1. 空间配置器部分

* 为什么要设计空间配置器？（重点）

因为主要是是想把空间分配和初始化工作分开，例如当使用new创建一个对象的时候，A\* a=new A()；通常就会先执行operator new()函数来分配内存空间，之后使用A的构造函数初始化这段空间，这样做会带来一些额外的开销，比如我只想获得10个A大小的空间 ，但是调用new后，不仅给了10个大小的A空间，而且还把这些空间初始化，这显然不是我们想要的，所以想设计空间配置器把空间的分配和初始化工作分开。另外一个原因我觉得还是处于效率考虑的，毕竟STL就是要追求极致的效率，通过设计二级分配的机制可以有效的管理小块的内存，防止出现太多的内存碎片。第一级空间配置器负责构建和回收大于128bytes的资源，而第二级空间配置负责维护free\_list，free\_list是一个指针数组，他维护了了16条链表（8-128），链表的节点是union，既可以表示为链表节点，又可以表示为分配出去的内存。通过维护free\_list二级配置器可以有效的管理小的内存区块。

* free\_list的使用（难点）

free\_list的定义为static obj\* volatile free\_list[EFreeLists::NFREELISTS]；可以看到free\_list是一个指针数组，虽然指针数组和数组指针的概念已经很熟悉了，但是刚刚接触free\_list的管理还是很头疼。总结一下free\_list管理的一些细节使用

obj\* volatile\* my\_free\_list; //这个指针用于指向free\_list其中的一项，free\_list里面存的是指向下一个内存的地址（指针）。

my\_free\_list=free\_list+FREELIST\_INDEX(n);//把my\_free\_list定位到适合的位置

\*my\_free\_list 得到的结果是一个地址，也就是my\_free\_list中存的内容（其中存的是地址）。\*\*my\_free\_list就得到了链表中的各个节点的具体实例了。

其实my\_free\_list=free\_list+FREELIST\_INDEX(n)这句可以换成

obj\* res=free\_list[FREELIST\_INDEX(n)]，这样更好理解。

1. 迭代器部分

迭代器部分主要是设计迭代器萃取器和类型萃取器。真正的迭代器是要和容器紧密联系在一起的，设计容器的时候也要设计其迭代器。迭代器模式是设计模式的一种。

迭代器萃取器主要是为了复杂获取到迭代器所指向内容的一些类型信息及其他的信息，为算法的使用提供一些方便，毕竟算法不直接和容器交互，是通过迭代器进行一些操作，所以有必要通过迭代器获取一些容器内的信息。但是要注意元素指针并不是类，所以他并没有嵌套类型，所以需要针对原生指针以及const原生指针设计迭代器萃取器的特化版本。

类型萃取器，类型萃取器可以获得类型是否具有trivial信息或者是否是POD类型，因为类型的这些信息关系到类型创建和销毁是不是可以直接在内存上进行操作，直接在内存上进行操作会很省时。类型萃取器需要设计一个模板版本，这个版本用于用户自定义类型的类型萃取，另外需要设计内置类型进行原生指针的特化版本。

1. Vector

Vector是一个比较简单的容器。其实vector的设计都是围绕着3个指针进行的，start，finish，end\_of\_storage进行的，vector 的实现并没有什么难点，但是注意一些小的知识点。

1. 使用const进行重载：如，iterator begin() {return start;}和const\_iterator begin() const{return start;}，因为前一个函数有一个隐藏的T\* this，后一个函数有一个隐藏的const T\* this，因此可以通过const进行重载
2. 为什么拷贝构造函数的参数可以访问私有成员？其实这块之前一直也没注意过，类成员的访问限定符针对的是类，而不是类的对象，因此在类作用域内，类的对象可以不加限制的访问私有成员。
3. 另外，vector的操作中只有insert的几个相关操作不容易实现，但是通过画图分析，也可以很快写出正确的实现，其本质也是对那3个指针进行处理。