STL

简介Standard Template Library，标准模板库，是C++的标准库之一，一套基于模板的容器类库，还包括许多常用的算法，提高了程序开发效率和复用性。包含六大部件：容器、迭代器、算法、仿函数、适配器和空间配置器。

- 容器：容纳一组元素的对象
- 迭代器：提供一种访问容器中每个元素的方法。
- 算法：包括查找算法、排序算法等等。
- 函数对象：一个行为类似函数的对象，调用它就像调用函数一样。
- 适配器：用来修饰容器等，比如queue和stack，底层借助了deque。
- 空间配置器：负责空间配置和管理

空间配置器

对象构造前的空间配置和对象析构后的空间释放，由<stl_alloc.h>负责，SGI对此的设计哲学如下：

- 向system heap要求空间。
- 考虑多线程状态。
- 考虑内存不足时的应变措施。
- 考虑过多“小型区块”可能造成的内存碎片问题。

双层级配置器

- 第一级直接使用**allocate()**调用**malloc()**、**deallocate()**调用**free()**,使用类似new_handler机制解决内存不足（抛出异常），配置无法满足问题。
- 第二级视情况使用不同的策略，当配置区块大于128bytes时，调用第一级配置器，当配置器小于128bytes时，采用内存池的整理方式：配置器维护16个（128/8）自由链表，负责16种小型区块的配置能力，内存池通过**malloc**获得内存，如果内存不足转第一级配置器处理。

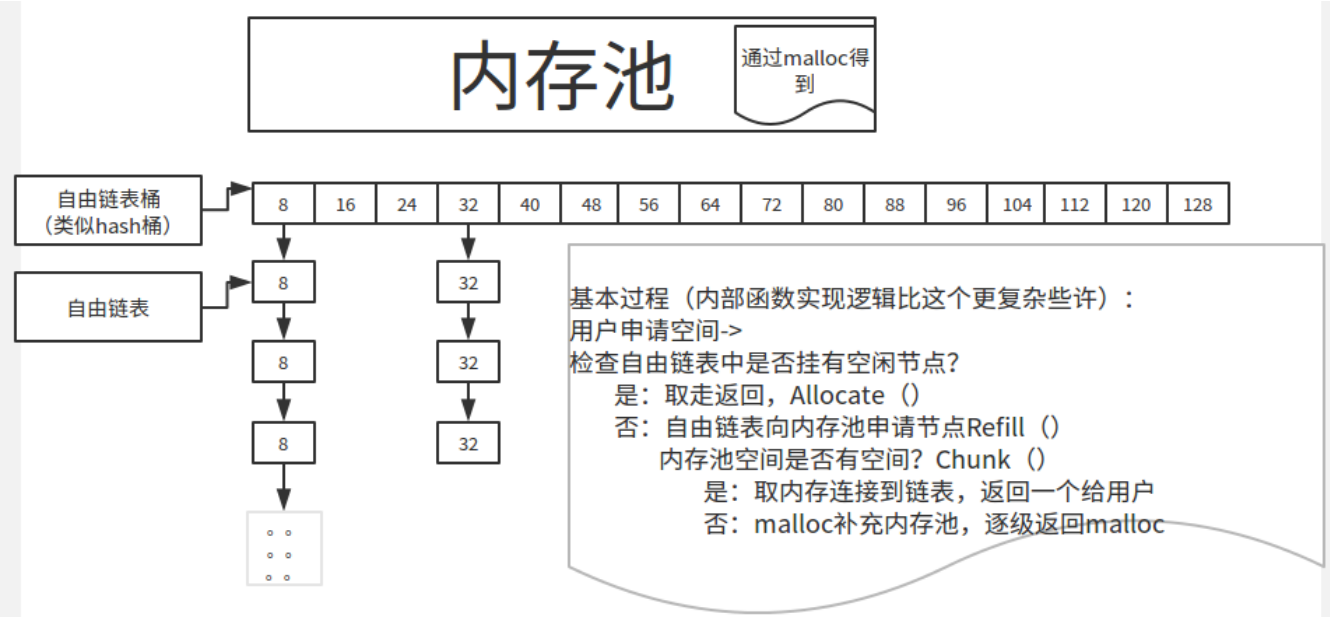
第一级配置器详解

```
template <int inst>
__malloc_alloc_template //一级配置器
```

```
// 当空间不足时，循环调用__malloc_alloc_oom_handler 试图释放出空间
// 然后再调用 malloc 申请空间，没有 handler 则抛出错误
-   static void *oom_malloc(size_t)
// 循环调用 handler 释放空间并试图 realloc 目标
-   static void *oom_realloc(void*, size_t)
// 自定义 handler，释放空间，初始设置为 0
-   static void (* __malloc_alloc_oom_handler)()
```

```
// 调用 malloc，失败则调用 oom 的 malloc
+ static void * allocate(size_t n);
// 调用 free
+ static void deallocate(void *p, size_t);
// 调用 realloc，失败则调用 oom 版本的
+ static void *reallocate(void *p, size_t, size_t new_sz);
// 设置释放空间的 handler
+ static void (* set_malloc_handler(void (*f)())) ();
```

第二级空间配置器详解



空间配置器存在的问题

- 自由链表所挂区块都是8的 整数倍，因此当需要非8倍数的区块，往往会导致浪费。
- 由于配置器的所有方法，成员都是静态的，那么他们存放在静态区，释放时机就是程序结束，这样会导致自由链表一直占有内存，自己进程可用，其他进程不能用。

各种容器的底层机制

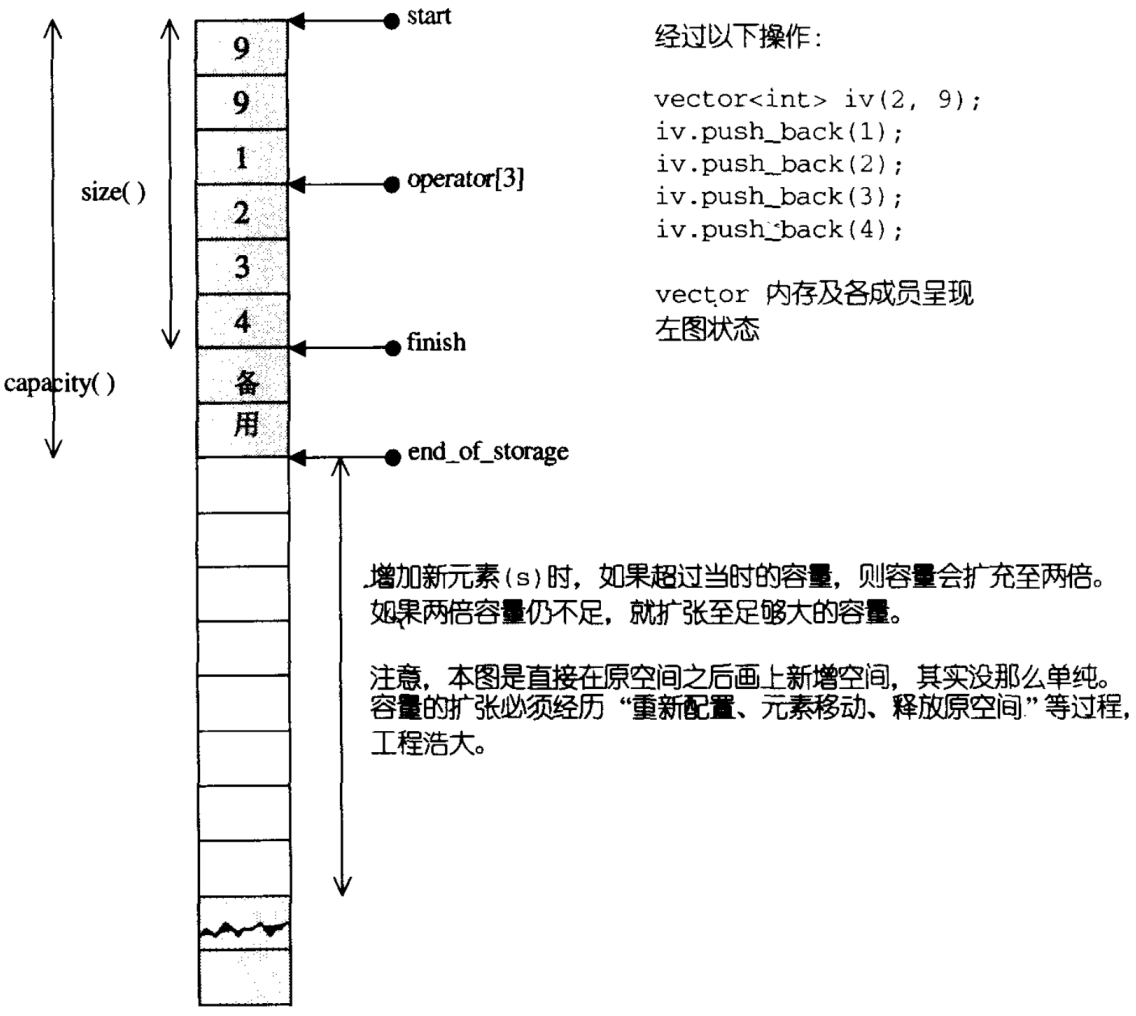
- **vector**： 可变大小的数组。支持随机快速访问，在尾部之外的位置插入或者删除元素可能很慢

- **deque**：双端队列，支持随机快速访问，在头尾位置插入/删除速度很快
- **list**：双向链表，只支持双向顺序访问。在list任何位置插入/删除速度很快
- **forward_list**:单向链表，只支持单向顺序访问
- **array**：固定大小的数组，支持快速随机访问，不能添加与删除元素
- **string**：与vector相似的容器，专门存储字符。

vector

- 底层原理：vector底层是一个动态数组，包含三个迭代器，start和finish之间是已经被使用的空间范围，end_of_storage是整块连续空间包括备用空间的尾部。

  当空间不够装下数据（vector.push_back(val)）时，会自动申请另一片更大的空间（2倍），然后把原来的数据拷贝到新的内存空间，接着释放原来的那片空间【vector内存增长机制】。当释放或者删除（vector.clear()）里面的数据时，其存储空间不释放，仅仅是清空了里面的数据。因此，对vector的任何操作一旦引起了空间的重新配置，指向原vector的所有迭代器会都失效了。



- vector中的reserve和resize的区别

  reserve()函数用来定义**预留空间**，改变capacity，不改变size。会去分配内存，但不会构造出对象。如果改变后的capacity比当前的capacity大，则capacity会变为改变后的capacity，反之capacity不变。

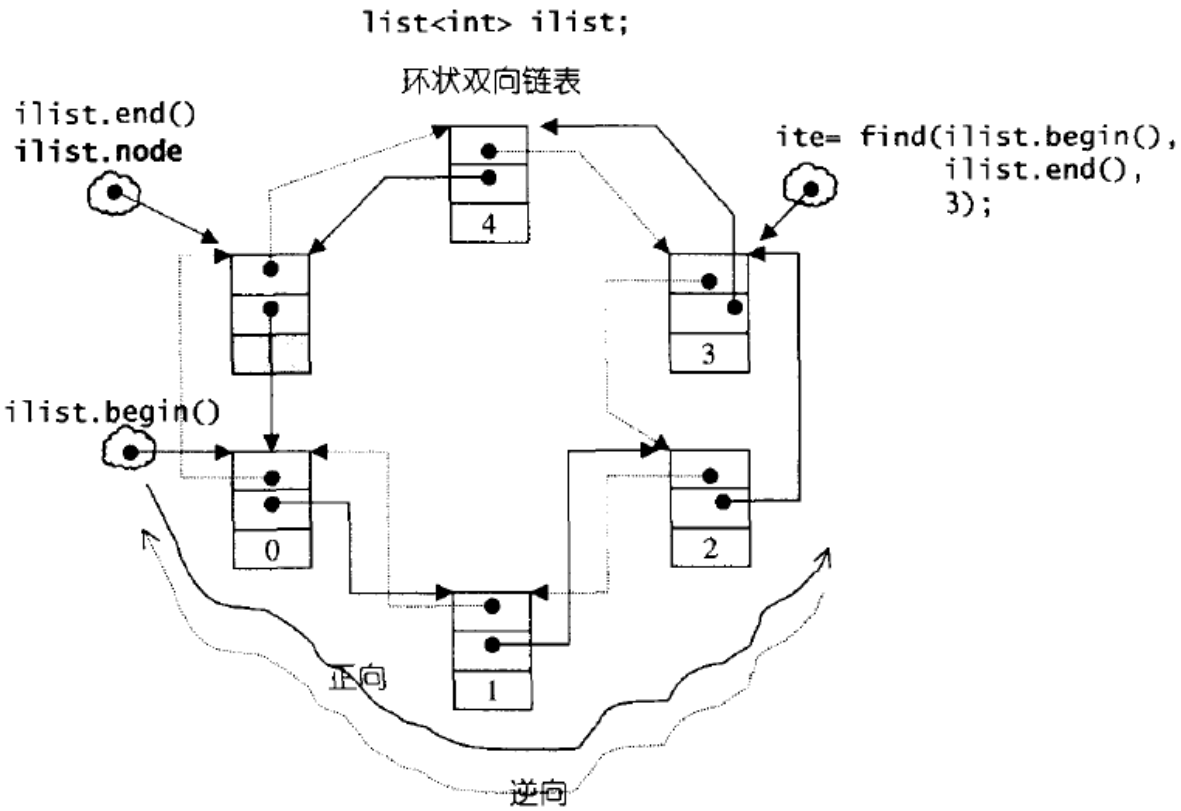
  resize用来改变vector的size，有可能也会改变capacity。如果改变后的size比当前capacity大，则capacity会变大，同时构造出多出来的对象；反之，capacity不变，同时析构一些不再需要的对象。

- vector的元素类型不可以是引用，vector的底层实现要求连续的对象排列，而引用并非对象没有实际地址，因此vector的元素类型不能是引用。
- vector 扩容为什么要以1.5倍或者2倍扩容?

  以2倍的方式扩容，导致下一次申请的内存必然大于之前分配内存的总和，导致之前分配的内存不能再被使用，所以最好倍增长因子设置为(1,2)之间。

list

- 底层原理：list的底层是一个双向链表，以结点为单位存放数据，结点的地址在内存中不一定连续，每次插入或删除一个元素，就配置或释放一个元素空间。list不支持随机存取，如果需要大量的插入和删除，而不关心随即存取。

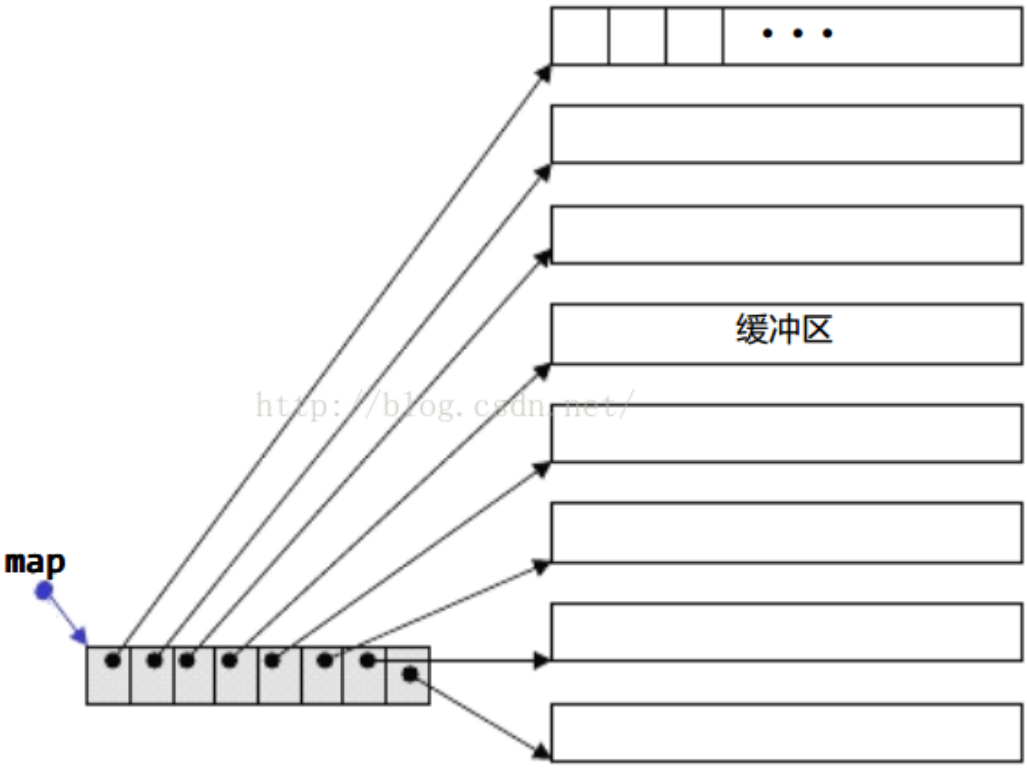


- 常用函数

```
list.push_back(elem)     在尾部加入一个数据
list.pop_back()          删除尾部数据
list.push_front(elem)    在头部插入一个数据
list.pop_front()         删除头部数据
list.size()              返回容器中实际数据的个数
list.sort()              排序，默认由小到大
list.unique()            移除数值相同的连续元素
list.back()              取尾部迭代器
list.erase(iterator)     删除一个元素，参数是迭代器，返回的是删除迭代器的下一个位置
```

deque

- 底层原理：deque是一个双向开口的连续线性空间（双端队列），在头尾两端进行元素的插入跟删除操作都有理想的时间复杂度。



- vector可以随机存储元素（即可以通过公式直接计算出元素地址，而不需要挨个查找），但在非尾部插入删除数据时，效率很低，适合对象简单，对象数量变化不大，随机访问频繁。除非必要，我们尽可能选择使用vector而非deque，因为deque的迭代器比vector迭代器复杂很多。list不支持随机存储，适用于对象大，对象数量变化频繁，插入和删除频繁，比如写多读少的场景。需要从首尾两端进行插入或删除操作的时候需要选择deque。
- 常用函数

```
priority_queue<int, vector<int>, greater<int>> pq;   最小堆
priority_queue<int, vector<int>, less<int>> pq;      最大堆
pq.empty()    如果队列为空返回真
pq.pop()      删除对顶元素
pq.push(val) 加入一个元素
pq.size()     返回优先队列中拥有的元素个数
pq.top()      返回优先级最高的元素
```

stack/queue

- stack与queue都是以deque为底层数据结构而实现的适配器，主要实现了栈与队列这两种数据结构特性。
- 常用函数

```
it.empty()
it.size()
it.push()
it.pop()
stack.top()
queue.front()
queue.back()
```

heap/priority_queue

- heap：并不属于STL的容器，其作为priority_queue的底层数据结构，而其实现机制中的max-heap实际上是以一个vector表现的完全二叉树。
- 常用函数

```
make_heap(_First, _Last, _Comp)   // 建立一个堆
push_heap (_First, _Last)         // 先在容器中加入数据，再调用push_heap ()
pop_heap(_First, _Last)           // 要先调用pop_heap()再在容器中删除数据
sort_heap(_First, _Last)          // 堆排序
```

- priority_queue：priority_queue：优先队列，其底层是**用堆来实现**的。在优先队列中，队首元素一定是当前队列中优先级最高的那一个。
- 常用函数

```
priority_queue<int, vector<int>, greater<int>> pq;   最小堆
priority_queue<int, vector<int>, less<int>> pq;      最大堆
pq.empty()   如果队列为空返回真
pq.pop()     删除对顶元素
pq.push(val) 加入一个元素
pq.size()    返回优先队列中拥有的元素个数
pq.top()     返回优先级最高的元素
```

map/set/mutilset/mutilmap

- 底层结构：epoll模型的底层结构也是红黑树，linux系统中CFS进程调度算法，也用到红黑树。

  红黑树特性：

  - 每个结点或是红色或是黑色；
  - 根结点是黑色；
  - 每个叶结点是黑的；
  - 如果一个结点是红的，则它的两个儿子均是黑色；
  - 每个结点到其子孙结点的所有路径上包含相同数目的黑色结点。

  对于STL里的map容器，count方法与find方法，都可以用来判断一个key是否出现，`mp.count(key) > 0`统计的是key出现的次数，因此只能为0/1，而`mp.find(key) != mp.end()`则表示key存在。
- 特点：set和multiset会根据特定的排序准则自动将元素排序，set中元素不允许重复，multiset可以重复。map和multimap将key和value组成的pair作为元素，根据key的排序准则自动将元素排序（因为红黑树也是二叉搜索树，所以map默认是按key排序的），map中元素的key不允许重复，multimap可以重复。因为存储的是结点，不需要内存拷贝和内存移动。其插入与删除效率比其他序列容器高。

unordered_map、unordered_set

- 底层数据结构

  unordered_map的底层是一个防冗余的哈希表（采用除留余数法）。哈希表最大的优点，就是把数据的存储和查找消耗的时间大大降低，时间复杂度为O(1)；而代价仅仅是消耗比较多的内存。通过设计哈希函数，使得每个函数的key都与一个函数值相对应，于用这个数组单元来存储元素。一般采用开链的方式来处理冲突。（通常buckets vector size为质数，可以避免冲突）
- hash冲突

- 开放定址法：线性探测法、平凡探测法d[i]+1^2、d[i]+2^2.....、伪随机探测再哈希
- 链地址法
- 再哈希法
- 建立公共溢出区

迭代器的底层机制

迭代器是连接容器和算法的一种重要桥梁，通过迭代器可以在不了解容器内部原理的情况下遍历容器。它的底层实现包含两个重要的部分：萃取技术和模板偏特化。

萃取技术可以进行类型推导，根据不同类型可以执行不同的处理流程，萃取出所需要的数据类型，萃取技术进行类型推导的过程会使用到模板偏特化，模板偏特化可以用来推导参数，如果我们自定义了多个类型，除非我们把这些自定义类型的特化版本写出来，否则我们只能判断他们是内置类型，并不能判断他们具体属于是个类型

迭代器类型

- 输入迭代器
- 输出迭代器
- 前向迭代器
- 双向迭代器
- 随机访问迭代器

vector中erase方法与algorithn中的remove方法区别

- vector中erase方法真正删除了元素，迭代器不能访问了
- remove只是简单地将元素移到了容器的最后面，迭代器还是可以访问到。因为algorithm通过迭代器进行操作，不知道容器的内部结构，所以无法进行真正的删除。