 和 `std::multimap` 将 `key/value pair` 当作元素进行管理。它们可根据 `key` 的排序准则自动为元素排序。 `std::multimap` 允许重复元素，`std::map` 不允许。



同样 `key` 需要可比较且遵循严格弱序。

`std::map` 和 `std::multimap` 通常以平衡二叉树完成。



`std::map` 和 `std::multimap` 也无法改变 `key` 的值。只能删除再插入。

operator[]

若 `key` 不存在，构造该元素后，返回元素的引用
若 `key` 存在，返回元素的引用

所以要警惕如下语句:

```
std::map<std::string, int> map;
std::cout << map["string"]; ///< 这里会默认插入一个元素 ("string", 0)
```

```
std::map<std::string, int> map;
map["string"] = 1;
map["string"] = 2;  ///< 会覆盖前面的值
```

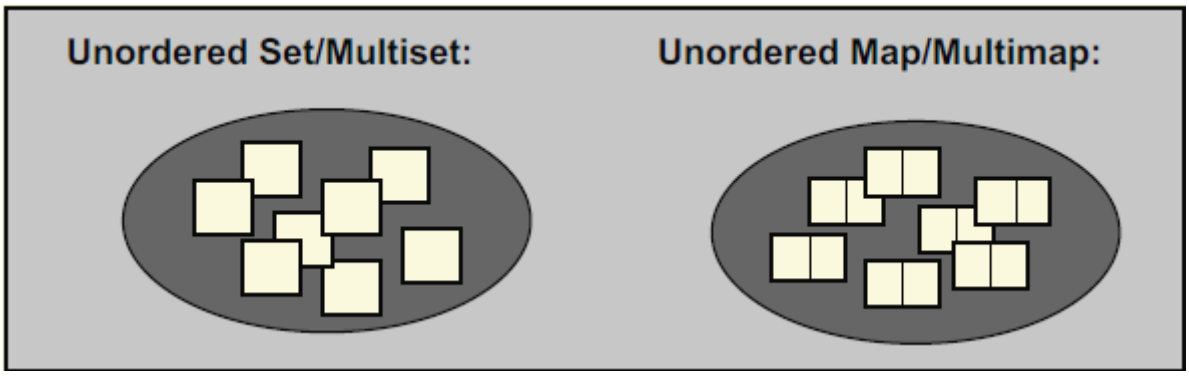
也可以利用这一特性用来计数:

```
std::map<char, int> map;
std::string str("Hello World!");
for(unsigned i = 0; i < str.size(); ++i)
{
    map[str.at(i)]++;
}
```

无序容器 (Unordered Container) C++11

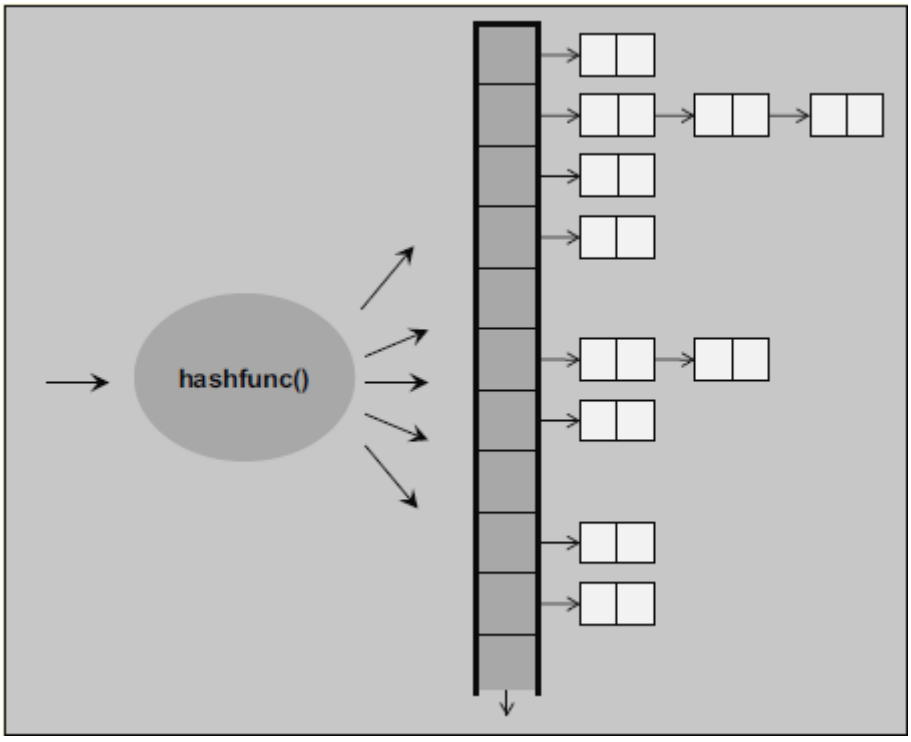
C++11 之前因为标准库中没有哈希表类的数据结构，所以很多程序库自己实现了诸如 `hash_set`、`hash_multiset`、`hash_map`、`hash_multimap`。

为了避免名称冲突，C++11 标准采用了不一样的名称，使用统一前缀 `unordered_`，即 `unordered_set`、 `unordered_multiset`、`unordered_map`、`unordered_multimap`。



`unordered_set`、 `unordered_multiset`、`unordered_map`、`unordered_multimap` 底层实现都是哈希表，所以 `key` 需要可哈希。

但是在链表是单链还是双链(意味着其迭代器可能不是双向迭代器)，重新哈希的时机这些都没有指定，根据实现而定。



定制哈希示例

```
class MY_HASH
{
public:
    std::size_t operator()(const CALL_INFO_C& st)
    {
        return ...;
    }
};

std::unordered_map<CALL_INFO_C, int, MY_HASH> map;
```

容器特性

安插、删除、查找元素大部分是 O(1), 但偶尔发生的重新哈希时间复杂度变为 O(n)

由于其迭代器只保证至少为前向迭代器，因此不提供包括 `rbegin()`、 `rend()` 以及不能使用要求双向迭代器的的 STL 函数如 `std::sort()`、`std::binary_search()`

你可以手动强制重新哈希。

重新哈希可能发生在以下调用之后: `insert()`、`rehash()`、`rehash()` 或 `clear()`。

`erase()` 函数并不会令指向其他元素的指针、引用和迭代器失效。

`insert()` 和 `emplace()` 可能令所有迭代器失效。但不会影响引用的有效性。

当重新哈希过程发生，元素的引用仍然有效。

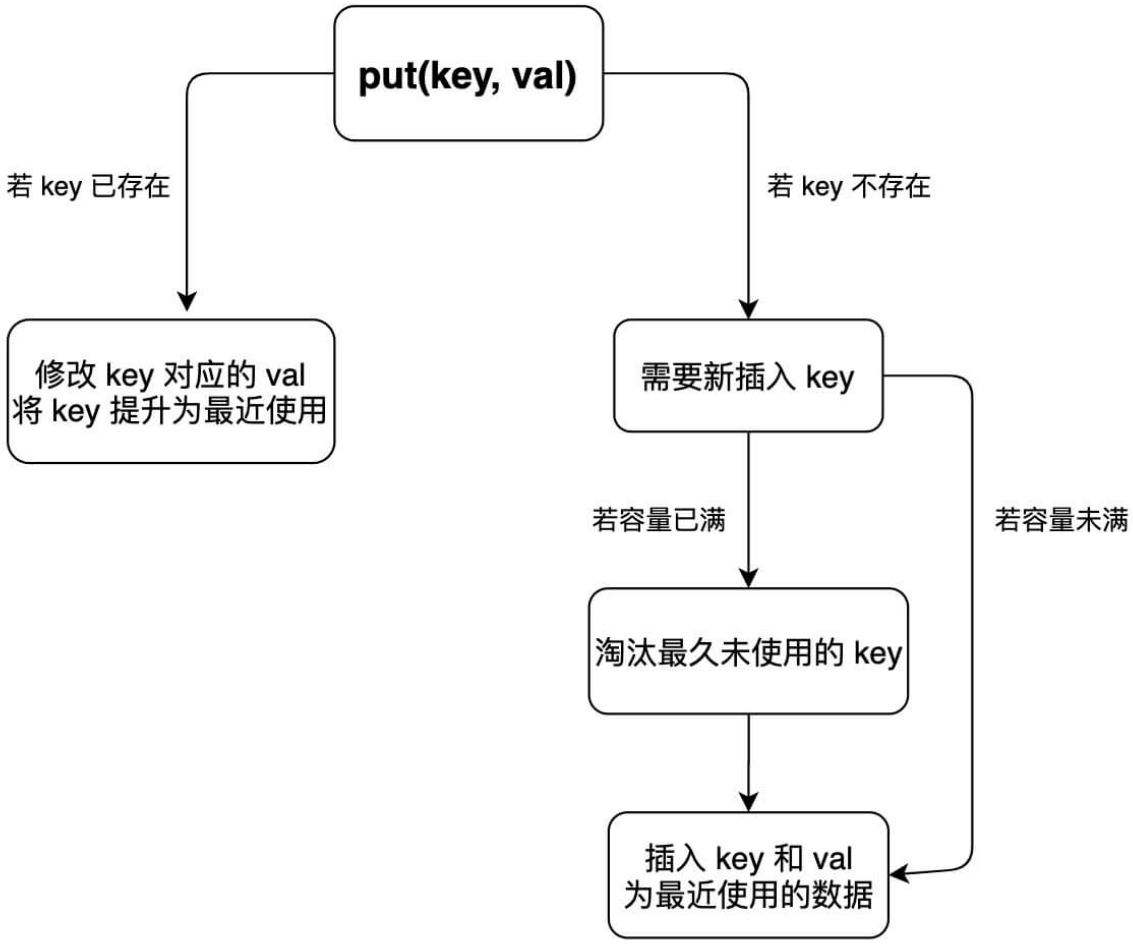
应用场景

假如缓存中，我们缓存若干最近访问和删除的记录至内存用于快速访问，使得插入记录和读取最近的记录的时间复杂度为 `O(1)`。

- LRU (Least recently used)

设计接口

- `LRUCache(int capacity)` 以 正整数 作为容量 `capacity` 初始化 LRU 缓存
- `int get(int key)` 如果关键字 `key` 存在于缓存中，则返回关键字的值，否则返回 `-1` 。
- `void put(int key, int value)` 如果关键字 `key` 已经存在，则变更其数据值 `value` ；如果不存在，则向缓存中插入该组 `key-value` 。如果插入操作导致关键字数量超过 `capacity` ，则应该擦除最久未使用的关键字。
- 函数 `get` 和 `put` 必须以 `O(1)` 的平均时间复杂度运行。



设计思路:

```
typedef std::pair<int, int> KEY_VALUE;

std::unordered_map<int, std::list<KEY_VALUE>::iterator> map;
std::list<KEY_VALUE> list;
```

`std::unordered_map` 特性: 单向迭代器, 增删元素 O(1), 增删元素后迭代器可能失效

`std::list` 特性: 迭代器永不失效，任意位置插入常量时间 O(1), 访问首尾元素 O(1)

完整代码

```
class LRUCache
{
public:
    LRUCache(int capacity) : max_size(capacity) {}

    int get(int key)
    {
        auto res = map.find(key);
        if (res != map.end())
        {
            list.splice(list.begin(), list, res->second);
            return res->second->second;
```

```
        }
        else
        {
            return -1;
        }
    }

    void put(int key, int value)
    {
        auto res = map.find(key);
        list.push_front(std::make_pair(key, value));
        if (res != map.end())
        {
            list.erase(res->second);
            map.erase(res);
        }
        map[key] = list.begin();

        /// 检查是否超出了最大数量
        if (map.size() > max_size)
        {
            auto last = list.end();
            --last;
            map.erase(last->first);
            list.pop_back();
        }
    }

private:
    typedef std::pair<int, int> KEY_VALUE;

    std::unordered_map<int, std::list<KEY_VALUE>::iterator> map;
    std::list<KEY_VALUE> list;
    int max_size = 0;
};
```

特殊容器

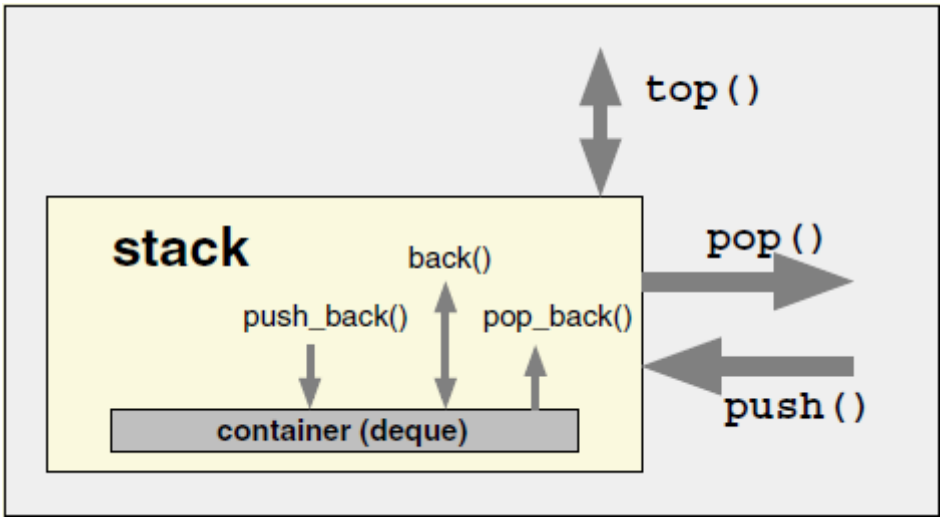
`std::string`

其被定义为: `std::basic_string<char>`

其中模板入参 `char` 可以换为 `unsigned char` 或 `wchar`

`std::stack`

后进先出

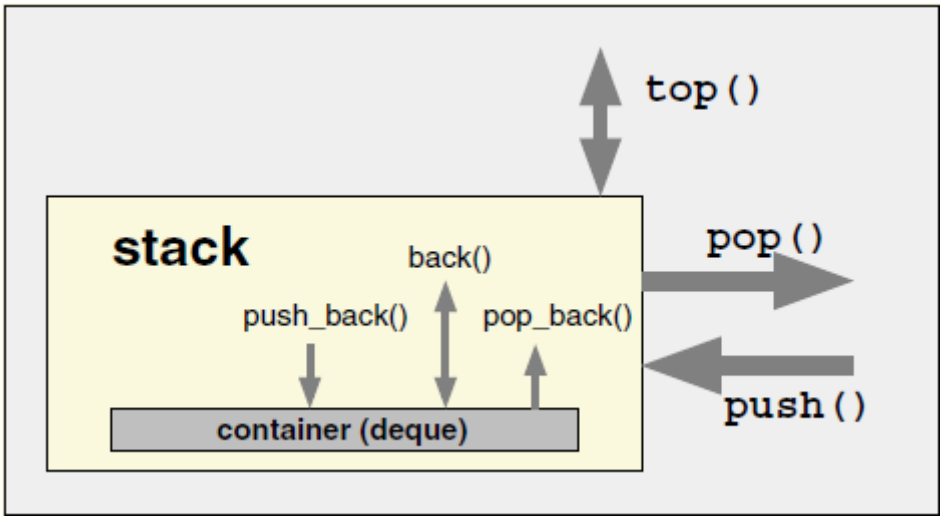


`std::stack` 定义如下:

```
template <typename T,
typename Container = deque<T>>
class stack;
```

其底层类型默认为 `std::deque`

之所以不选择 `std::vector` 是因为在内存管理上 `std::deque` 比 `std::vector` 更有效率。

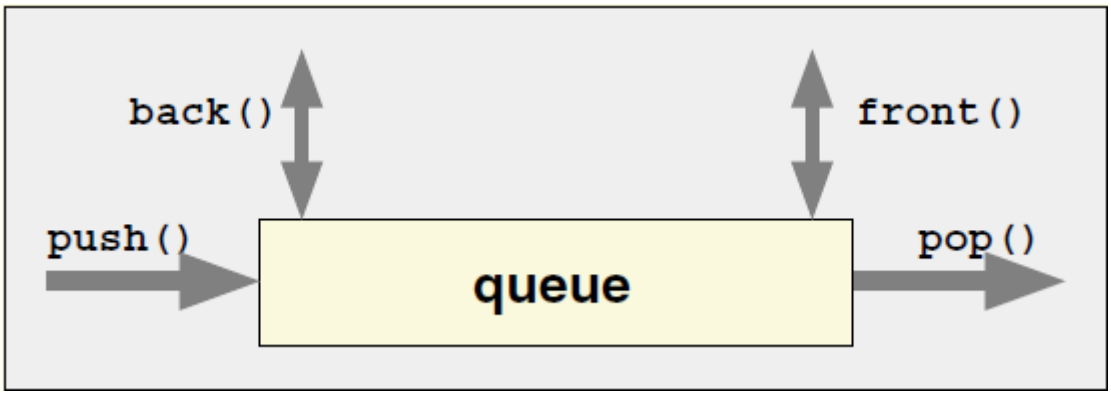



```
template <typename T,
typename Container = deque<T>>
class stack
```

```
{
public:
    T& top()
    {
        return m_deque.back();
    }

    void push(const T& value)
    {
        m_deque.push_front(value);
    }

    void pop()
    {
        c.pop_back();
    }

private:
    Container m_deque;
};
```

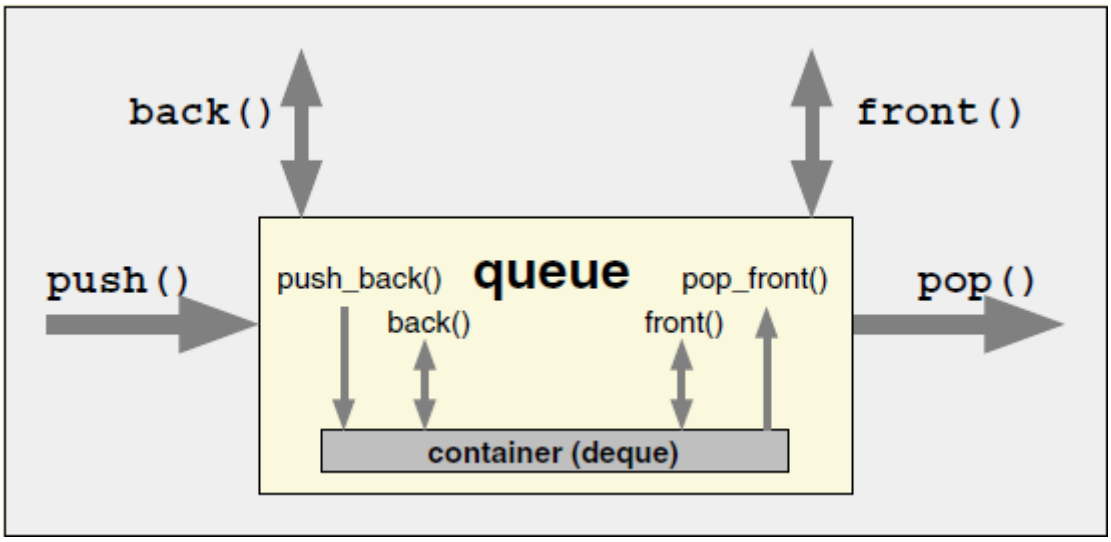
`std::queue`

先进先出



底层实现默认采用 `std::queue`

```
template <typename T,
typename Container = deque<T>>
class queue;
```

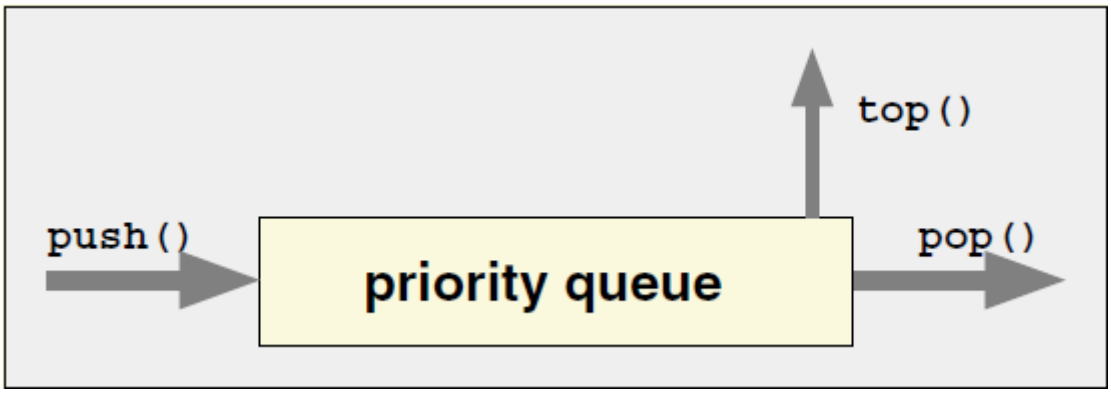



```
template <typename T,
typename Container = deque<T>>
class queue
{
public:
    T& top()
    {
        return m_deque.front();
    }

    void push(const T& value)
    {
        m_deque.push_back(value);
    }

    void pop()
    {
        c.pop_front();
```

```
    }

private:
    Container m_deque;
};
```

std::priority_queue

实现出一个队列，其中的元素按优先级存储。




```
template <typename T,
typename Container = vector<T>,
typename Compare = less<typename Container::value_type>>
class priority_queue;
```

应用实例:

求数据流中的中位数

```
class MedianFinder
{
public:
    void addNum(int num)
    {
        if (cnt % 2 == 0)
        {
            mi.push(num);
            num = mi.top();
            mi.pop();
            mx.push(num);
        }
        else
        {
            mx.push(num);
            num = mx.top();
            mx.pop();
            mi.push(num);
        }

        cnt++;
    }

    double findMedian()
    {
        if (cnt & 1)
        {
            return (double)mx.top();
        }
        else
```

```
        {
            return (mx.top() + mi.top()) / 2.0;
        }
    }

private:
    std::priority_queue<int, std::vector<int>, std::less<int> > mx;
    std::priority_queue<int, std::vector<int>, std::greater<int> > mi;
    int cnt = 0;
};
```

`std::bitset`

`std::bitset` 内含一个元素值为 `bit` 或 `bool` 值且大小固定的 `array`。当你需要管理各式 `flag`, 并以 `flag` 的任意组合来表现变量时, 就可运用 `std::bitset`。

可容纳任意个数的标志位(编译期确定数量)

`std::bitset` 编译期确定大小 `std::vector<bool>` 可动态增长。

桶式排序

给 1000 个数字排序, 数字范围 [0, 99]

```
unsigned a[1000] = {10, 99, 4, 4, 5, 6, 7};
std::bitset<100> bitset;
int length = sizeof(a) / sizeof(a[0]);

for (unsigned i = 0; i < length; ++i)
{
    unsigned pos = a[i];
    bitset.set(pos);
}

for (unsigned i = 0; i < bitset.size(); ++i)
{
    if (bitset[i])
    {
        std::cout << i << ", ";
    }
}
```

给 40 亿个 `unsigned` 数字([0, 99])中寻找不存在的数值

汉明距离

两个整数之间的 汉明距离 指的是这两个数字对应二进制位不同的位置的数目。多用于数据传输中的差错控制。

```
/// 0 <= x, y <= 2^31 - 1
int hammingDistance(int x, int y)
{
    std::bitset<32> temp(x^y);
    return temp.count();
}
```

迭代器介绍

`Iterator`(迭代器)是一种"能够迭代某序列内所有元素"的对象，可通过改变自寻常pointer的一致性接口来完成工作。`Iterator`奉行一个纯抽象概念：任何东西，只要行为类似iterator，就是一种iterator。然而不同的的iterator具有不同的行进能力。

迭代器种类

迭代器种类	能力	提供者
Output 迭代器	向前写入	Ostream,inserter
Input 迭代器	向前读取一次	Istream
Forward 迭代器	向前读取	Forward list、unordered containers
Bidirectional 迭代器	向前和向后读取	List、set、multiset、map、multimap
Random-access 迭代器	以随机访问方式读取	Array、vector、deque、string、C-style array

Output迭代器允许一步一步前行并搭配write动作。因此你可以一个一个元素地赋值，不能使用output迭代器对同一区间迭代两次。事实上，甚至不保证你可以将一个value复制两次而其迭代器不累进。我们的目标是将一个value以下列形式写入一个黑洞。

```
while(...) {
  *pos = ...;
  ++pos;
}
```

Output 迭代器无需比较操作。你无法检验output迭代器是否有效，或写入是否成功。你唯一可做的就是写入。通常，一批写入动作是以一个"额外条件定义出"的"特定output迭代器"作为结束。
见下表Output迭代器操作

表达式	效果
*iter = val	将val写至迭代器所指的位置
++iter	向前步进(step forward), 返回新位置
iter++	向前步进(step forward), 返回旧位置
TYPE(iter)	复制迭代器(copy 构造函数)

Input迭代器

Input迭代器只能一次一个以前行方向读取元素，按此顺序一个个返回元素值。

Input迭代器的各项操作

表达式	效果

表达式	效果
*iter	读取实际元素
iter->member	读取实际元素的成员(如果有的话)
++iter	向前步进(step forward), 返回新位置
iter++	向前步进(step forward), 返回旧位置
iter1 == iter2	判断两个迭代器是否相等
iter1 != iter2	判断两个迭代器是否不相等
TYPE(iter)	复制迭代器(copy 构造函数)

Input迭代器只能读取元素一次。如果你复制input迭代器, 并令原input迭代器和新产生的拷贝都向前读取, 可能会遍历到不同的值。
所有的迭代器都具备input迭代器的能力，而且往往更强。 `Pure input`迭代器的典型例子就是"从标准输入设备读取数据"。同一个值不会被读取两次。一旦从`input stream`读入一个字(离开input缓冲区), 下次读取时就会返回另一个字。

对于input迭代器, 操作符==和!=只用来检查"某个迭代器是否等于一个past-the-end迭代器(指指向最末元素的下一个位置)".这有其必要, 因为处理input迭代器的操作函数通常会有以下行为。

```
InputIterator pos, end;

while (pos != end) {
  ... // read-only access using *pos
  ++pos;
}
```

没有任何保证说，两个迭代器如果都不是past-the-end迭代器, 且指向不同位置，他们的比较结果会不相等(这个条件是和forward迭代器搭配引入的)。

也请注意, input迭代器的后置式递增操作符(`++iter`)不一定会返回什么东西。不过通常它会返回旧位置。
你应该尽可能优先先选用前置式递增操作符(`++iter`)而非后置式递增操作符(`iter++`), 因为前者效能更好。因为后者会返回一个临时对象。

Forward(前向)迭代器

Forward迭代器是一种input迭代器且在前进读取时提供额外保证。

表达式	效果
*iter	访问实际元素
iter->member	访问实际元素的成员
++iter	向前步进(返回新位置)

表达式	效果
iter++	向前步进(返回旧位置)
iter1 == iter2	判断两个迭代器是否相等
iter1 != iter2	判断两个迭代器是否不等
TYPE()	创建迭代器(default构造函数)
TYPE(iter)	复制迭代器(拷贝构造函数)
iter1 = iter2	对迭代器赋值(assign)
和input迭代器不同的是, 两个forward迭代器如果指向同一元素, `operator==` 会获得 `true`, 如果两者都递增, 会再次指向同一元素。	
例如:	

```
ForwardIterator pos1, pos2;

pos1 = pos2 = begin; /// both iterator refer to the same element
if(pos1 != end) {
  ++pos1; /// pos1 is one element ahead
  while(pos1 != end) {
    if(*pos1 == *pos2) {
      ... // precess adjacent duplicates
      ++pos1;
      ++pos2;
    }
  }
}
```

Forward迭代器由以下对象和类型提供:

- Class<forward_list>
- Unordered container
  然而标准库也允许 `unordered` 容器的实现提供 `bidirectional` 迭代器。
  如果forward迭代器履行了output迭代器应有的条件, 那么它就是一个mutable forward迭代器, 即可用于读取，也可用于涂写。

Random-Access(随机访问)迭代器

`Random-access` 迭代器在 `bidirectional` 迭代器的基础上增加了随机访问能里。因此它必须提供 `iterator` 算数运算。也就是说，它能增减某个偏移量、
计算距离(`difference`), 并运用诸如<和>等管理操作符(`relational operator`)进行比较。

随机访问迭代器的新增操作:

表达式	效果
iter[n]	访问索引位置为n的元素
iter+=n	前进n个元素(如果n是负数, 则改为回退)
iter-=n	回退n个元素(如果n是负数, 则改为前进)
iter+n	返回iter之后的第n个元素
n+iter	返回iter之后的第n个元素
iter-n	返回iter之前的第n个元素
iter1-iter2	返回iter1和iter2之间的距离
iter1 < iter2	判断iter1是否在iter2之前
iter1 > iter2	判断iter1是否在iter2之后
iter1 <= iter2	判断iter1是否不在iter2之后
iter1 >= iter2	判断iter1是否不在iter2之前

`Random-access`迭代器由以下对象和类型提供:

- 可随机访问的容器(`array`、`vector`、`deque`)
- String(`string`、`wstring`)
- 寻常的C-Style(`pointer`)

迭代器应用

判断字符串是否为回文。

```
constexpr bool is_palindrome(const std::string_view& s)
{
    return std::equal(s.begin(), s.begin() + s.size()/2, s.rbegin());
}

is_palindrome("1000000000个字符"); ///< 时间复杂度: O(1)
is_palindrome(str); ///< 时间复杂度: O(n)。 n = str.size();
```

```
bool is_palindrome(const std::string& s)
{
    return std::equal(s.begin(), s.begin() + s.size()/2, s.rbegin());
}
```

迭代器失效场景

迭代器非法化