OOP代码阅读报告

计71 张程远 2017011429

0内容简述

报告的选题是SGI STL源码的阅读。报告内容包括三部分：STL整体架构、算法（Algorithm）部分的工作流程以及代码所使用的面向对象技术。算法部分选择了一些比较复杂的算法进行具体解释。

1整体架构

STL由6个部分组成：adapter（适配器）、algorithm（算法）、allocator（配置器）、container（容器）、functor-function object（仿函数）、iterator（迭代器）。

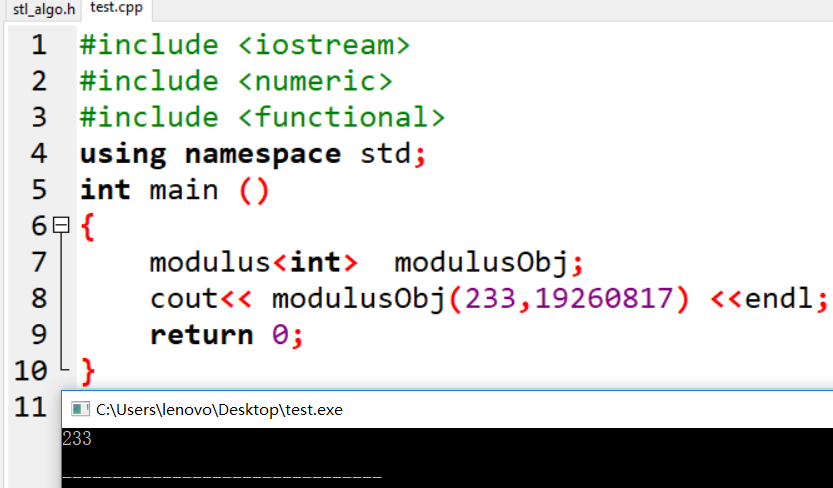
容器：包括关联容器（如map, set等通过映射来访问元素的容器）与顺序容器（如vector，stack等通过元素在容器的顺序来访问元素的容器）两部分，用于储存数据。

算法：包含对数据进行一系列操作的函数，如加减乘除、求内积、排序等。

配置器：管理对象的内存分配以及删除，可以实现空间分配功能。

仿函数：所谓仿函数是指一些模拟函数效果的类，它们通常是一些struct，每个struct重载了括号运算符，以达到使用时模拟函数的调用效果。（其实就是函数对象）仿函数的具体作用在于很多函数在执行功能时可能存在重合的部分，functor将这些部分的代码抽象成为仿函数类，方便重复使用，也可以减少函数指针的使用。

使用例子如：



输出233对19260817取模的结果233，这里的modulus Obj是一个modulus类的对象而不是函数。

适配器：用于规范容器、迭代器、仿函数的接口。

迭代器：提供一种能够遍历容器中元素而又与容器内部实现无关的方法。通过迭代器方法，原本彼此独立的容器和算法部分结合在一起，从而能够完整地实现功能。

六部分的交互方式及联系：首先容器通过配置器的分配作用取得内存；之后算法部分可通过迭代器读取容器中的数据，并对这些数据进行排序、反转等操作；仿函数可运用于算法具体实现当中使得代码得到简化；适配器能够规范容器、迭代器、仿函数等的接口。这样的交互方式意义在于，大大降低了具体数据结构与具体算法之间的耦合度，使得同一套算法对于不同容器、不同迭代器都能够适用，避免了代码的大量修改，使得算法代码能够最大程度复用。

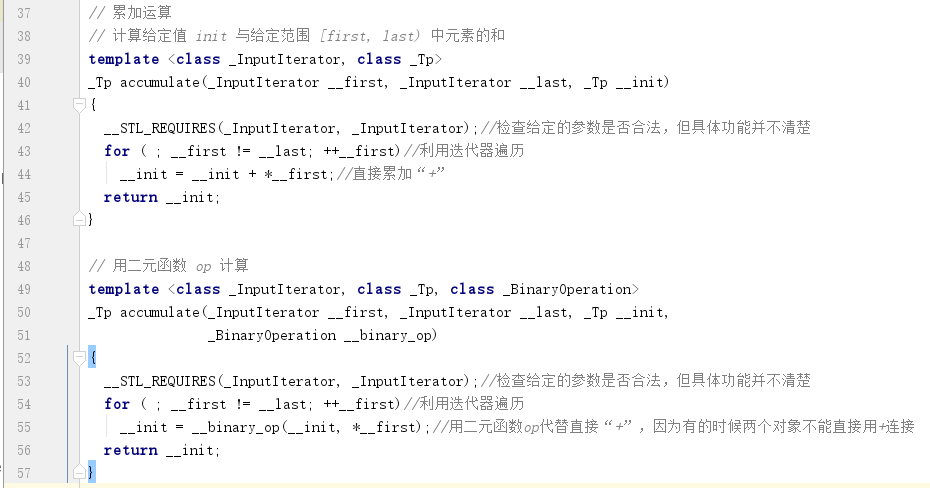
2算法解释

在STL代码中，算法可分为”numeric”（数值算法）、“algobase“（基础算法）以及”algo”（其他算法）三部分。下面将选取一些算法进行解读和分析。

2.1 numeric部分

该部分提供了一些针对数值运算功能的算法，如比大小、求累加和等。需要注意的是，有的时候默认的运算不能满足用户的所有需求，因此在algorithm里每个算法都提供了两个版本——能够直接使用符号运算和采用自定义函数（仿函数）进行运算的版本。

2.1.1 累加与求内积



累加算法，计算[first, last) 区间内所有元素与给定值Init的和。版本一是默认版本，使用的是默认的加号，版本二则允许用户定义自己的累加运算\_\_binary\_op，这样一方面可以通过函数使得一些不能直接相加的对象也能够使用累加算法，另一方面也可以让用户定义自己的累加运算，例如定义x+2y为和等，大大增强了默认版本代码的可扩展性。求内积的模式与累加基本相同，由于只涉及二元运算，默认的加法和乘法分别由两个自定义二元运算\_\_binary\_op1与\_\_binary\_op2替代。

在局部求和算法中，也允许用户自定义求和方法，除此之外还稍有一点不同。



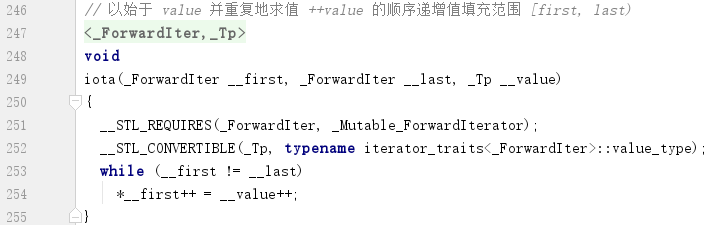
该算法的功能是计算前缀和，计算结果Y[i]为X0累加到Xi的和，如果能够确定传入数据的类型，则直接使用第一个默认函数即可；若不行，可以使用第二个函数，判断和是否为空的同时获得\_\_first的类型，以便调用第一个函数进行计算。\_\_VALUE\_TYPE由头文件type\_traits.h提供。

2.1.2 求相邻元素的差



该算法的功能是计算相邻两项的差，计算结果Y[i]=X[i+1]-X[i]。使用的方法和前面累加、 求内积算法基本相同。

2.1.3 递增填充（iota）



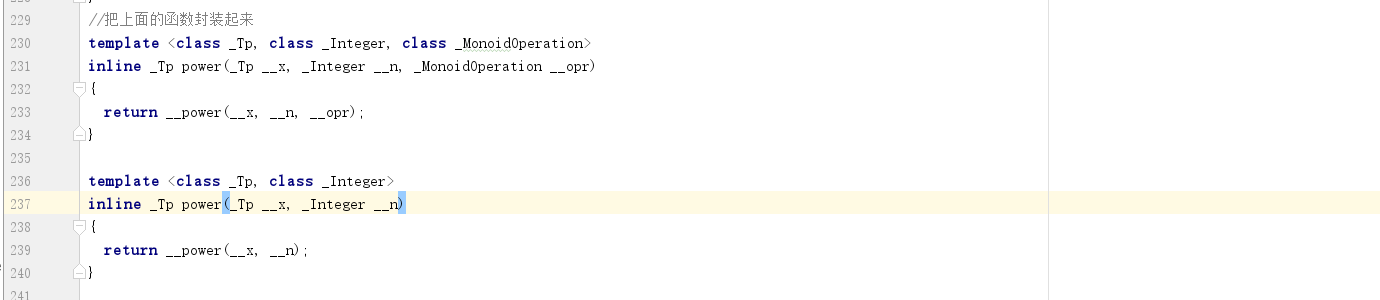
该算法的功能是从某一个value开始递增，并将值依次填入【first, value) 中。

2.1.4 幂次方（power）



该算法的功能是求x^n，当n>=0时返回其值。内部使用了快速幂算法，大大降低了时间复杂度。第一个版本中乘法运算由opr代替；第二个版本是朴素的快速幂，使用functor里的multiplies函数对象实现对第一个函数的调用。这里的opr不一定要符合交换律（opr (result, x)不一定等于opr(x, result)），具体应用如向量的快速幂，但要符合结合律。将求幂运算与实际乘法运算分离，提高了代码的兼容性和可扩展性。

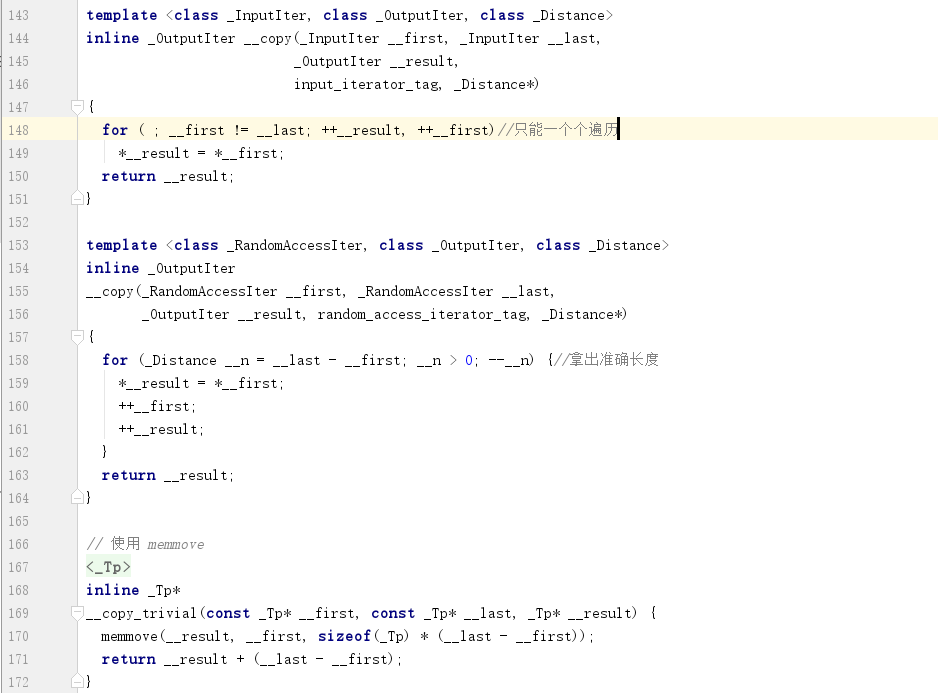
下面的部分则是对上面的两个函数进行封装。



2.2 algobase部分

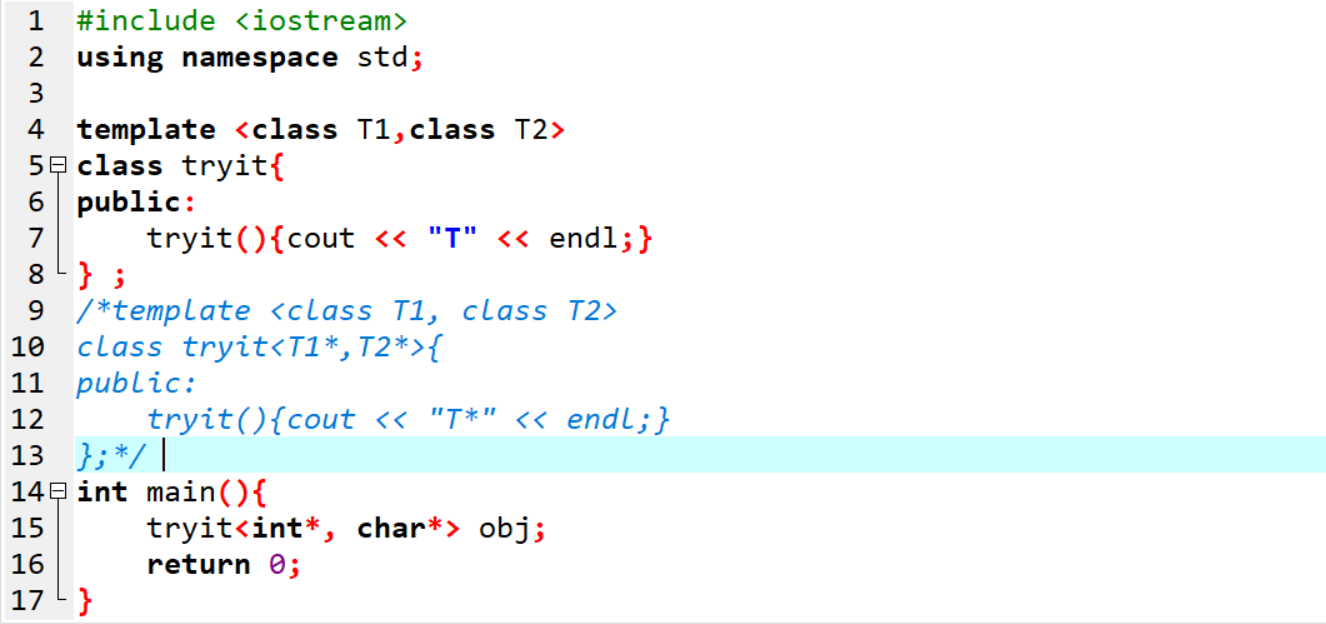
这部分提供了一些基本算法，包括equal, copy, fill, compare等功能。这里将重点阐述对于copy，equal, lexicographical\_compare的理解。

2.2.1 copy

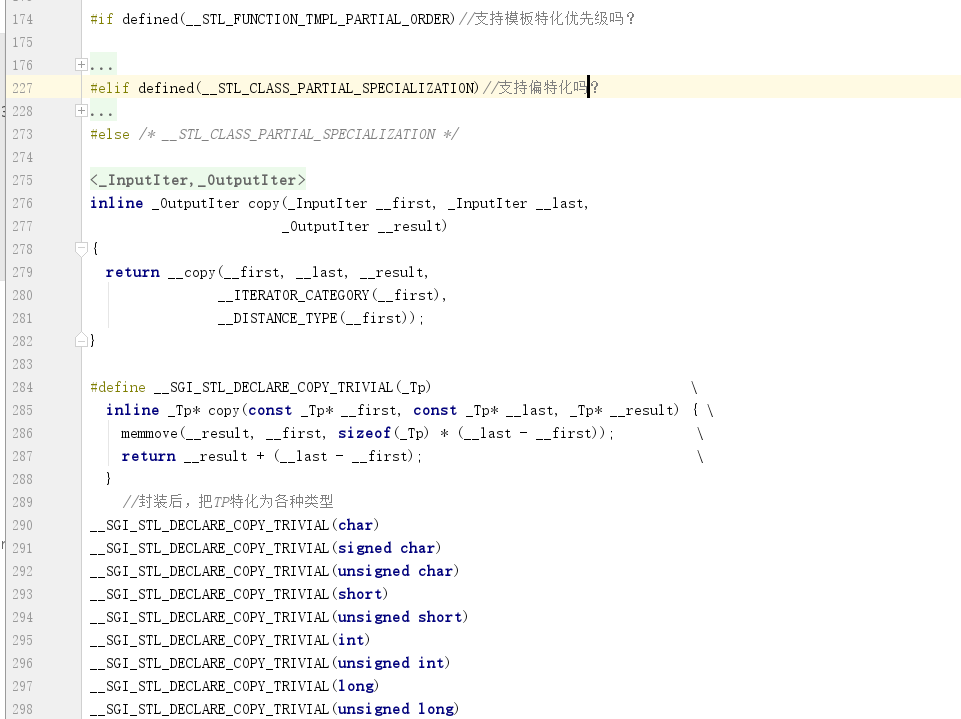


Copy函数能够将[first, last)数据拷贝到[result, result+last-first)中。这里实现了3个版本：如果传入的不是迭代器，调用平凡的copy，这里使用了memove函数：void \*memmove( void \*dest, const void \*src, size\_t count)，可将src所指内存区域的count个字符复制到dest所指区域。Memmove是移动构造，使用也比较简便，因此能使用则尽量使用。如果传入了迭代器，情况又分两种：一种是InputIter，这种迭代器只能一个个遍历，如链表；另一种是随机迭代器，这种迭代器满足+n操作，也可以直接访问中间的某个元素，如vector。如果是随机迭代器，那么就将循环写成含有两个迭代器之间准确距离的形式，这样使得迭代器很难越界，提高代码安全性。

除此之外，Copy函数针对支持模板偏特化特性及模板特化优先级特性的编译器提供了更高效率的优化版本。模板偏特化就是针对template参数的进一步限制所设计的版本。例如对于模板类class T，我们可以特化其为T\*、const T\*时执行的函数。如下图。



这里输出的结果是T，而去掉注释输出的结果是T\*。当模板特化和模板偏特化同时存在时，优先级也是一个需要考虑的问题，这里编译器通常遵守先特殊规定后一般规定的原则。Copy函数内部分别为支持两个特性的编译器提供了优化版本，并在优化版本内部还为未重载赋值运算符的情况设计了不同的函数，并提供了完全泛化的对外接口。最后Copy进行了封装并提供了各种数据类型的特化。



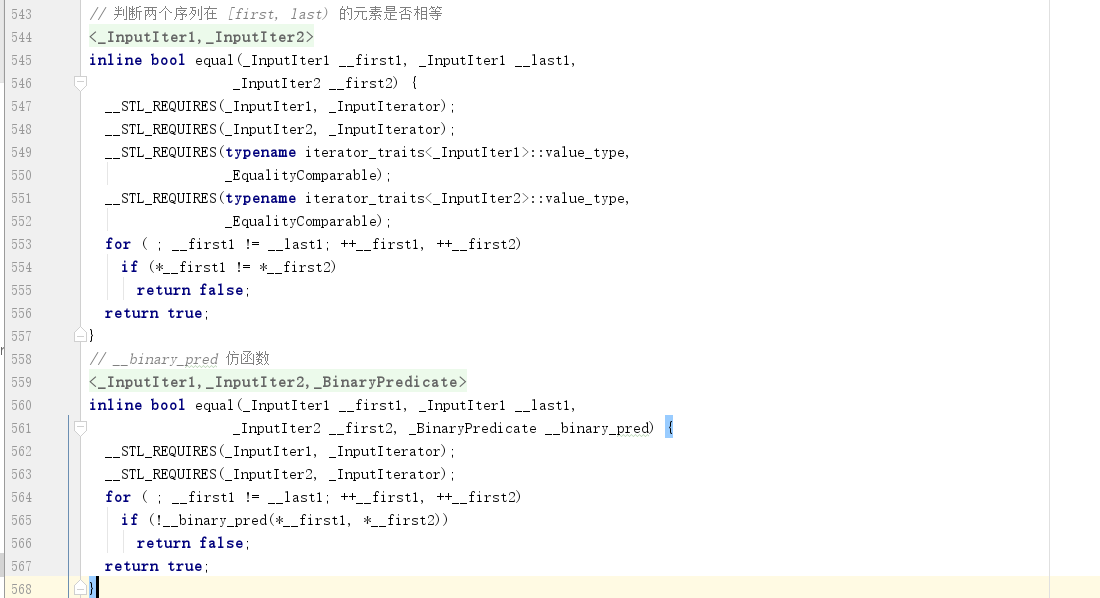
除基础的copy函数外，STL在algobase里还提供了copy\_backward（逆向复制）、copy\_n（从指定位置复制size个元素）等功能，其设计思想基本与Copy相同。可以说Copy函数的可扩展性、封装性和效率都是十分优秀的。

Fill与fill\_n的设计比较简单，思路基本与Copy相似，区别在于fill所提供的特化使用了memset。 由于memset的作用是对于内存地址的每一个字节进行修改，因此只适用于按字节填充的方式，如unsigned char\*的填充。对于signed char\*和char\*，fill使用了强制类型转换来把它们显式地转换为unsigned char\*。

2.2.2 mismatch and equal

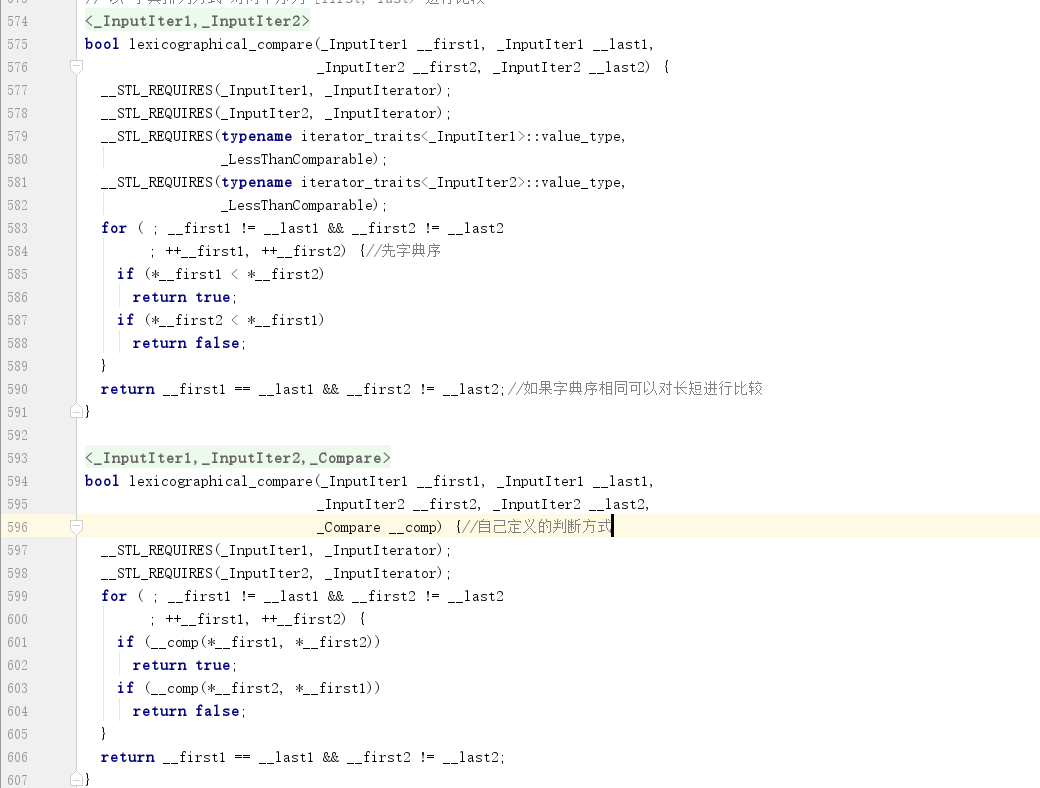


Mismatch函数可以判断两个序列从哪一个位置开始出现不同，并返回一对迭代器代表在两个序列里各自的位置。默认的相同是指operator==，但同样它也通过提供仿函数参数的方式允许用户自主定义什么是“不同”。

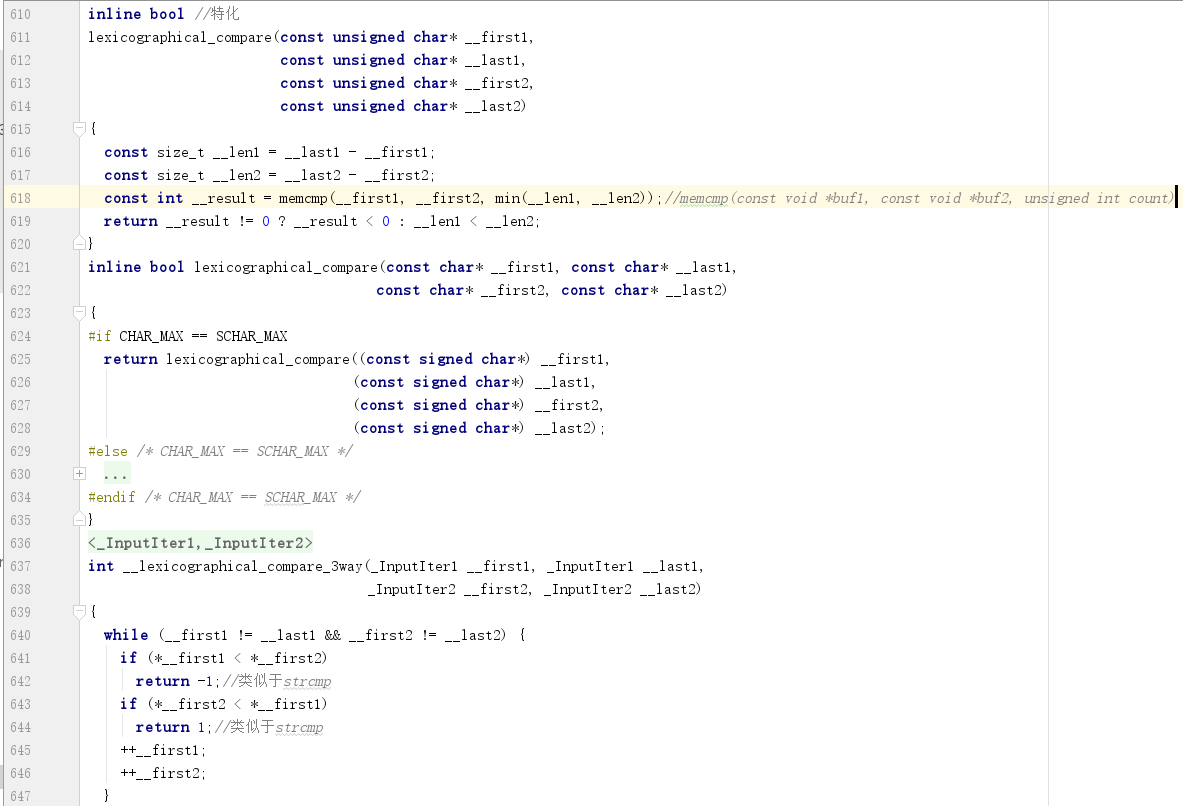


Equal是一个bool类型，能够检查两个序列是否完全一样。但如果两个序列长度不一样，只能检查长度相同的部分。与mismatch一样，它也可以允许用户自定义比较规则。

2.2.3 lexicographical\_compare



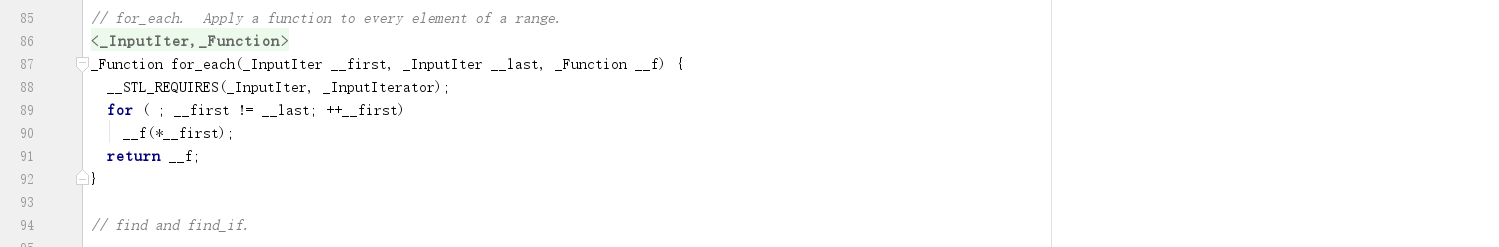
该函数的功能是按照字典序对一段序列进行比较。如果字典序相等，则还要比较长度，即1序列的头尾迭代器重合时2序列的头迭代器是否达到了尾部。允许用户自定义比较方式。与fill一样，它也有针对char\*的特化以提高效率，使用的是memcmp函数。同时它还提供了类似于strcmp的泛化版本，在1序列小于2序列时返回-1，大于2序列则返回1。见下图。



2.3 其他算法（algo部分）

<stl\_algo.h>文件里包含了很多算法，大致上可分为查找、复制、排序、修改、合并等操作，还有对set、heap等结构进行专门操作的函数。下面将选取一些比较常见的函数进行分析。

2.3.1 for\_each



For\_each的功能是将仿函数\_\_f应用于[first, last)的每一个元素。但是\_\_f无法改变[first, last）里元素的值，所以不能用for\_each来修改容器内元素值，只能把元素取出来再进行操作，原有的容器内元素值不变。

2.3.2 find与search



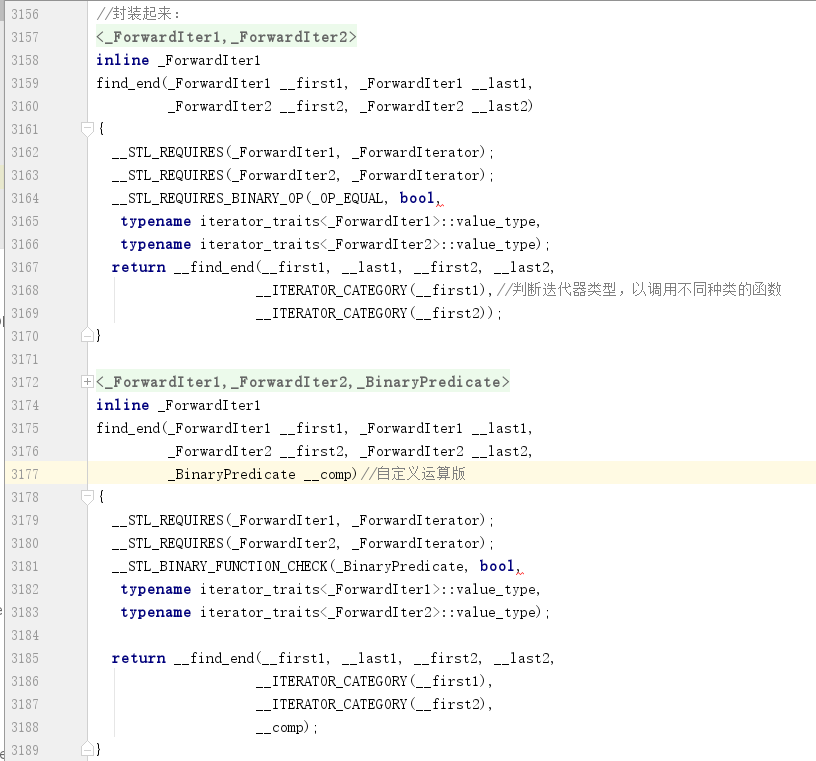
上图为adjacent\_find函数实现。

STL里提供了丰富的查找函数。find可用于查找序列里第一个与目标值相同的元素，并返回其位置；find\_if可用于查找符合条件的第一个元素，例如在vector中寻找第一个偶数，可调用find\_if(vec.begin(), vec.end(), IsEven); adjacent\_find可用于寻找序列中第一次重复的相邻元素，并返回相邻元素的第一个元素位置；find\_first\_of可用于寻找A序列和B序列第一个共有元素在A序列中出现的位置；find\_end则可以在B序列中寻找A序列（此时A作为B的一个子序列）最后一次出现的位置。

find\_end函数比较特殊，因为它是找最后一次出现的位置。在前面的find里面，传入的迭代器都是ForwardIter, 但是在find\_end里面传入ForwardIter显然会拖慢效率，因为无论是否找到了子序列都要把B序列从头到尾跑一遍。相比之下，传入BidirectionalIter（双向迭代器）会高效很多。而对于find\_end，STL也提供了对双向迭代器的偏特化版本，如下图：



下面是对find\_end的封装。



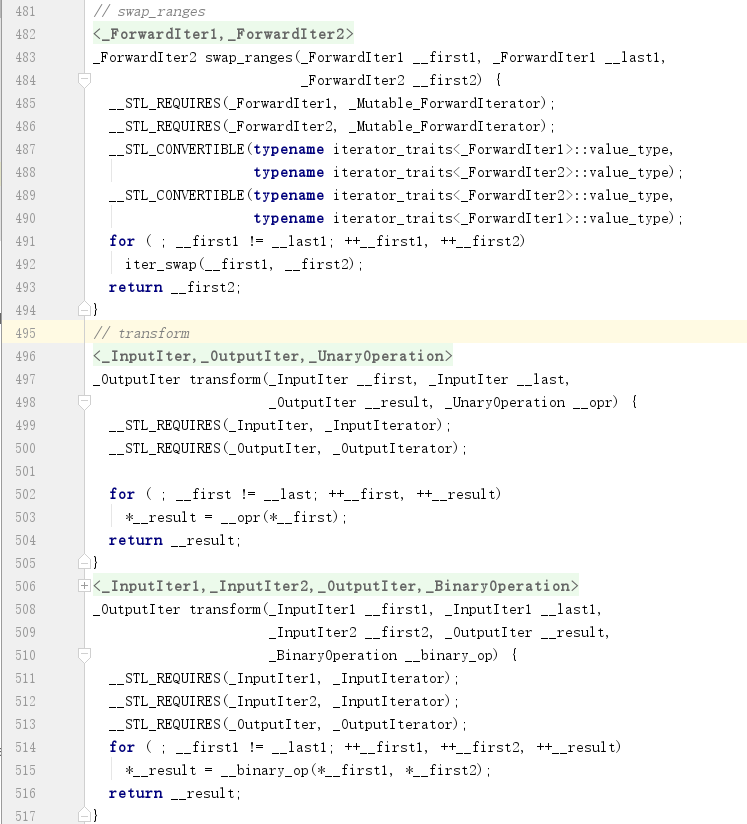
通过\_\_ITERATOR\_CATEGORY可以判断迭代器的类型，从而调用不同种类的函数。

在find\_end查找序列时用到了search函数，search也算是查找，故把search放在这里介绍。下面是find\_end调用的search代码。



Search实现的功能是在[first1, last1)里寻找[first2, last2)第一次出现的位置，并返回第一个元素所在的位置。其核心思想是把找序列转化为找元素，利用之前已经实现的find函数进行操作。首先排除掉特殊情况以提高效率，之后一旦发现不可能成为子序列（如元素不匹配或者长度不够）就立即跳出。具体流程可见注释。其衍生的search\_n函数可用于在区间内寻找第一个连续符合同一条件值value的序列，并返回这段连续序列的头部。

2.3.3 swap\_range与transform



Swap\_range的功能是把[first1, last1)里的序列与另外一个序列中从first2开始相同个数的元素交换。两个序列可以在同一容器中或不同容器中，因此这个函数好像不太安全——一方面没有判断第二个序列的迭代器是否越界，一旦越界则交换结果就无法知道；另一方面如果在同一容器中并且两个序列有重叠的部分，那么交换结果也未必和预期一样。

Transform提供了两个版本，第一个版本是用仿函数\_opr作用于元素产生新的序列；第二个版本则是在两个序列之间建立映射，两个序列各出一个元素，经过二元运算的作用产生新的序列。

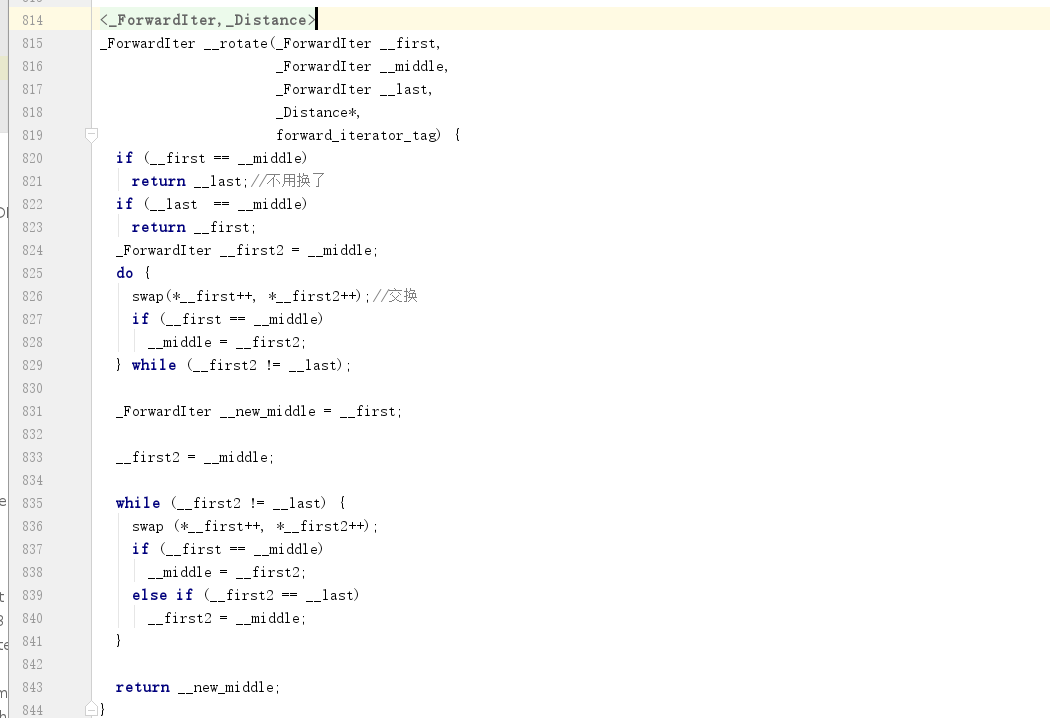
2.3.4 replace & remove

Replace与remove 各提供了4个版本的函数：默认版本，\_if, \_copy, \_copy\_if。

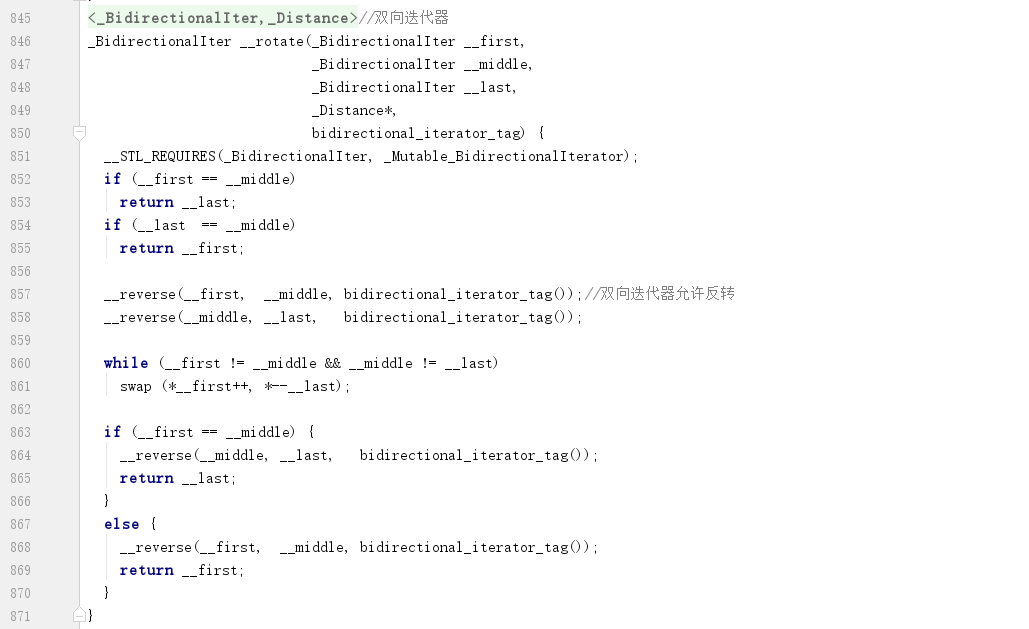
Replace默认版本是将区间内所有的旧值用新值替代；\_if版本则是提供一个判断仿函数，可将区间内所有判断值为true的旧值替换为新值；加入了copy的则是不改变原有容器内元素的值，而是将产生的新序列复制到一个新的容器中。新的容器可以与是原有的容器，也可以与原有容器重叠。Remove可以去掉与value值相等的元素，remove\_if可以去掉令判断函数为真的元素，但去掉的时候不改变容器的大小，也就是说在执行移除操作后将元素重新整理。实际上是调用了remove\_copy函数，将处理后的结果复制进原有的容器当中。

2.3.5 copy

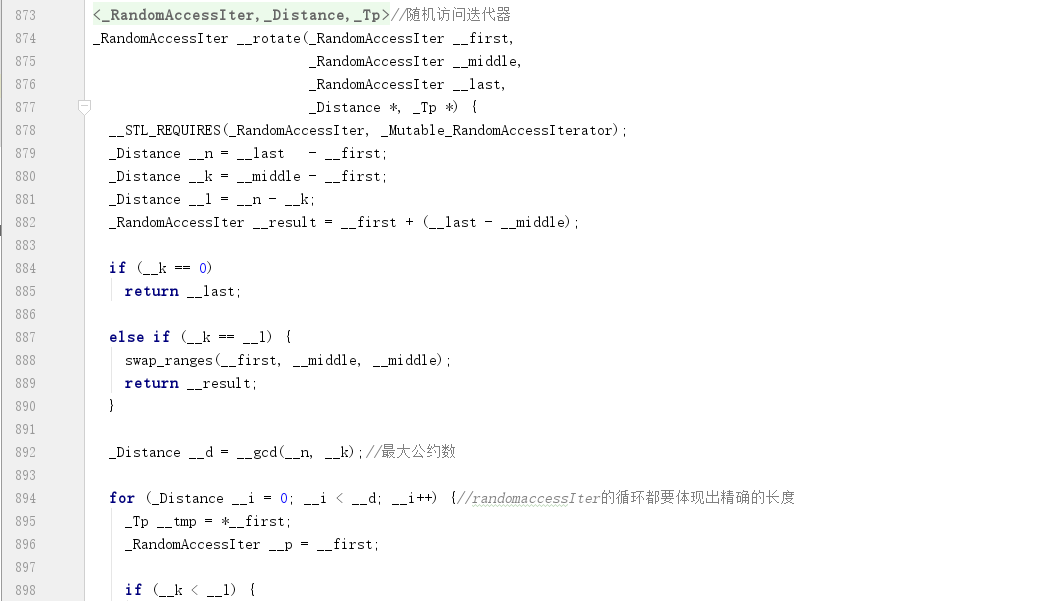
除去algobase里的copy之外，STL还在algo中提供了3种copy类型：unique\_copy, reverse\_copy, 以及rotate\_copy。Unique\_copy是在copy的同时，对于相邻的相同元素只取第一个；reverse\_copy是逆序复制；rotate\_copy则是对[first, middle)和(middle, middle+middle-first]进行交换+倒序。实现不同copy的前提是要实现unique, rotate, reverse三个函数，而对于不同的函数，使用不同的迭代器其效率也不同。因此，对每一种操作，STL都为不同迭代器提供了各自的特化版本。下图以rotate为例说明这一点。（在之前的find\_end中我们已经见过对于不同迭代器提供不同版本的设计）



上图：传入的迭代器类型为forward\_iterator\_tag，调用\_ForwardIter \_\_rotate；



上图：传入的迭代器类型为BidirectionalIter\_iterator\_tag，调用\_BidirectionalIter\_\_rotate；



上图：传入的迭代器类型为RandomAccessIter\_iterator\_tag，调用\_RandomAccessIter\_\_rotate。

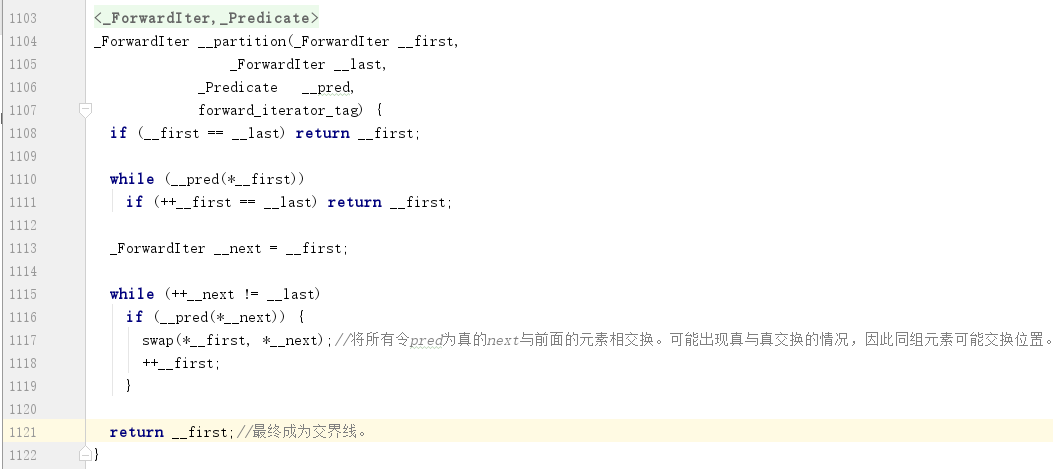
2.3.6 random

这部分的代码实现了随机数和随机排列的产生。Random\_number使用了rand，可以返回[0,n）的随机数值；random\_shuffle是对原有元素进行随机排列，它有两个版本：一个是用random\_number提供随机数，另一个允许用户使用自己写的随机数生成程序。



2.3.7 partition

用途：将使得某个判断函数为真的所有元素都放在区间前端。用这个函数可以将序列分组并返回两个组之间的边界位置。

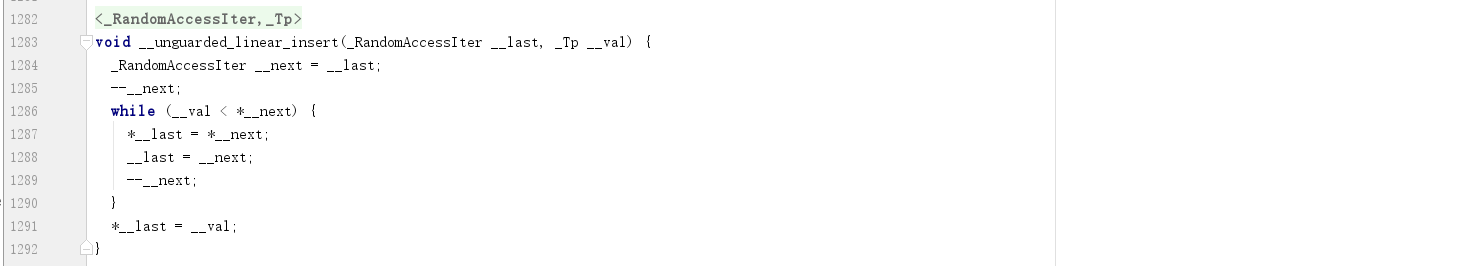


STL中还设计了unguarded\_partition函数，可以用来寻找这个枢纽值所在的位置。大致方法为设计两个迭代器first和last从头尾分别出发，在靠近彼此的同时看first与last跟枢纽值的比较，如果两者已经发生了交错则互换位置，直到双方找到枢纽位置。Unguarded冠名的函数里没有判断迭代器是否越界的语句，但这里两个迭代器是往数组中间走，所以理论上不会越界。Partition的设计为后面的快速排序打下了基础。

2.3.8 sort

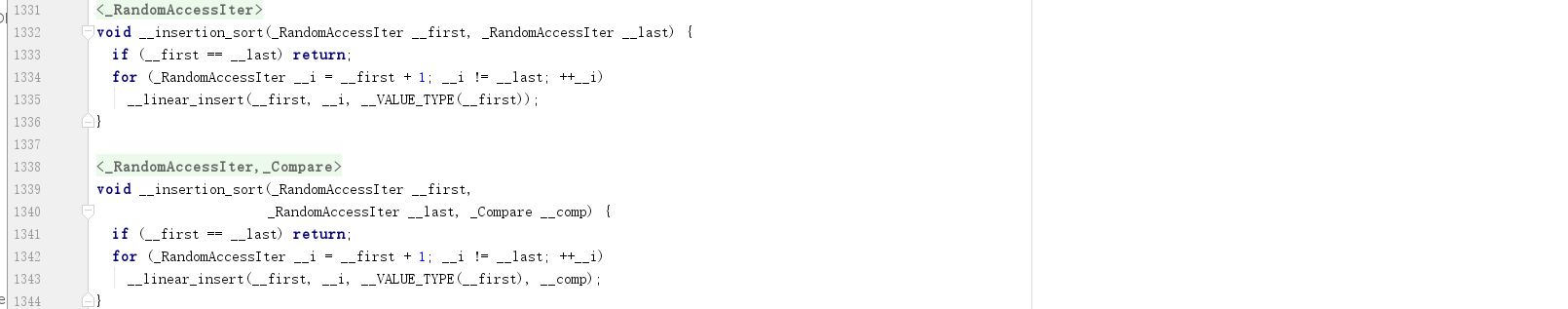
本部分将介绍sort及相关辅助函数的实现。

（1）线性插值：插入value，使得value到last里的元素都大于等于value。

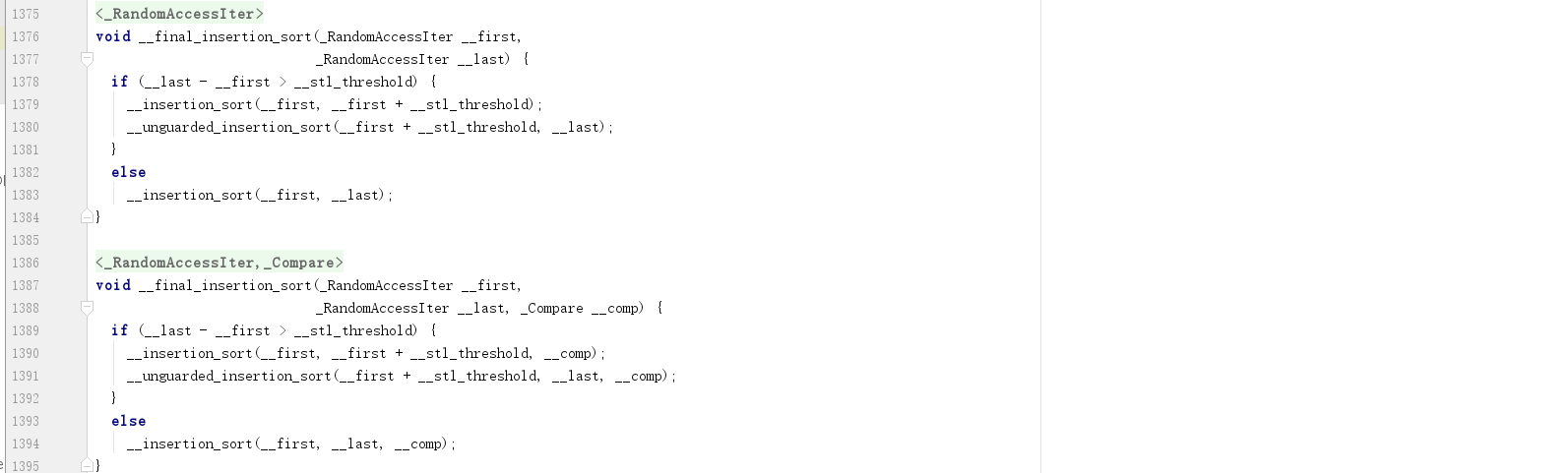


从数组尾端开始，置一前一后两个迭代器next和last，最终以last指向位置为value插入的地方。如果value比next还小，那就让两个迭代器分别前移。这样保证了value后端的值都大于等于value。Linear\_insert的功能稍有不同，它可以将尾部(last)的值插入到[first, last)中。Unguarded\_linear\_insert和linear\_insert是插入排序的基础。

（2）插入排序

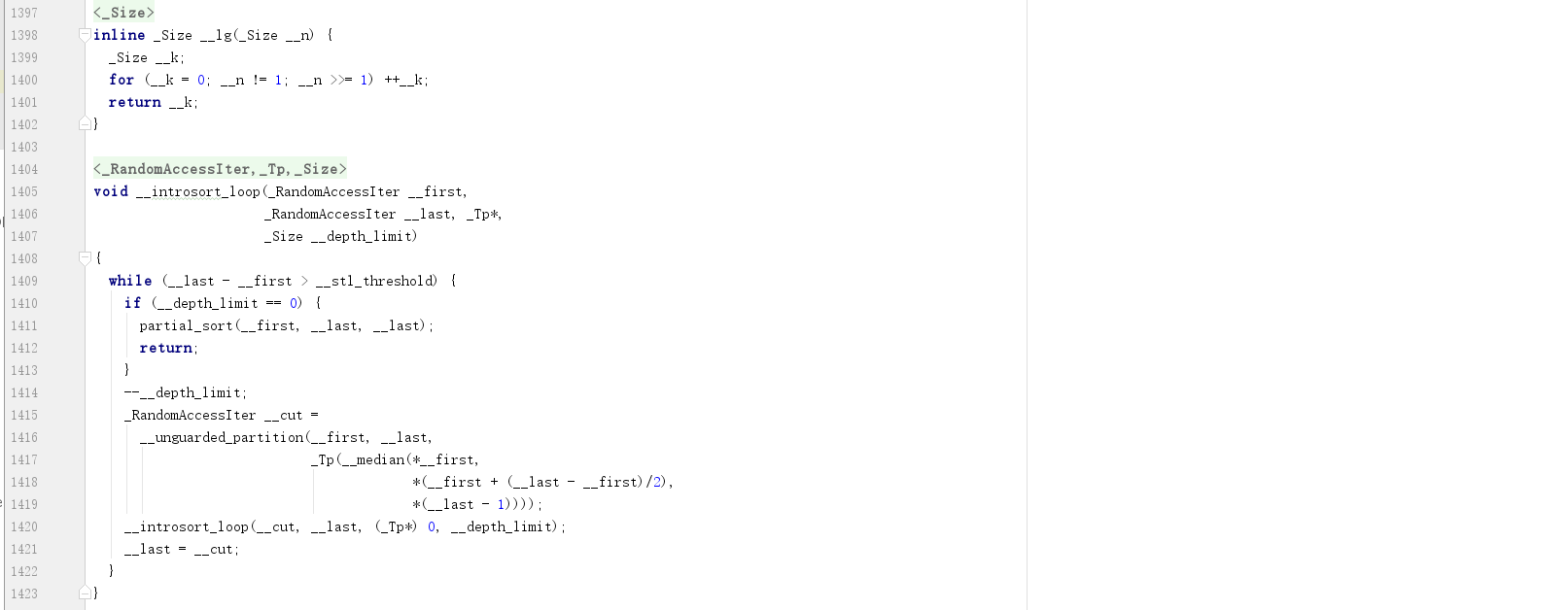


外部大循环每次决定一个子区间[first, i)，然后将i的值插入到前面的序列里。插入排序的复杂度虽然是N^2，但它对于长度较短的序列排序效率往往比快排高。所以在final\_insertion里面使用时，对于长度大于设定阈值（16）的序列，先把[first,first+16)的部分排序好，再将后面的元素插入进前面的元素序列。由于后段元素总是比前段元素大，因此效率也比较高。



（3）Introsort

所谓Introsort的原理实际上和快排差不多，但快排在最坏情况下的复杂度也可以达到N^2，为了解决这一问题，STL引入了Introsort，即对于Quicksort的优化版本。



快排的问题在于，对于数据量较小的序列，快排会为了产生极小的序列而调用较多的递归，开销比较大。而Introsort可以解决这个问题，它使用了lg函数来监控序列的递归次数。如果递归深度等于0，（也就是说产生了空的子区间，完全没达到分割的效果，出现了浪费效率的情况）转而调用效率更高的部分排序。在sort内部，长度小于等于16的序列将采用插入排序完成。如果没有出现最坏情况，则使用三数取中划分的方式进行递归排序，\_\_cut是利用前面的unguarded partition找到的基准点。

最终的sort封装：

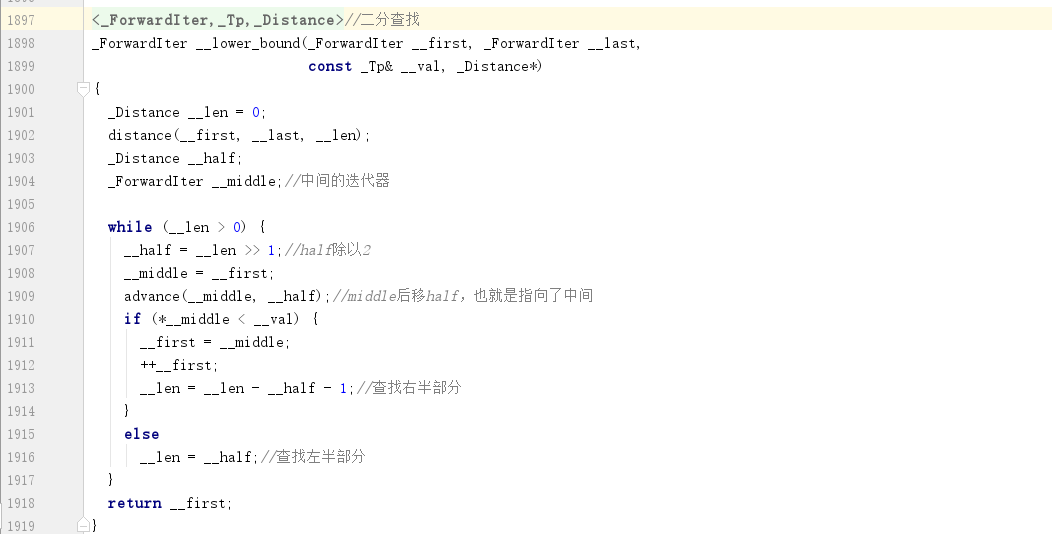


可以看出sort只接受随机访问迭代器。Sort能够通过评估序列的大小采用不同方式进行排序。数据量较大的时候采用Introsort，当数据量小于某个阈值的时候，为避免递归带来的不必要的开销则直接使用插入排序。这样做使得sort的效率达到了最大化。

2.3.9 二分查找

（1）lower\_bound 与upper\_bound

以lower\_bound为例，该函数的功能是查找序列中第一个不小于给定value的元素所在位置。



图为lower\_bound默认版本的源码，采用了二分查找的方式，比逐个查找效率要高很多。但是这个函数查找的前提是给定序列必须是有序的，否则将会出错。

（2）equal\_range

该函数的功能是找到与value相等的相邻元素的始末位置，返回一对迭代器。



同样要求序列是有序的。

（3）binary\_search

二分查找可用于寻找元素在序列中的位置，并返回该位置。实现时调用了lower\_bound。

2.3.10 permutation

STL里还提供了两个函数：next\_permutation和prev\_permutation。这两个函数可用于求区间内元素的排列，next是给定排列下一个比它大的排列，prev是给定排列上一个比它小的排列。



图以next\_permutation为例。每次从后往前找第一个比分界点大的元素并交换，保证了是原有排列的下一个排列。通过这个算法可以很方便地得出元素按字典序的下一个排列。

3 SGI STL所使用的部分面向对象技术

3.0 编程思想

STL的核心编程思想在于算法与数据结构的相互独立性，在实现算法的时候不让算法依附于任何具体的数据结构，最大限度地将算法抽象化，这减少了算法代码的冗余，也提高了算法和容器部分代码的可复用性。

3.1 模板

STL的源代码中大量使用了模板类，一方面使得代码适配多种数据结构，另一方面在编译时检查数据类型以确保其安全，且让代码具有很高的复用价值。另外STL代码中对于模板大量进行了偏特化处理，使得代码在保留模板类型无关的特性同时最大程度地优化效率。

3.2 仿函数

仿函数的使用允许用户能够自定义函数内部的具体操作，例如在sort里通过传入仿函数compare用户可以定义自己的排序方式，这在原有算法骨架的基础上大大提升了代码的可扩展性，使得算法的使用能够更加灵活。

3.3 策略模式

策略模式是指定义一系列算法并加以封装，在使用时这些算法可以根据实际情况替换的模式。STL在algorithm里大量使用了这一模式，同一算法为不同数据规模、不同迭代器模式提供了多种版本，并在最后进行了封装，使得其能够根据实际情况灵活调用内部的不同函数，提高了代码的效率。

3.4 迭代器模式

STL内部提供了多种迭代器，每一种迭代器可以适用于多种容器。根据容器性质的不同，STL又将迭代器分类，包括前进式迭代器、双向迭代器、随机访问迭代器等，这样在编写算法时可以将适用某种迭代器的所有容器归为一类，将迭代器抽象出来作为参数传入，减少代码的冗余。

4 总结

阅读STL尤其是算法部分的代码，第一感受是简洁易懂，条理清晰，大部分算法的流程都很容易理解；第二感受是注重效率，代码在很多地方做了优化，例如针对不同迭代器、针对是否有重载运算符等情况都提供了不同版本的代码，设计的比较完善。虽然只是粗枝大叶地阅读，但我觉得学到了很多东西，平时使用algorithm库里的算法很方便，但一直不知道是怎么实现的，这次阅读让我确实有所收获。希望之后能有时间继续把STL的其他部分代码再好好阅读一下。