第一章：STL概论

STL六大组件：容器，算法，迭代器，仿函数，配接器，配置器

第二章：空间配置器

空间配置器：为STL存放数值的各种容器配置空间的，空间包括但不限于内存

allocator有许多接口

自行实现一个简单的空间配置器：主要实现了分配内存并返回指针给对象的功能，用到了new，delete等运算符，还有对应对象的构造函数析构函数等。

次配置力的SGI空间配置器：SGI STL每一个容器都已经指定缺省的空间配置器为alloc，SGI也有一个符合标准的名为allocator的空间配置器，但是SGI不建议我们使用，因为效率不佳，只是把C++的new和delete做了一层包装。

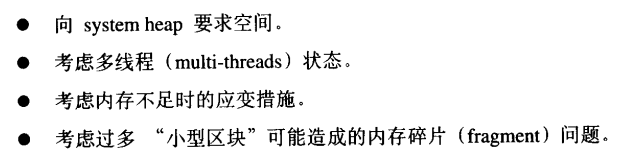
一般的内存分配：new和delete new先分配内存，再构造对象内容，delete先调用析构再释放内存

在STL空间配置器中：两阶段被分开，alloc::allocate()配置内存，alloc::deallocate()释放内存 对象构造由::construct()负责，析构由::destroy()负责

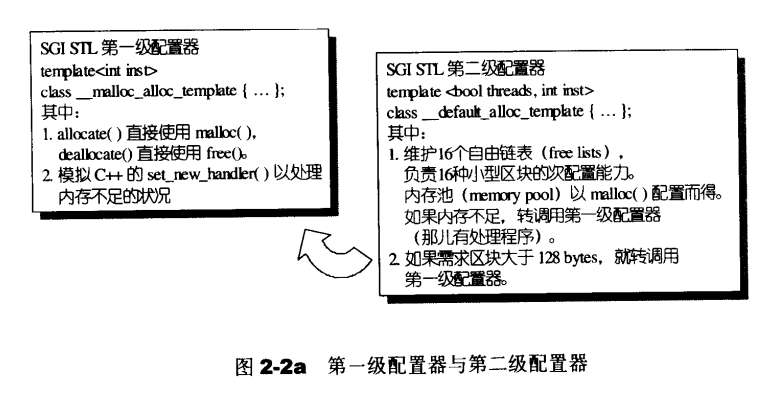
配置器定义在<memory>中：而SGI<memory>包含stl\_alloc.h和stl\_construct.h两个文件，stl\_alloc.h定义了一二级配置器，stl\_construct.h定义了全局函数construct() destroy()，负责对象的构造和析构。

stl\_construct.h中使用construct()接收一个指针p和一个初值value，这个函数用途是将初值设定到指针所指的空间上，new来构造对象，返回指针，destory()有直接析构的版本，接收指针，将指针所指之物析构掉，有接收迭代器的版本，将first和last范围的析构掉

stl\_alloc.h控制内存配置和释放，设计哲学

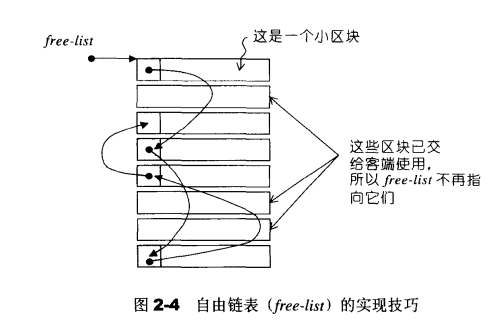


为了避免小型区块造成内存破碎，SGI使用了双层级配置器，第一层直接使用malloc和free，第二层则根据配置区间来决定，超过128字节使用第一级配置器，小于则使用内存池整理方式



C++ new handler机制是：你可以要求系统在内存配置需求无法满足时，调用一个你指定的函数。可以重复使用处理函数直到内存足够。

SGI STL二级配置器是每次配置一大块内存，并维护对应之自由链表。下次若再有相同的内存需求，就直接从free-lists中播出，如果客端释放归还小额区块，就由配置器会受到free-list中，并且SGI第二级配置器会主动将任何小额区块内存需求量上调至8的倍数并维护16个free-lists，为了节省内存，链表节点还用的是union，一处两用



空间配置函数allocate()，身为配置器的标准接口函数。首先判断区块大小，大于128byte调用第一级配置器，小于128byte检查对应的free list。如果freelist之内有可用的区块，就直接拿来用，如果没有可用区块，就将区块大小上调8倍数边界，然后调用refill()，准备为free list重新填充空间

空间释放函数deallocate()，大于128bytes就调用第一季配置器，小于128bytes就找出对应的freelist，将区块回收

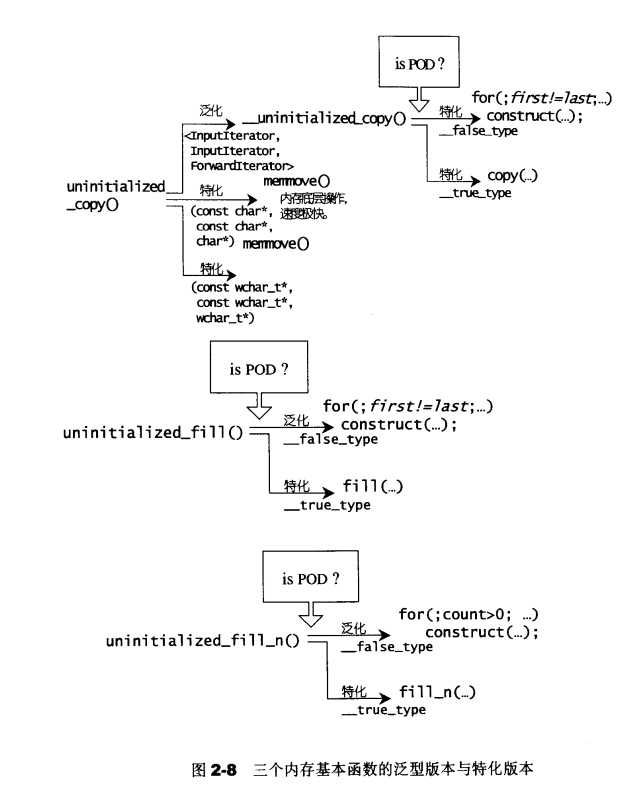
refill()函数：当freelist没有区块了，就调用refill()函数，为freelist重新填充空间。新的空间取自内存池。缺省取得20个新节点。

内存池：从memory pool取得空间给free list使用，是chunk\_alloc()的工作。先判断内存池的容量，够就直接给20个到freelist，不够二十但还有一个往上就给这么多，完全没了就从堆里再申请新的内存给内存池，如果整个系统堆内存都不够了，再寻找没使用完区块的另一个freelist，找到了就分一点给现在这个，找不到就要用第一级配置器了，它有out of memory机制，或许能分配成功，最终失败则发出badalloc异常。

内存基本处理工具：uninitialized\_copy() 函数把result范围内的每一个迭代器都指向未初始化区域，则uninitialized\_copy()会使用copy constructor给输入来源的范围内的每一个对象产生一份复制品，放进输出范围里。如果你需要实现一个容器，uninitialized\_copy()这样的函数可以配置内存区块，足已包含范围内的所有元素，使用uniniitalized\_copy()，在该内存区块上构造元素。

uninitialized\_fill函数会在范围上产生第三个参数x的复制品，范围内要么全部产生，要么全部别产生，如果有任何一个copy constructor出现异常，就必须析构掉其他所有的元素

uninitialized\_fill\_n接受三个参数，迭代器first指向初始化空间的起始位置，n表示欲初始化空间的大小，x表示初值，



第三章：迭代器概念与traits编程技法

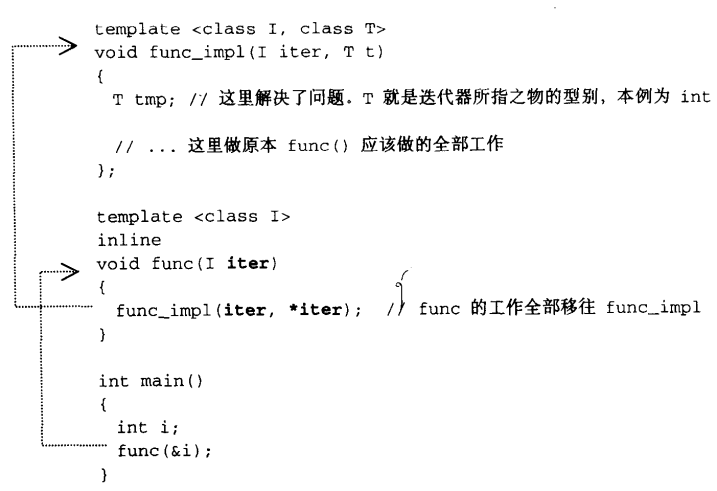
迭代器（iterator）模式：提供一种方法，使之能够依序巡防某个聚合物（容器）所含的各个元素，而又无需报炉该聚合物的内部表述方式

容器的泛化：类模板 class template

算法的泛化：函数模板 function template

迭代器扮演容器和算法中间的粘胶角色，迭代器是一种智能指针

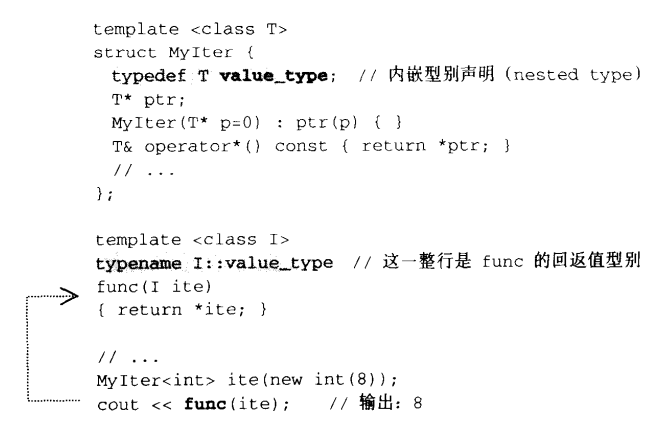
怎么得到迭代器的类别？：利用函数模板的从参数推导机制例如：



而利用此机制可以得到迭代器指向的所属的元素的类别，

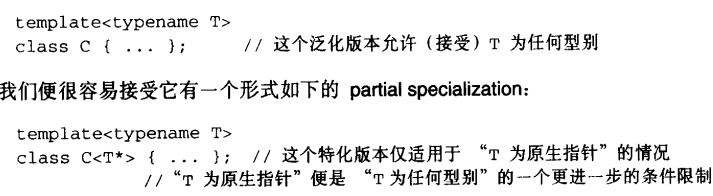
Traits编程技法：

迭代器所指对象的类型：该迭代器的value type。万一value type要作为函数的传回值，可以用声明内嵌类型来解决：



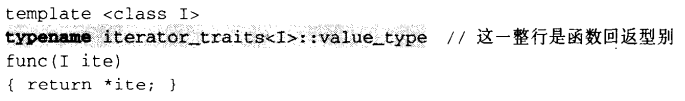
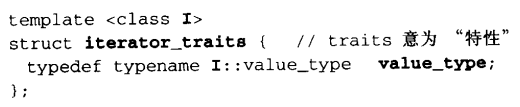
typename不能忘，因为T是一个template参数，未具现化之前，编译器对T一无所知，，通过typename告诉编译器这是一个类型，才能顺利通过编译

特化：任何模板template参数更进一步的条件限制所设计出来的一个特化版本，如下图：



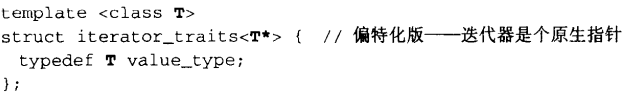
那如果迭代器的类型是一个特化的，要求迭代器为指针，就无法为它们定义内嵌型别。

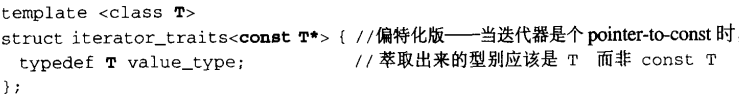
针对迭代器的特化版本，用traits



假设I有value\_type，多了一层间接性，好处是迭代器可以拥有特化的版本了，迭代器是一个原生指针，不是一种class类型，也可以通过在这种方式得到类型了。

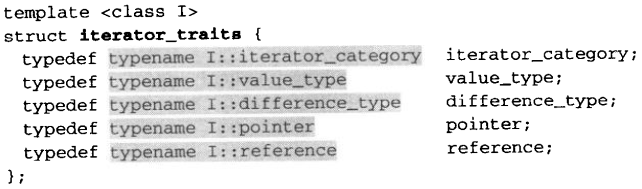
不管迭代器还是原生指针，int\*或者const int \*都可以拥有特化的版本了。





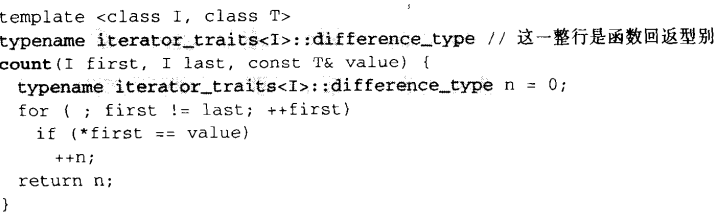
每一个迭代器必须必须遵守约定，自行以内嵌型别定义的方式定义出相应型别，否则就不能兼容于STL这个大家庭

最常用的五种迭代器型别：

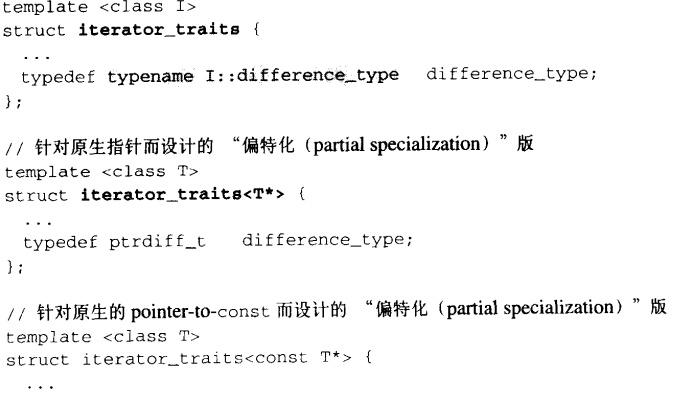


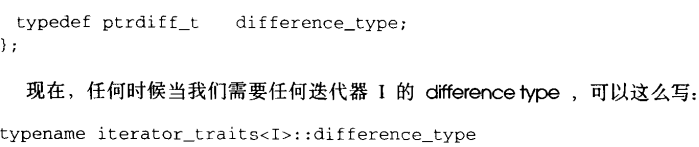
value type：迭代器所指对象的类型，任何一个与STL算法有搭配的类，都应该定义自己的value type内嵌型别

difference type：用来表示两个迭代器之间的距离，可以用来表示一个容器的最大容量，例如为一个泛型算法提供计数功能。



特化版本，针对指针和const的指针，以C++内建的ptrdiff\_t作为原生指针的difference type

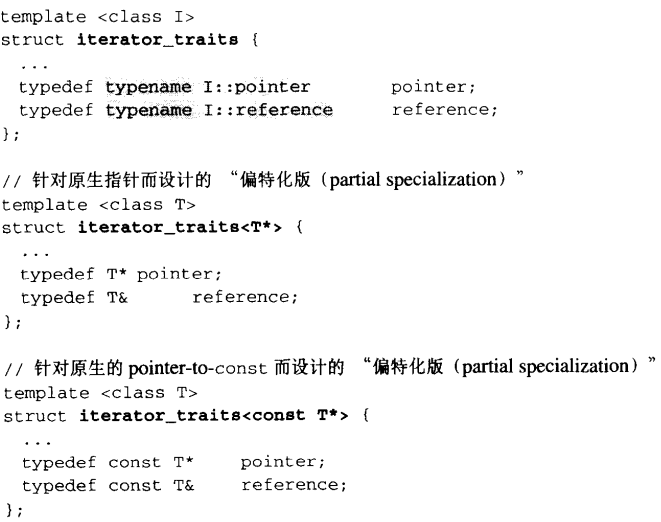




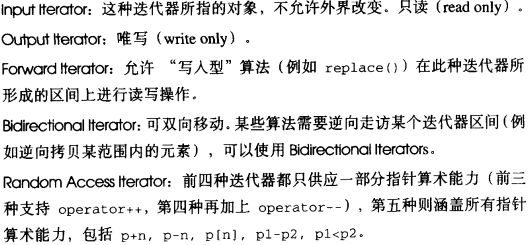
reference type：

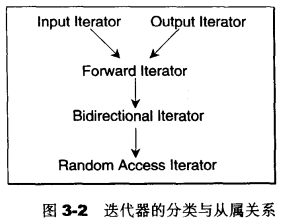
pointer type：返回指针，返回引用



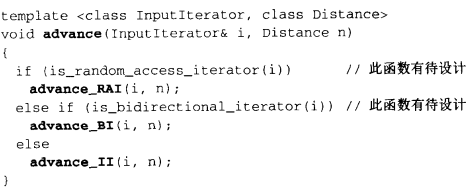


Iterator\_category：首先迭代器分成五类



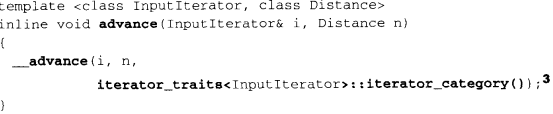


为了效率，应该为合适的算法选择合适的迭代器类型，比如一个算法接受三种迭代器，那可以写成下面这样：



执行期可以确定选择一个最有效率的函数，而函数重载机制可以让程序在编译期就选择正确的版本

例如一个advance函数，实现它的接口：

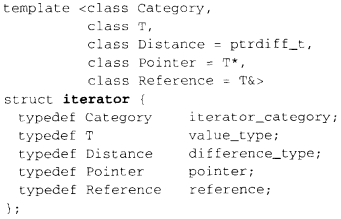


而\_advance就可以用重载来实现，根据最后一个参数，编译器来决定调用哪一个\_advance函数

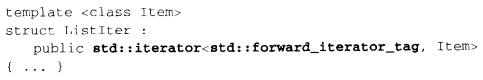
所以迭代器要增加一个类型 iterator\_category



任何一种迭代器都要有这五个内嵌的参数，否则便无法被STL框架承认，STL提供了一个iterators如下：

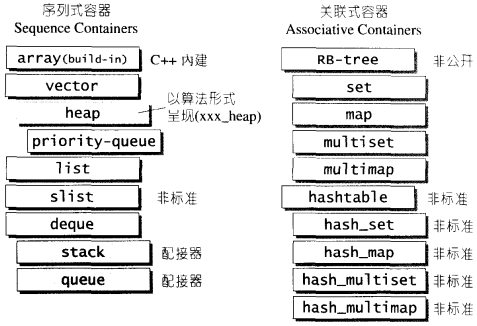


后三个参数都有默认值，自行定义迭代器只需要提供前两个参数即可



第四章：序列式容器

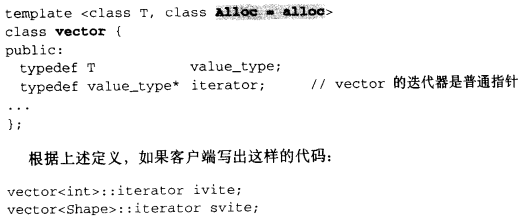
SGI STL的各种容器：



上下包含的关系heap包含vector，priority-queue包含有heap

C++语言本身提供了一个array，STL提供了其他序列式容器，其中stack和queue是把deque改头换面，技术上被归类为adapter

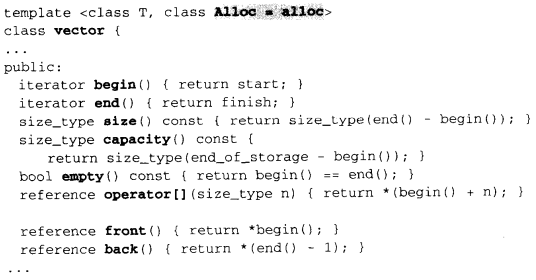
vector：迭代器是指针，因为指针满足所有的vector迭代器的必要条件



指针支持随机存取，所以，vector提供的是随机存取迭代器

vector所采用的数据结构为线性连续空间，两个迭代器start和finish分别指向配置得来的联系空间中已使用的范围，迭代器end\_of\_storage指向整块连续空间的尾端

vector实际配置的大小比客户端需求量更大，以备将来可能的扩充，这就是容量的概念，一个vector的容量永远大于或等于其大小。



缺省使用alloc作为空间配置器，构造函数中使用了空间配置器，为分配的空间赋值，有uninitialized\_fill\_n这样的空间配置器函数的参与。并且会根据第一参数的类型，决定使用算法fill\_n()还是construct()

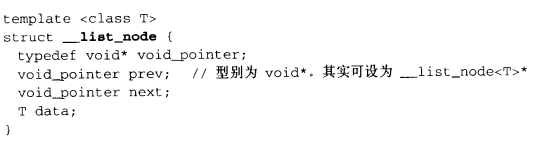
尾端插入：检查是否还有备用空间，有就直接在备用空间上构造，并调整finish，使vector变大，没有就扩充空间

Vector的动态增加大小：并不是在原空间之后接续新空间，而是以原大小的两倍另外配置一块较大的空间，然后将原内容拷贝过来。，然后在原内容之后构造新元素，释放原空间，对于vector的任何操作，一旦引起空间重新配置，原来的迭代器就失效了。分配内存回收内存啥的也都用的是空间配置器。

vector的插入删除等操作也和空间配置器息息相关，在某个位置插入时如果当前的空间不够还要找新的空间

list：每个元素插入或者删除，就配置或者释放一个元素空间，任何元素的插入或者元素移除，list永远是常数时间

list的节点：



list的插入和合并不会使原来的迭代器失效，这在vector是不成立的，因为vector插入可能会导致全部换位置

list数据结构：list是一个环状的双向链表，只要一个指针，就可以表示整个环状双向链表，这个指针指向刻意加载链表尾端的空白节点上

list构造：一个空list含有一个前后指针都指向自己的空节点

list的sort：不能用stl的sort算法，因为其用的是随机读取迭代器，可以用list自身的sort成员函数。用的是快排

deque：双向开口的连续线性空间。

deque和vector的区别：deque允许常数时间对头端进行元素的插入和移除。二是deque没有容量的概念，它是动态地以分段连续空间组合而成，随时可以增加一段新的空间并链接起来。

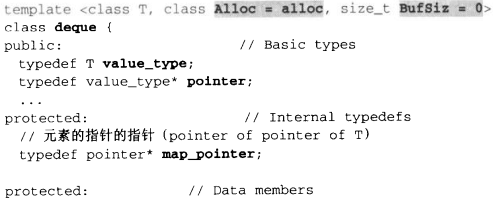
deque的迭代器：比vector要复杂很多，vector就是一个指针。应该尽可能的选择用vector，效率更高

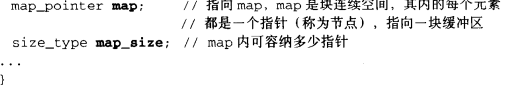
deque的排序：为了最高效率，可将deque先完整地复制到vector，然后排序后，在复制回来

vector的成长：找新空间，复制，释放原空间。

deque：一段段定量连续空间，一旦要在deque的前端或尾端增加新空间，变配置一段定量连续空间，串接在整个deque的头端或尾端。deque的最大任务就是在这些分段的定量连续空间上，维护其整体连续的假象，并提供随机存取的接口，避开“重复配置、复制、释放”的轮回，代价是复杂的迭代器架构。

deque的主控：一块map（不是STL的那个map），是一小块连续空间，类似虚表。其中每个元素都是指针，指向另一段较大的连续线性空间，称为缓冲区。缓冲区才是deque的储存空间。SGI STL允许指定缓冲区，默认值0表示将使用512字节缓冲区。



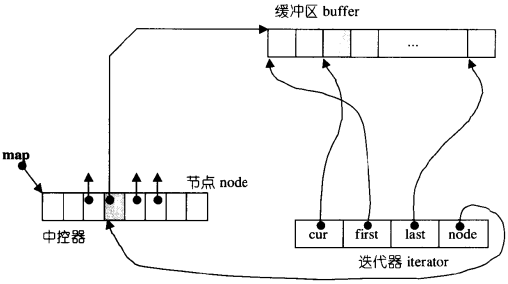


map是一个T\*\*，它指向缓冲区的指针，缓冲区的指针又指向T类型的一块空间

当map的使用率已经满载，便需要再找一块更大的空间来作为map

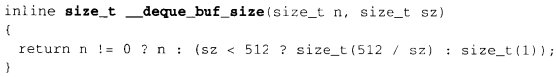
维持deque的整体连续假象：迭代器的++和—两个运算符。

deque的迭代器，中控器，缓冲区的相互关系：

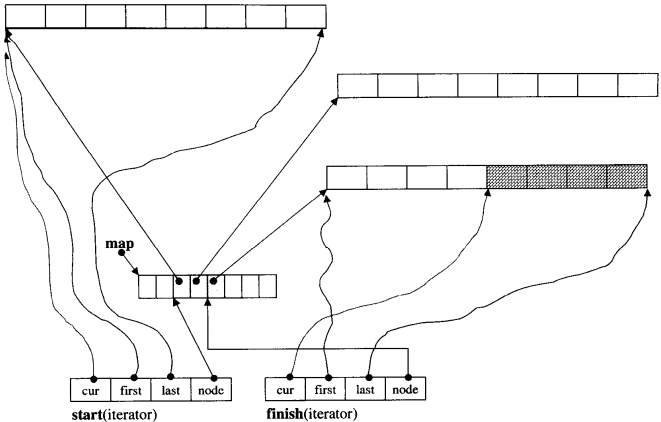


缓冲区存放元素个数



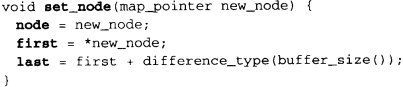


如一个缓冲区32字节，可存储8个int值，那二十个int存储进这个deque，需要三个缓冲区，end()和begin()传回的迭代器如下图：



map有8个节点，目前用了三个节点

在缓冲区的边缘时，如果需要，要进行缓冲区跳转，使用set\_node



deque的数据结构：维护指向map的指针，指向第一个缓冲区第一个元素的start迭代器，指向最后一个缓冲区最后一个元素的finish迭代器，还要记住map大小，map提供的节点不足，必须重新配置一块更大的map

deque的构造和内存管理：deque定义了两个专属空间配置器。关键在于前后插入时如果没有位置了，申请新的缓冲区，会用到set\_node

map满了之后：分前面和后面，在map表中，最先被使用的是中间的位置过程也是空间配置器分配新空间，复制，回收旧空间。

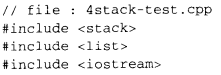
插入删除等操作也是在缓冲区上进行的一些删除复制移动的操作。

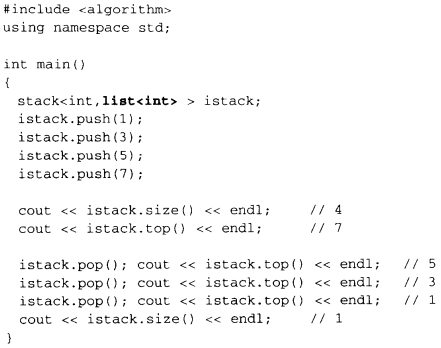
stack：SGI STL中以deque为缺省状况下的stack底部结构。封闭stack的头端开口，便形成一个stack。

adapter：像stack这样把deque修改接口，形成另一种形态的，被称为容器adapter而不被单纯归类为容器。

stack没有迭代器：不提供走访功能，只有顶端的元素才有机会被取用。

list作为底层容器的stack：list也是前后开口的数据结构，如下：



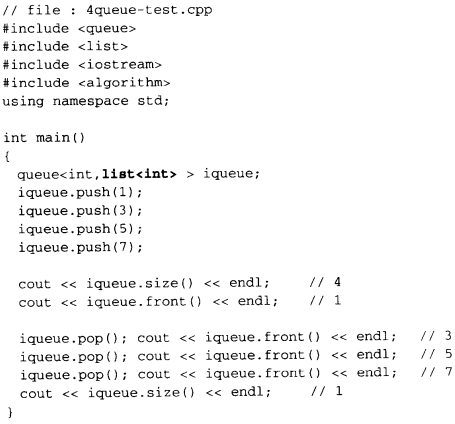


queue：fifo数据结构，除了pop，和push可以接触到前后元素，没有其他办法可以得到其他元素，不允许遍历。

adapter：同样归属于容器adapter而非单纯的容器，底层结构用deque，因为deque是两端的访问，因此用deque来实现queue很方便

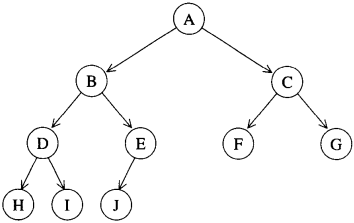
queue同样没有迭代器

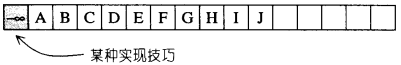
list也是双向开口的数据结构，通过list实现queue如下：



heap：不属于STL容器组件，是priority queue的助手。优先级队列允许用户将任何元素放入容器中，取出时一定是从优先级最高的元素开始取，最大堆具有这样的特性，适合优先级队列的低层机制。

完全二叉树：完全的，除了最后一层其他都填满了，通过数组实现的完全二叉树如下：





一个节点的左子节点位于array的2i处，右子节点位于array的2i+1处，父节点位于i/2处，array轻易实现完全二叉树

因此，堆的实现需要一个vector和一堆heap算法

最大堆和最小堆：大 每个节点的键值都大于或等于其子节点键值，小 每个节点的键值都小于或等于其子节点键值

STL提供的是最大堆（以下以最大堆为例）

插入新节点的上溯程序：将新节点拿来与父节点进行比较，如果键值大于父节点，就交换双方的值。如此一直上溯，直到不需对换或直到根节点为止。

取走最大值的下溯程序：把最大值读出，然后把vector中第一个节点（最大值）和最后一个节点（最小值）交换，然后进行下溯，与子节点进行比较，并与较大的子节点进行交换，如此一直下放，直到它的键值大于左右两个子节点。

heap\_sort堆排序：每次都pop\_head，把最大的放到vector最后面去，等到进行多次，就得到一个从小到大的序列。

make\_heap算法：从第一个父节点（第i/2个节点，使用了完全二叉树的性质）开始，与子节点进行比较，和比它大的子节点中最大的交换位置，交换后，原来的子节点执行上溯，父节点执行下溯

heap同样没有迭代器

priority\_queue：带有权值概念的队列，其内的元素并非按照被推入的次序排列，而是自动依照元素的权值排列，缺省情况下优先级队列用一个最大堆完成，后者是一个向量表现得完全二叉树，最大堆可以满足优先级队列需要的按权值高低自动递增排序的特性，总是推出最大的元素

同样被看作container adapter，而不被认为是container

同样没有迭代器

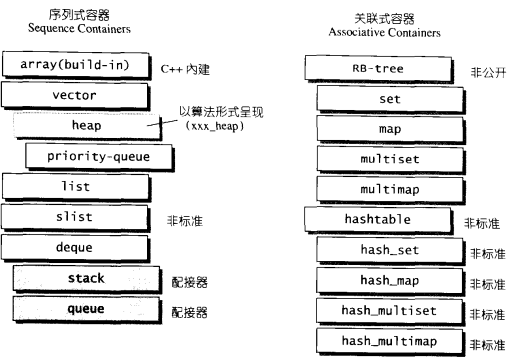
slist：STL list是一个双向链表，STL还提供了一个单向链表，名为slist。

与list的主要差别：slist属于单向的forward iterator，而list的迭代器是双向的bidirectional iterator。slist的功能就受到了许多限制，但是单向链表所耗用的空间更小，某些操作更快。

与list的共同特点：插入，移除，接合等操作并不会造成原有的迭代器失效。

slist的缺点：无法插入新元素到指定元素之前，

第五章：关联式容器 associative container

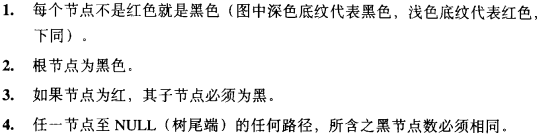


关联式容器：每个数据都有一个键值（key）和一个实值（value），容器底层结构（可能是红黑树或者哈希表）将以某种特定的规则依照键值大小将元素放入合适的位置，关联式容器没有所谓头尾，也不会有push，pop，begin，end这样的操作

关联式容器内部结构：平衡二叉树，如AVL-tree，RB-tree，用的最多的是红黑树。

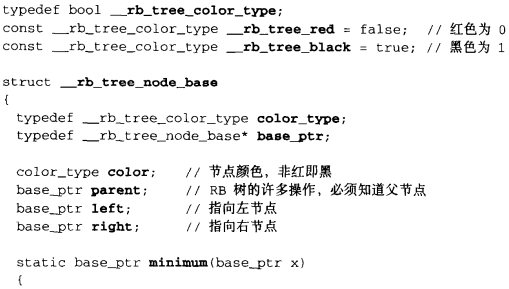
AVL树：主要在于需要平衡，左左右右左右右左，分别需要进行单旋转和双旋转

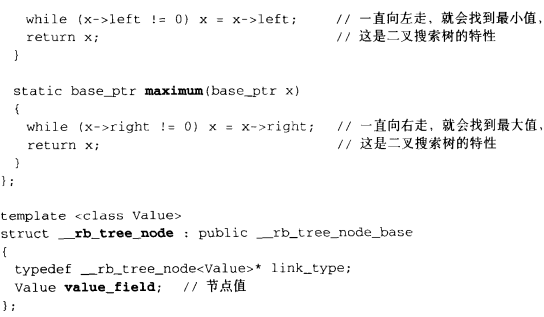
RB-tree：满足以下条件的二叉搜索树



由上到下的修正程序：假设新增节点为A，只要看到某节点X的两个子节点都为红色，就把X改为红色，并把两个子节点改为黑色。如果X的父节点是红色，那就要进行旋转，改变颜色，之后再插入A。

RB-tree的节点设计：下面是SGI STL的实现源代码





迭代器：++ -- 等操作，是在树节点之间进行交换

find成员函数：查找特定值的函数吗，时间复杂度是对数级别的，因为是BST，所以其实挺好写的

set：特性是所有元素会根据元素的键值进行排序。而且set的键值就是实值，set不允许两个元素有同样的键值。

set的迭代器：不能通过set的迭代器改变set中元素的值，set<T>::iterator被定义为底层RB-tree的const\_iterator

set增加或删除元素对原有的迭代器没有影响

set的那些操作RB-tree基本上都实现了，因此set的实现也就是调用rbtree的接口

map：特性是所有元素都会根据元素的键值自动被排序，map的所有元素都是pair，同时拥有实值和键值，pair第一元素视为键值，第二元素视为实值，map不允许两个元素拥有同一个键值。

map的迭代器：不能通过迭代器改变map的元素的键值但可以改变实值，新增和删除操作对之前的迭代器都不会有影响

map的操作RB-tree基本都实现了，因此map的实现也是用rebtree的接口

multiset：允许键值重复，因此底层插入操作使用的是insert\_equal而不是insert\_unique()

multimap:允许键值重复，也用的是insert\_equal

rbtree：广泛运用的平衡二叉搜索树，RB-tree（红黑树），不仅在树形上的平衡表现的不错，在效率和时间复杂度上也保持着相当的平衡，对数平均时间的复杂度，前提是输入数据有足够的随机性

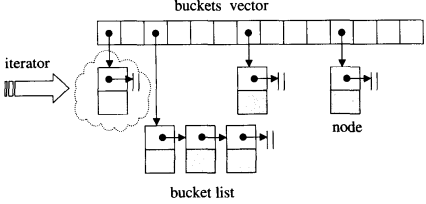
hashtable：哈希表，在插入删除搜寻上有常数时间性能，而且不需要依赖数据的随机性

解决碰撞：

线性探测，负载系数在0~1之间，平均情况是一半长度

二次探测：线性探测是发现了碰撞加1，二次探测是发现了碰撞加1平方，2平方，3平方这样

开链：buckets vector 和bucket list这样的处理冲突



Bucket使用的list并不采用STL的list或者slist，而是自行维护的list

如果目前节点是list的尾端，就跳到下一个bucket上

hashset：底层是hashtable，运用set目的是为了快速搜寻元素，rbtree和hashtable都可以，反应出来的结果就是，set的元素有自动排序而hashset没有

hashmap：底层是hashtable，有实值和键值，同样与rbtree为底层的map相比没有自动排序。

hash\_multiset的特性与multiset完全相同，，底层用hashtable实现，插入用insert\_equal,而不用insert\_unique

hash\_multimap与multimap特性完全相同，底层用hashtable实现，插入用insert\_equal而不用insert\_unique

第六章：算法

算法复杂度：大O表示法

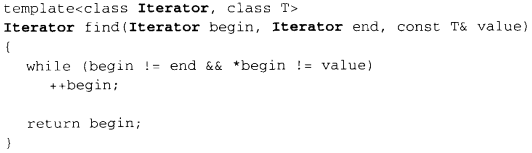
STL的算法：涉及元素的查找，复制，特定元素计数，相等判断，填充，更改容器内的元素，替换元素，判断属性（是否是堆，是否有序），求特征值（max，min等），集合运算，堆操作

非质变算法：不改变操作对象的值

STL算法的一般形式：所有泛型算法的前两个参数都是一对迭代器，通常为first和last，stl的算法的声明是一个算法的迭代器的最低要求，例如，find（）要求一个inputiterator，这是他的最低要求，给更高类型的迭代器也可以。

质变算法通常提供两个版本，一个就地进行版，另一个是copy版，将对象的内容复制一份副本，然后在副本上进行修改并返回副本。

算法的泛化：迭代器+函数模板，进行抽象化，不用暴露太多算法内部的细节，并且可以适用于不同的容器



数值算法：

求总和：accumulate，双版本，一个求和，一个二元操作并求和

求相邻元素差值：adjecent\_difference

求内积的和：inner\_product

部分和：partial\_sum

乘幂：power，底层用位操作实现控制

基本算法：

set的相关算法：并集，交集，差集，对称差集

数据处理算法（用到了补充记录，因为太多。）

第七章：函数对象（仿函数）

要让某种操作作为算法的参数，唯一办法就是将该操作设计为一个函数，再将函数指针当做算法的一个参数，或者是将该操作设计为一个所谓的仿函数（语言层面是一个类），再将该仿函数产生一个对象，并以此对象作为算法的一个参数。

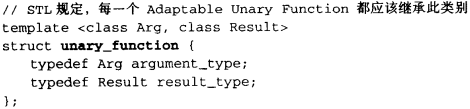
比如sort算法中总要自定义一个compare函数传递函数指针给sort算法，这个compare函数就是一个函数对象。

扮演“策略”角色：让STL算法更加灵活，关键在于STL仿函数的可配接性

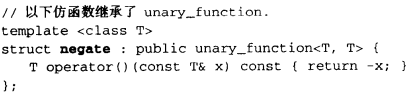
就像是迭代器要定义五个相应类型一样，仿函数也要定义自己的相应类型，这是为了能够被配接器（adapter）取出，就像迭代器被用于算法时能得到相应的类型。

仿函数的相应类型主要表现函数参数类型和传回值类型，stl\_function.h中定义了两个classes，分别代表一元仿函数和二元仿函数。任何仿函数，只要按照需求选择继承一个，便自动拥有了相应类型，就自动拥有了配接能力。

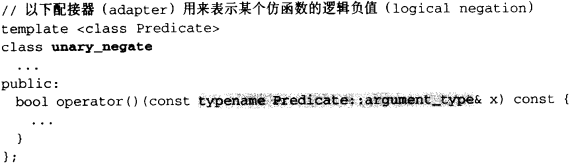
unary\_function定义一元函数的参数类型和返回值类型



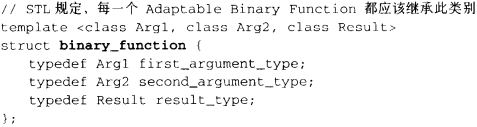
以下仿函数继承了unary\_function,



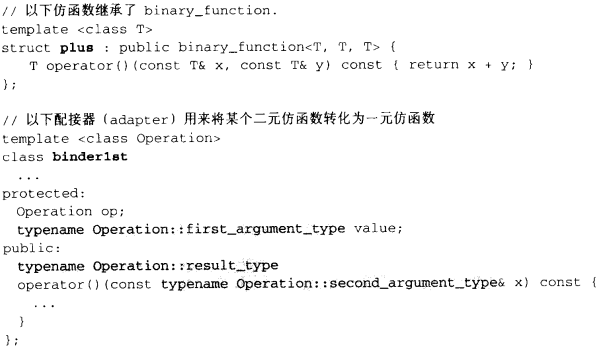
配接器可以通过以下灰字部分的方式获得仿函数的类型



binary\_function用来呈现二元函数的第一参数类型，第二参数类型，以及返回值类型



仿函数继承了binary\_function，用户便可以取得仿函数的各种相应类型



算术类仿函数：加减乘除模取，否定，都是二元运算。仿函数主要是搭配STL算法，比如对一个向量中的每一个元素进行乘法运算



关系运算类仿函数：六种运算，每一个都是二元运算。等于，不等于，大于，大于等于，小于，小于等于六种。

逻辑运算类仿函数：支持逻辑运算中的与或非三种运算，非是一元运算

证同、选择、投射仿函数：将参数原封不懂得传回，不在泛型程序或STL中使用原本极其简单的identity project select等操作，是为了增加一层间接性，从而实现抽象化。

第八章：配接器adapter

设计模式中对adapter的定义：将一个类的接口转换为另一个类的接口，使原本接口不兼容而不能合作的类可以一起运作。

STL中提供的adapter：改变仿函数接口的叫做function adapter 改变容器接口的，叫做container adapter，改变迭代器接口的，成为iterator adapter

container adapter：提供两个容器 queue和stack，

iterator adapter：STL提供了许多应用于迭代器身上的配接器，包括insert iterator，reverse iterator，iostream iterator。insert iterators就是将一般的迭代器赋值操作转变为插入操作，这样的迭代器包括尾部插入的尾插迭代器，头部插入的头部迭代器，任意位置插入的插入迭代器。reverse iterators可以将一般迭代器的行进方向逆转，使迭代器的++操作变--，对于从尾端开始的算法上，有很大的方便性。IOStream iterator，可以将迭代器绑定到某个iostream对象上，绑定到istream对象称为istream\_iterator拥有输入功能，绑定到ostream对象上，称为ostream\_iterator，拥有输出功能，这种迭代器运用于屏幕输出，非常方便。可以以它为蓝图，迭代器可以绑定到任何输出和输入装置上。Function adapters总是以仿函数作为参数，给人“拿某个配接器来修饰某个仿函数”的直观感受，这里介绍的迭代器配置器很少以迭代器为直接参数，所谓迭代器的修饰只是一种直观上的感受，你可以千变万化的写出适合自己用的任何迭代器。

functor adapters：价值在于通过它们的绑定、组合、修饰能力，几乎可以无限制的创造出各种可能的表达式，例如有仿函数less<int>，就可以通过适配器组成不小于12元素的个数这样的表达式 not1(bind2nd(less<int>(),12))。就成为了不小于12的语意，整个成为一个表达式，又比如有适配器compose，可以把不同的仿函数复合到一起compose1(f(x),g(y)),，如果一个仿函数希望自己能够被配接，那么它本身应该可配接，比如一元仿函数要继承自unary\_function

多态中一般虚成员函数的指针也可以被配接器修饰后经由STL中的算法进行调用，可以在STL中体现多态功能。