一、STL 六大组件

STL 提供六大组件，彼此可以组合套用：

1.容器（containers）：各种数据结构，如 vector,list,deque,set,map

2.算法（algorithms）：各种常用算法，如 sort,search,copy,erase

3.迭代器（iterators）：扮演着容器与算法之间的粘合剂，本质是一个 class template,所有 STL 容器都附带自己专属的迭代器

4.仿函式（functors）：重载了 operator()的 class 或 class template

5.配接器（adapters）：用来修饰容器、仿函数或迭代器接口的东西。

6.配置器（allocators）：负责空间配置和管理，实现了一个动态空间配置、空间管理、空间释放的 class template。

六大组件的关系：container 透过 allocators 取得数据存储空间，Algorithm 透过 iterator 存取 container 内容，functor 协助 algorithm 完成不同的策略变化，adapter 修饰 functor

二、空间配置器（单例模式，一个程序只需要一个空间配置器）

每个容器都已经预定其预设的空间配置器，如：

```
template <class T, class Alloc = alloc>
class vector {};
```

内存分配规则：

如果用户需要的区块大于 128，则直接调用第一级空间配置器（直接 malloc()和 free()）

如果用户需要的区块大于 128，则到自由链表中去找 （启用二级空间配置器）

如果用户需要是一块 n 字节的区块，且 n <= 128（调用第二级配置器），此时 Refill 填充是这样的：（需要注意的是：系统会自动将 n 字节扩展到 8 的倍数也就是 RoundUP（n），

再将 RoundUP（n）传给 Refill）。用户需要 1 块 n 大小的内存，且自由链表中没有，因此系统会向内存池申请 nobjs * n 大小的内存块，默认 nobjs=20

如果内存池大于 nobjs * n，那么直接从内存池中取出，1 块给用户，19 块给 free_list

如果内存池小于 nobjs * n，但是比一块大小 n 要大，那么此时 1 块给用户，其他的块数给自由链表。

如果内存池连一个区块的大小 n 都无法提供，那么首先先将内存池残余的零头给挂在自由链表上，然后向系统 heap 申请(40 倍大小的客户需求的满 8 倍数)空间，1 个给用户，19 个给 free_list，20 个留在内存池，申请成功则返回，

申请失败则到自己的自由链表中看看还有没有可用区块返回，如果连自由链表都没了最后会调用一级配置器。

自由链表是一个指针数组，有点类似与 hash 桶，它的数组大小为 16，每个数组元素代表所挂的区块大小，比如 free_list[0]代表下面挂的是 8bytes 的区块，free_list[1]代表

下面挂的是 16bytes 的区块…….依次类推，直到 free _ list[15]代表下面挂的是 128bytes 的区块同时我们还有一个被称为内存池地方，以 start_free 和

end_free 记录其大小，用于保存未被挂在自由链表的区块，它和自由链表构成了伙伴系统。

三、迭代器概念与 traits 编程技法（迭代器是个模板类）

3.1 迭代器思维

迭代器是将容器（数据结构）和算法的粘合剂。

3.2 迭代器是一种 smart pointer

3.3 traits 编程技法：偏特化

四、序列式容器

4.1 vector（连续的线性空间，支持快速随机访问）

vector 实现源码：

```
template <class T, class Alloc = alloc>
class vector
{
public:
    typedef T value_type;
    typedef value_type* pointer;
    typedef value_type*       iterator;
    typedef value_type& reference;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
protected:
    iterator start;
    iterator finish;
    iterator end_of_storage;

    void insert_aux(iterator position, const T& x);
    void deallocate(){
        if(start){
            data_allocator::deallocate(start, end_of_storage - start);
        }
    }
    void fill_initialize(size_type n, const T& value) {
        start = allocate_and_fill(n, value);
        finish = start + n;
        end_of_storage = finish;
    }
public:
    iterator begin() {return start;}
    iterator end() {return finish;}
    size_type size() const {return size_type(end() - begin());}
    size_type capacity() const{ return size_type(end_of_storage - begin()); }
    bool empty() const {return begin() == end();}
    reference operator[](size_type n){ return *(begin() + n); }
```

```
vector() : start(0),finish(0),end_of_storage(0){}
vector(size_type n, const T& value) { fill_initialize(n, value); }
vector(int n, const T& value) { fill_initialize(n, value); }
vector(long n, const T& value) { fill_initialize(n, value); }
explicit vector(size_type n) { fill_initialize(n, T()); }

~vector()
{
    destory(start,finish);
    deallocate();
}
reference front(){return *begin();}
reference back(){return *(end() - 1)}
void push_back(const T& x)
{
    if(finish != end_of_storage){
        construct(finish,x);
        ++finish;
    }
    else{
        insert_aux(end(),x);
    }
}
void pop_back(){
    --finish;
    destory(finish);
}
iterator erase(iterator position) {
    if (position + 1 != end())
        copy(position + 1, finish, position);
    --finish;
    destroy(finish);  // 全域函式，见 2.2.3 节。
    return position;
}
void resize(size_type new_size, const T& x) {
    if (new_size < size())
    erase(begin() + new_size, end());
else
    insert(end(), new_size - size(), x);
}
```

```
    void resize(size_type new_size) { resize(new_size, T()); }
    void clear() { erase(begin(), end()); }
};
```

4.2 list（环型双向链表，支持快速增删）

push_front,push_back,erase,pop_front,pop_back,clear,remove,unique,merge,reverse,sort

4.3 deque（一个中央控制器和多个缓冲区，支持收尾快速增删，也支持随机访问）

4.4 stack（deque 或 list 实现）

先进后出，stack 没有迭代器，只有顶端元素才能被外界取出，不提供走访功能。

4.5 queue（deque 或 list 实现）

先进先出，queue 没有迭代器，只有顶端元素才能被外界取出，不提供走访功能。

4.6 heap（堆，最小堆，最大堆）

堆是一颗完全二叉树（对于一个树高为 h 的二叉树，如果其第 0 层至第 h-1 层的节点都满。如果最下面一层节点不满，则所有的节点在左边的连续排列，空位都在右边。这样的二叉树就是一棵完全二叉树），

将堆存放在 vector（array）中，技巧：将 array 的 #0 元素保留（或设为无限大值或无限小值），那么当 complete binary tree 中的某个节点位于 array 的 i 处，其左子节点必位于 array 的 2i 处，

其右子节点必位于 array 的 2i+1 处，其父节点必位于「 i/2 」处（此处的「」权且代表高斯符号，取其整数）。通过这么简单的位置规则， array 可以轻易实作出 complete binary tree

最大堆:

push_heap 算法：入堆，首先 push 到 end()，然后与父节点比较，不满足则交换父子节点，不断往上，直到其没有父节点或满足条件。

pop_headp 算法：出堆，首先 pop()，取走根节点，用最后一个节点填补，然后与 2 个子节点比较，并与较大的子节点交换位置，直到没有子节点或满足条件。

sort_heap 算法：堆排序，不断地对 heap 做 pop 动作，便可达到排序效果。

4.7 priority_queue（带权值的队列）

利用 heap 实现，每次弹出的都是权值最高的。

4.8 slist（双迭代器，list 只有单向迭代器）

4.9 总结

vector 底层数据结构为数组 ，支持快速随机访问

list 底层数据结构为双向链表，支持快速增删

deque 底层数据结构为一个中央控制器和多个缓冲区，支持首尾（中间不能）快速增删，也支持随机访问

stack 底层一般用 list 或 deque 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时

queue 底层一般用 list 或 deque 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时

（stack 和 queue 其实是适配器,而不叫容器，因为是对容器的再封装）

priority_queue 的底层数据结构一般为 vector 为底层容器，堆 heap 为处理规则来管理底层容器实现

set 底层数据结构为红黑树，有序，不重复

multiset 底层数据结构为红黑树，有序，可重复

map 底层数据结构为红黑树，有序，不重复

multimap 底层数据结构为红黑树，有序，可重复

hash_set 底层数据结构为 hash 表，无序，不重复
hash_multiset 底层数据结构为 hash 表，无序，可重复
hash_map 底层数据结构为 hash 表，无序，不重复
hash_multimap 底层数据结构为 hash 表，无序，可重复