**STL学习笔记**

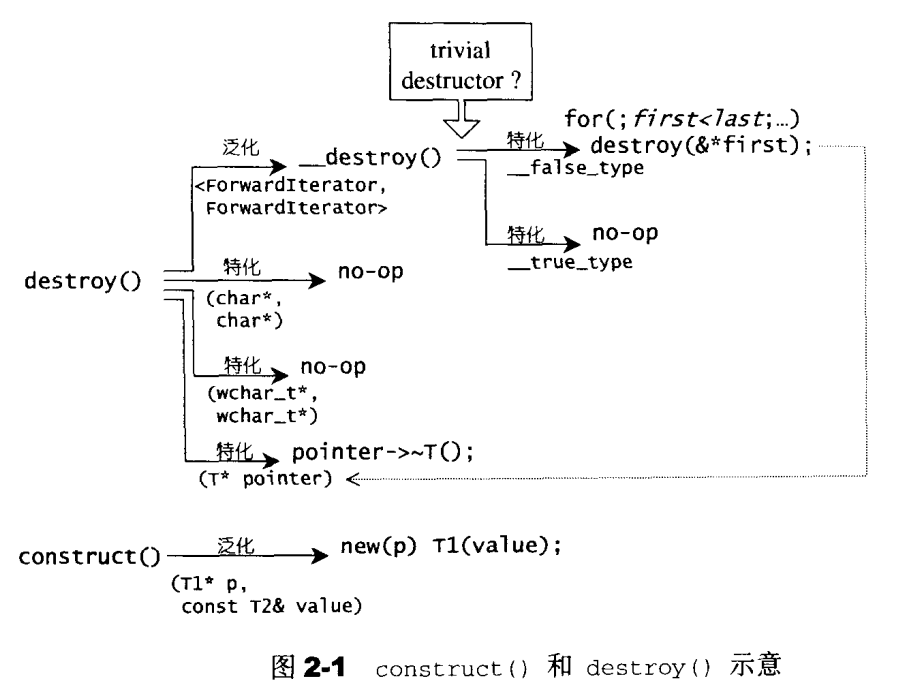
# 第二章 配置器allocator

配置器负责空间配置与管理，从实现的角度来看，配置器是一个实现了动态空间配置、管理、空间的class template。整个STL的操作对象(所有的数值)都存放在容器之内，而容器一定需要配置空间以存放内容。

## 2.1. 具有次配置力(sub-allocation)的SGI空间配置器

构造和析构的基本工具：construct()和destroy()

1. construct(): 接受一个指针和一个初值value，将初值设定到指针所指的空间上(placement new).
2. destroy(): 如果对象析构函数是trivial，则不必有任何操作；否则遍历整个范围并逐个调用析构函数。

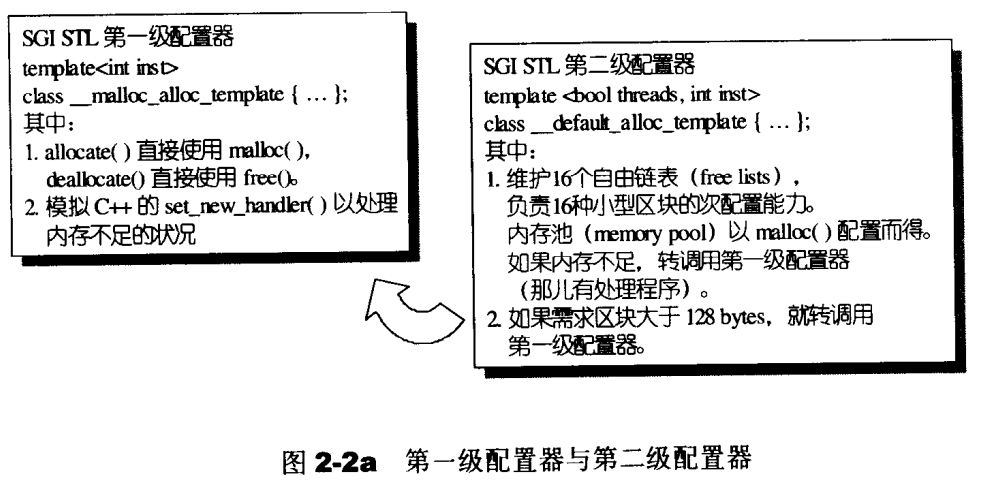


考虑到小型区块可能造成内存碎片问题，SGI采用两级配置器：

第一级配置器以malloc()，free()，realloc()等C函数执行实际的内存配置、释放、重配置操作，并实现出类似C++ new-handler机制。

第二级配置器的原理：

1. 当配置区块超过 128 bytes，就使用第一级配置器
2. 当配置区块小于 128 bytes，使用内存池管理

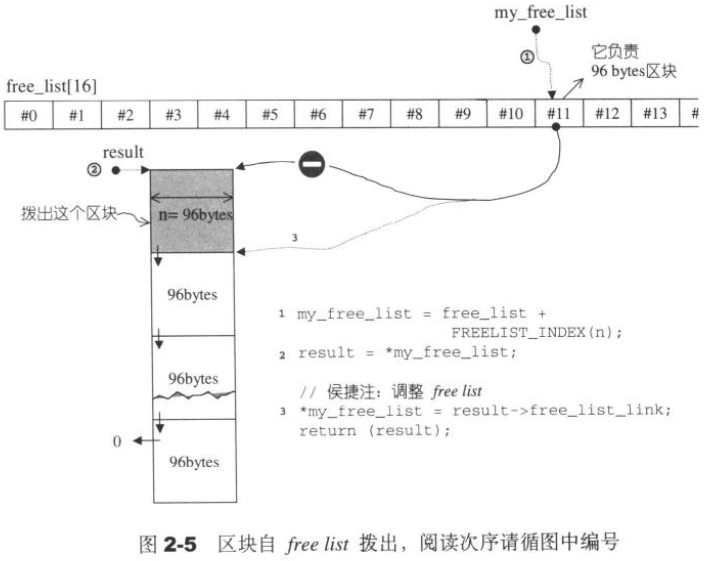


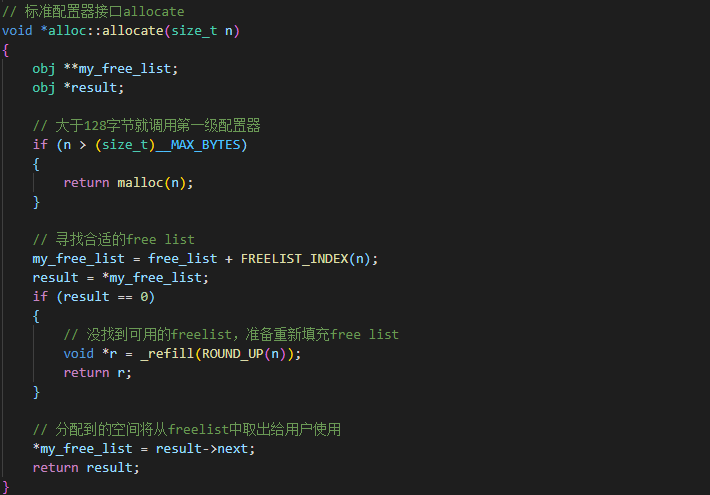


其中free-list是指针数组，16个数组元素，就是16个free-list，各自管理大小分别为8，16，24，32，...128 bytes(8的倍数)的小额区块。小额区块的结构体 union \_Obj 使用链表连接起来。配置器负责配置，同时也负责回收。

### 2.1.1. 空间配置函数allocate()

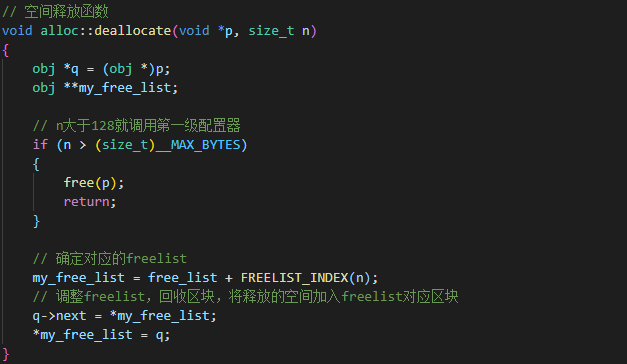
函数首先判断区块大小，大于128字节就调用第一级配置器，小于128bytes就检查对应的free list。如果free list内有可用的区块，就直接拿来用，如果没有可用区块，就将区块上调至8倍数边界，然后调用refill()，准备为free list重新填充空间。

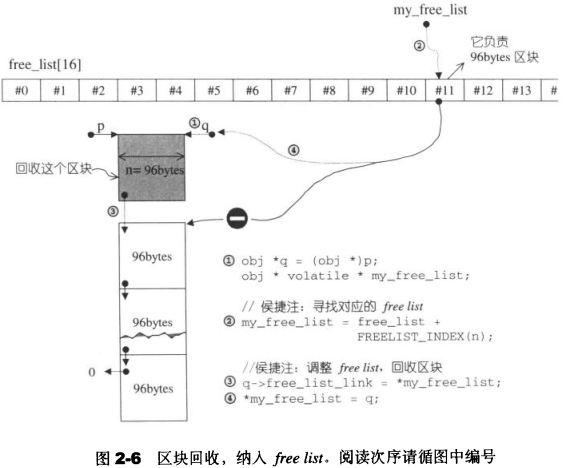




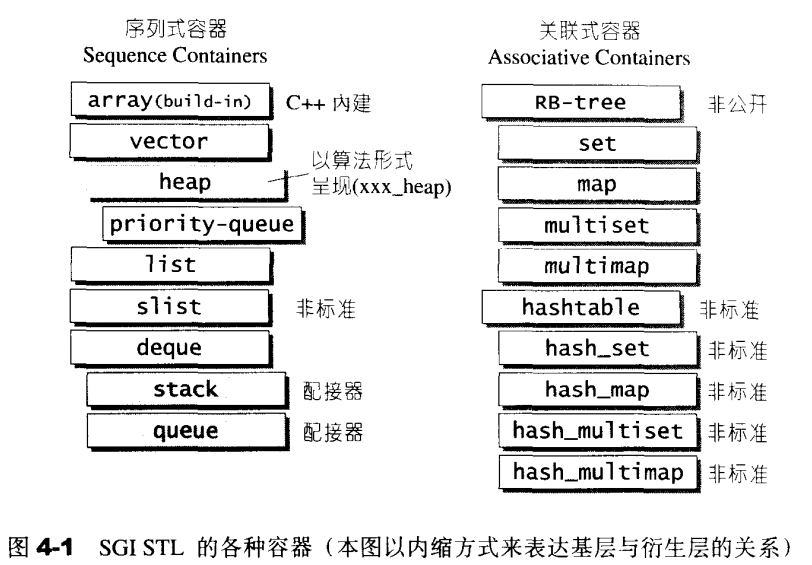
### 2.1.2. 空间释放函数deallocate()

该函数首先判断区块大小，大于128bytes就调用第一级配置器，否则就找出对应free list，将区块回收。





# 第四章 序列式容器



## 4.1. array

array是固定大小的数组，声明时必须指定元素类型和数量，此后大小固定，不会增减。如：array<int, 5> arr;

由于array大小固定，故其max\_size()、size()方法的返回值都是元素个数。没有push\_back()、push\_front()、push()、pop()等方法。

data()方法返回数组首地址。

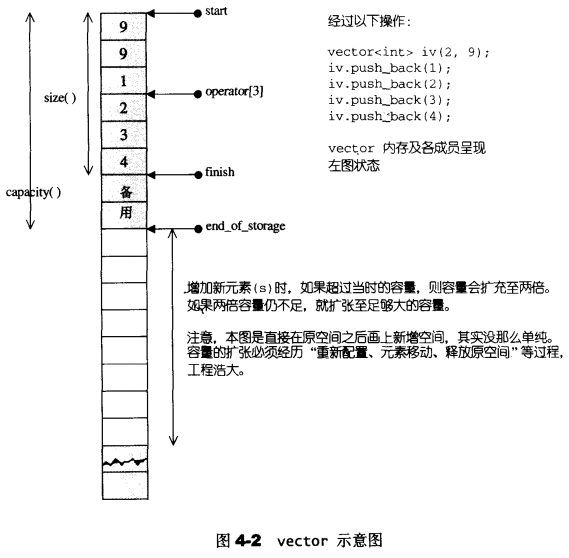
## 4.2. vector

vector是可变长度的数组，**当剩余空间不够时，size()增大为原来的两倍；而且，vector并不是在原地扩充size，由于vector分配的是连续空间，所以并不能保证在原vector后有足够空间扩充**。

heap的底层实现是vector，priority\_queue底层实现是heap。







### vector上的常见操作复杂度

* 随机访问——常数O(1)
* 在末尾插入或移除元素——均摊常数O(1)
* 插入或移除元素到vector结尾的距离成线性O(n)

### vector的迭代器类型Random Access Iterator

vector的迭代器，其实是所容纳元素类型的指针。vector的迭代器涵盖了指针所有的算术能力(operator\*，operator->，operator++，operator--，operator+，operator-，operator+=，operator-=)， 同时vector支持随机存取，所以vector提供是Random Access Iterator。

### copy功能

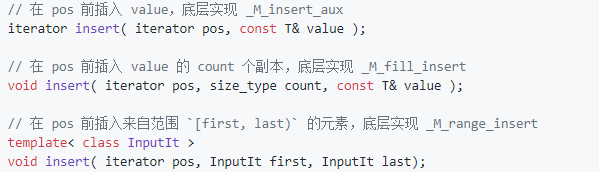


函数的功能：复制 [first, last)所定义的范围中的元素到始于d\_first的另一范围。

函数的参数：

* first, last - 要复制的元素范围
* d\_first - 目标范围的起始

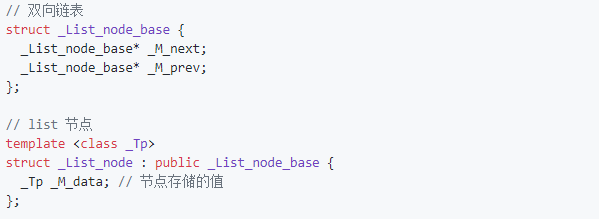
### vector insert形式



## 4.3. list

list是一个双向链表，同时还是一个环状的双向链表；对于任何位置的元素插入或元素移除，list永远是常数时间。

### 4.3.1. list节点

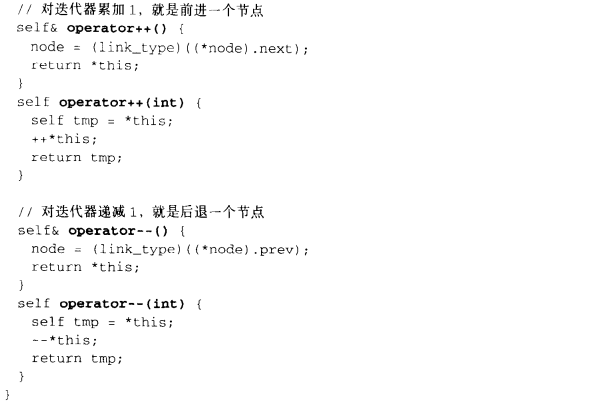


### 4.3.2. list的迭代器

list的节点可以在存储空间中不连续，所以list的迭代器必须具备前移、后移的能力，list 提供是Bidirectional iterator。

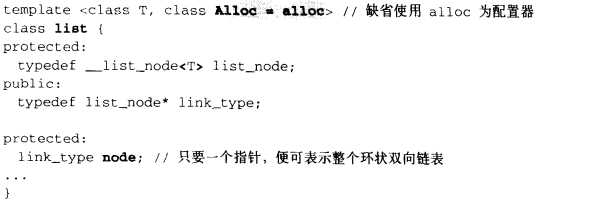
List有一个重要的性质：插入操作insert和接合操作splice都不会造成原有的迭代器失效。（这在vector是不成立的，因为vector的插入操作偶可能导致记忆体重新配置，导致原有的迭代器全部失效）。List的删除操作erase，也只有“指向被删除元素”的那个迭代器失效，其他迭代器不受影响。



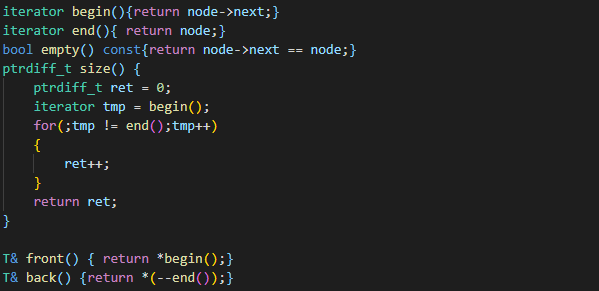


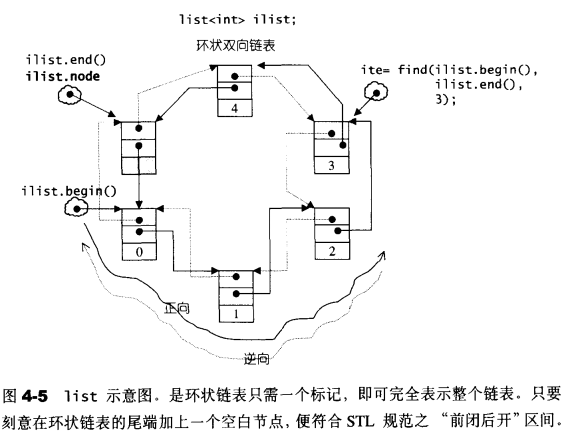
### 4.3.3. list的数据结构

Lsit是一个环状双向链表，所以只需要一个指针，便可以完整表现整个链表：



为附和STL对于前闭后开区间的要求，list内部有一个刻意置于尾端的一个空白节点node。这样一来，对于下列方法，都可以很方便完成：



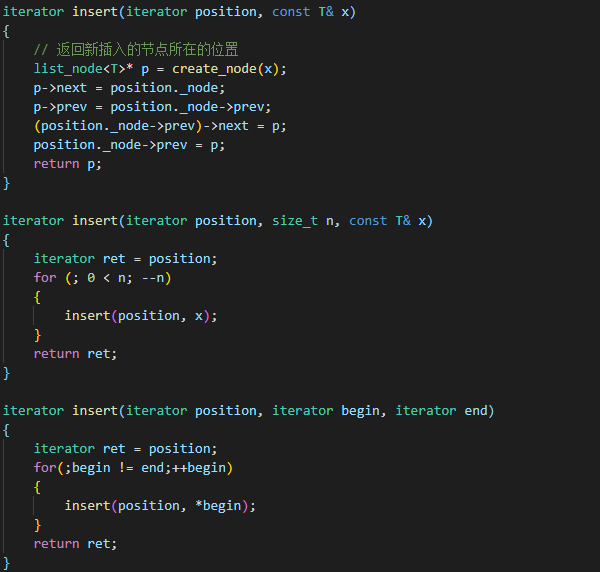


### 4.3.4. list的构造和内存管理

List默认使用alloc作为空间配置器。

#### insert()

insert()是一个重载函数，有多重形式，**均返回新插入节点所在的位置的迭代器**。



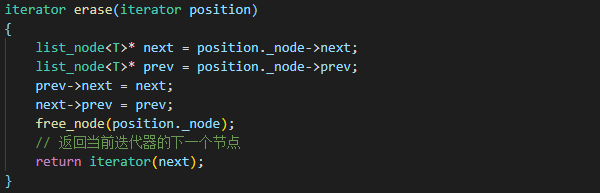
#### push\_back()和push\_front()

push \_back()将元素插入到list尾端，函数内部调用的是insert。push\_front()同理。



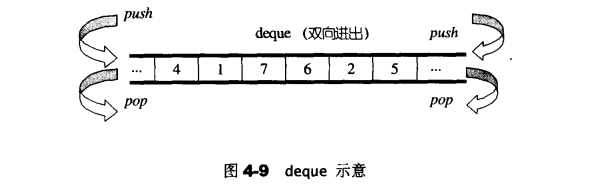
#### erase()

erase()移除迭代器所指节点，**返回当前迭代器的下一个节点**：



## 4.4. deque

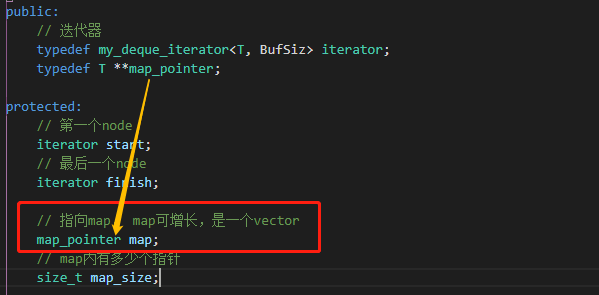
vector是单向开口的连续线性空间，deque则是一种双向开口的连续线性空间。deque可以在头尾两端分别做元素插入和删除操作。（vector也可以在头尾两端操作，但头部操作效率奇差）。deque允许在常数时间内对头部进行元素的插入和删除操作，没有容量capacity的概念，因为它是**动态地以分段连续空间组合而成，随时可以增加一段新的空间并链接起来**。

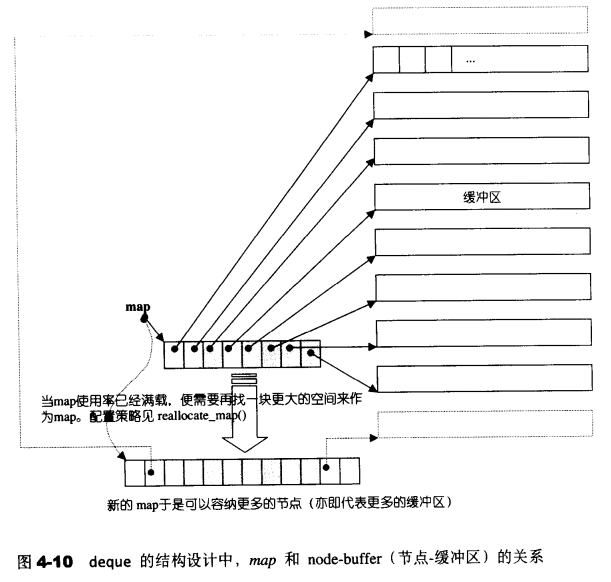


### 4.4.1. deque的中控器

deque由一段一段的定量连续空间构成，一旦有必要在deque的前端或尾端增加新的空间，便配置一段定量连续空间，串接在整个deque的头端或尾端。deque的任务之一，**是在这些分段的定量连续空间山个，维护其整体连续的假象，并提供随机存取的接口**。

deque以一个所谓的map（实质上是**元素类型的二级指针**）作为主控。map指向一小块连续空间，其中的每个元素是一个指针，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的存储空间主体。我们可以指定缓冲区的大小，默认值0表示将使用512字节的缓冲区。



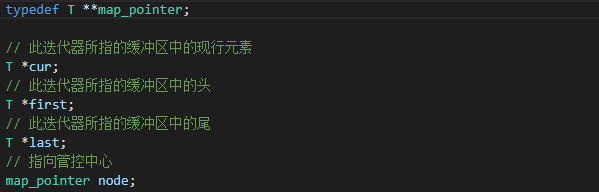


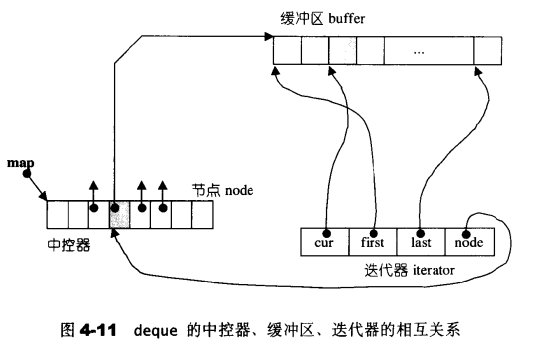
### 4.4.2. deque的迭代器

deque是分段连续空间，维护其整体连续假象的任务，由迭代器的运算子operator++和operator--完成。

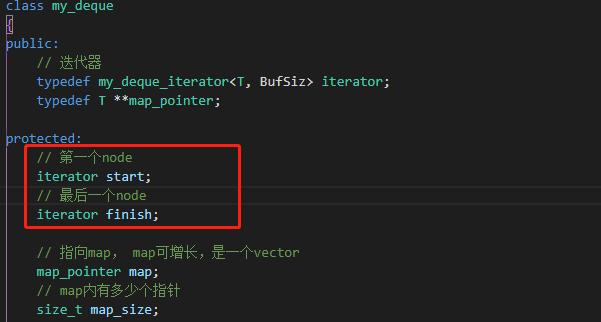
deque迭代器要能够指出分段连续空间（即缓冲区）在哪里（在map的什么位置），其次它必须能够判断自己是否已经处于其所在缓冲区的边缘，一旦前进或后退时就必须跳跃至下一个或者上一个缓冲区。为了能够正确跳跃，deque必须随时掌握管控中心。

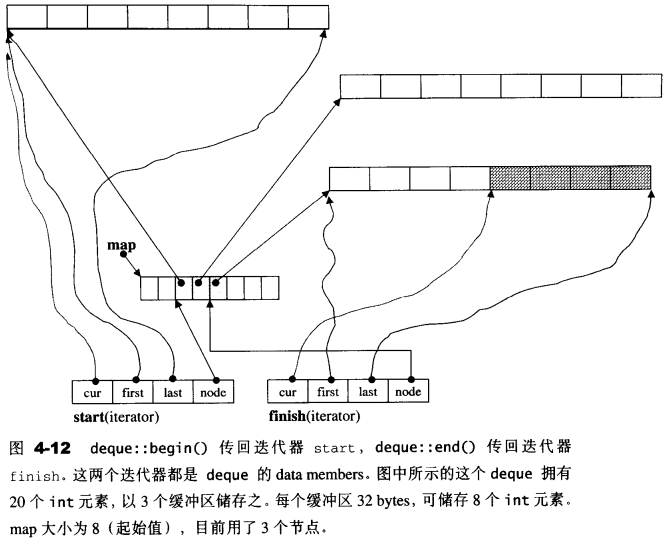
deque迭代器的元素及元素与管控中心、缓冲区的关系如下所示：





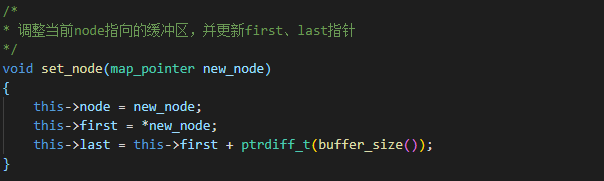
deque的数据结构里设计了两个迭代器start和finish，其中start指向第一个已使用的node，finish指向最后一个已使用node：





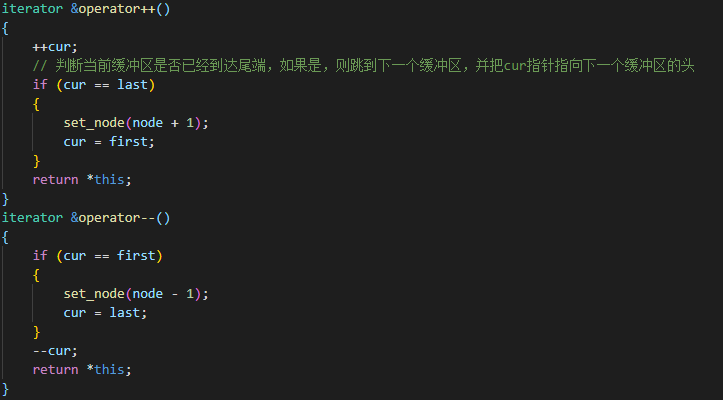
#### set\_node()

deque迭代器的一个重要操作：一旦行进到缓冲区边缘，需要调用该函数调整当前node指向的缓冲区，更新first、finish迭代器：



#### operator++和operator--

有了set\_node()之后，这两个操作就较为容易实现：



### 4.4.3. deque的内存构造与管理

deque维护着管控中心map，迭代器start和finish，也必须记住当前map的大小，一旦map所提供的节点不足，就必须再配置更大的一块map。

**map何时需要重新整治：**

1. 如果map的尾端节点备用空间不足-->reallocate\_map()：在尾端再添加一个元素，到达当前缓冲区的尾端时，引发新缓冲区的配置，同时也导致迭代器finish的状态改变。
2. 如果map的前端节点备用空间不足-->reallocate\_map()：在头端再添加一个元素，到达当前缓冲区的头端时，引发新缓冲区的配置，同时也导致迭代器start的状态改变。

**reallocate\_map():**

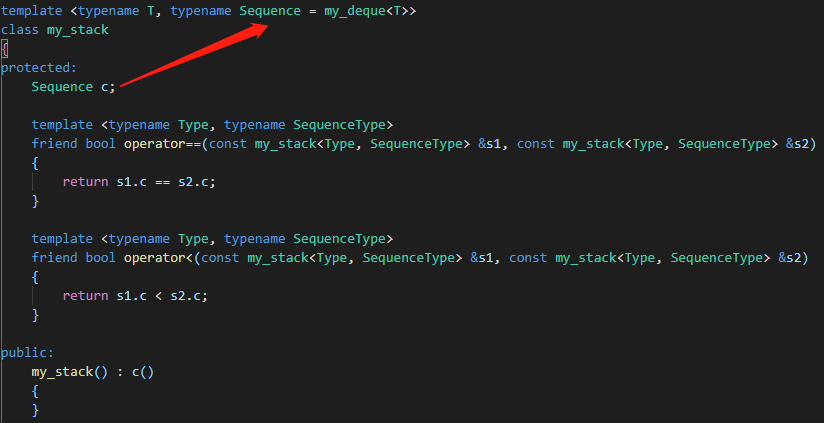
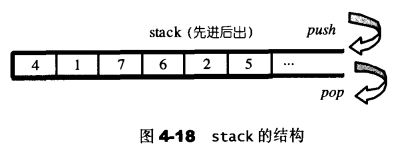
1. map的所有节点数>新的节点数(此时说明并不是map被占满了，而是一端map已经满了),调整start和finish迭代器。
2. map的所有节点数 < 新的节点数
3. 配置一块空间，准备给新map使用
4. 把原map内容拷贝过来
5. 释放原map
6. 设定新map的起始地址与大小

注：

1. 以上令nstart,nfinish指向map所拥有之全部节点的最中央区段，保持在最中央，可使头尾两端的扩充能量一样大，每个节点可对应一个缓冲区。
2. deque的最初状态（无任何元素时）保留一个缓冲区，即map中至少有一个节点。

## 4.5. stack

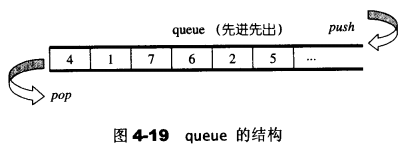
stack是一种先进后出的数据结构，它只有一个出口。stack允许新增元素、取得最顶端元素、移除元素。stack不允许有遍历行为。



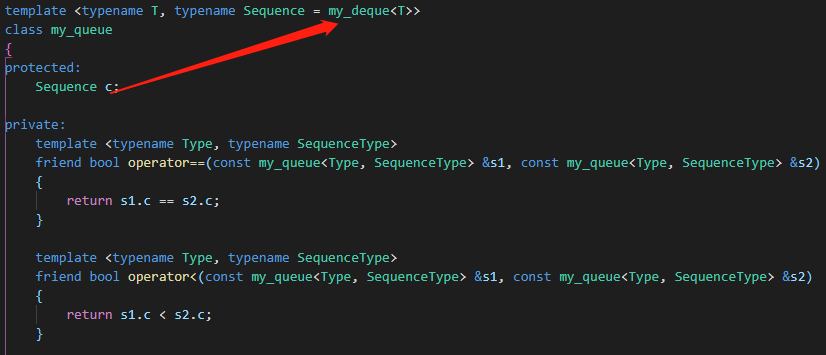
stack以deque作为底部结构，并封闭其头端开口。也可以以手动改为以list为底部结构。

## 4.6. queue

queue是一种先进先出FIFO的数据结构，它有两个出口。queue允许新增元素、移除元素、从最低端加入元素、取得最顶端元素。queue不允许有遍历行为。

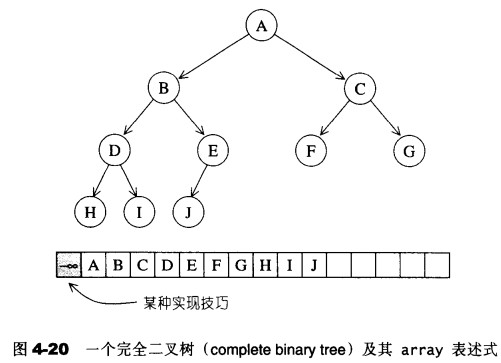


queue以deque作为底部结构，并封闭其底端的出口和头端的入口。也可以以手动改为以list为底部结构。



## 4.7. heap

heap并不属于STL容器组件，它是priority queue的幕后助手。priority queue允许用户以任何次序将任何元素推进容器内，但取出时一定是从优先权最高的元素开始取。priority queue的底层机制是二叉堆。



二叉堆是一颗完全二叉树，整棵树没有任何漏洞。二叉堆可用数组实现，数组编号从1开始，对于节点i，2\*i是其左子树，2\*i+1是其右子树。使用一个动态数组vector和一组heap算法可以实现元素插入、删除、取极值。最大堆的最大值在根节点，最小堆的最小值在根节点，根据上图的表示，二叉堆的极值位于数组的起始处。

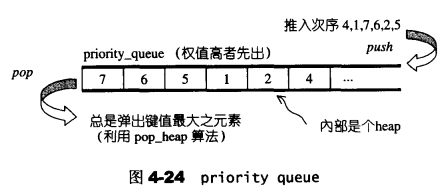
heap算法包括push\_heap()、pop\_heap()、sort\_heap()。

注意：**heap没有迭代器**，因为heap的所有元素都必须遵循特别的排列规则。

## 4.8. priority\_queue

priority\_queue是一个拥有权值的queue，它允许加入新元素、移除旧元素、审视元素值等功能。由于这是一个queue，所以只允许在顶端取出元素、在尾部加入元素，priority\_queue没有迭代器。

默认情况下，priority\_queue底部以一个最大堆完成，后者是一个以vector为底部数据结构的完全二叉树。



# 第五章 关联式容器

标准STL关联式容器包括set、map及其衍生的multiset和multimap，它们底层实现都是红黑树。不在STL标准之内的关联式容器包括hash table，以及以hash table为底层机制的unordered\_set、unordered\_map、unordered\_multiset和unordered\_multimap。

关联式容器的每个数据由一个键值(key)和一个实值(value)组成。当元素被插入到关联式容器时，容器内部结构依据键值大小，以特定规则将元素放置于适当位置。关联式容器没有所谓的头尾，只有最大值和最小值，因此没有push\_back()、push\_front()、pop\_back()、begin()、end()等操作。

## 5.1. 树

## 5.2. 红黑树

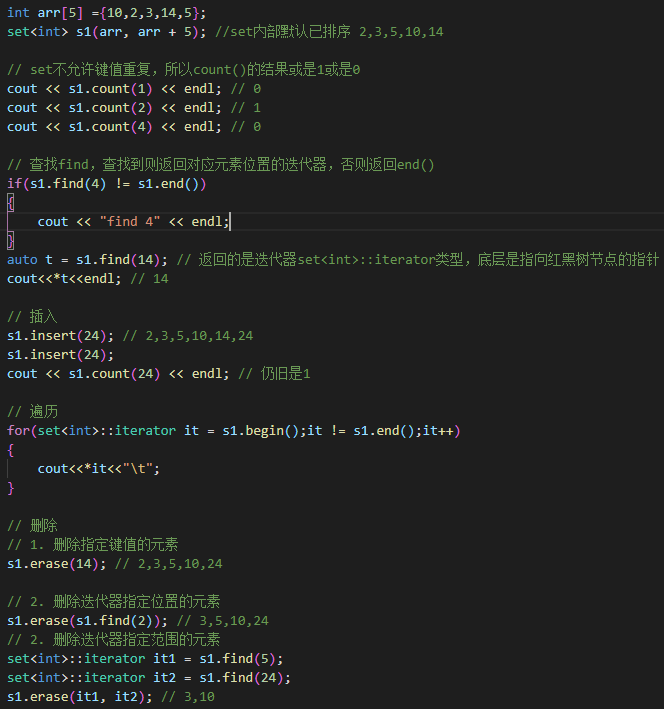
## 5.3. set

set的特性是，所有元素都会根据元素的键值自动被排序。**set的键值key和是指value是同一个**，即键值就是实值，实值就是键值。set不允许两个元素有相同的键值。

不允许改变set的元素值，因为set的元素值同时也是其键值key，改变了key会影响set组织。

与list一样，当客户端对set进行新增insert或删除erase操作后，操作之前的迭代器在操作完成后依然有效，被删除的那个元素的迭代器除外。

#### set使用示例



#### 问题1：为何每次insert之后，以前保存的iterator不会失效？:

iterator这里就相当于指向节点的指针，内存没有变，指向内存的指针怎么会失效呢(当然被删除的那个元素本身已经失效了)。相对于vector来说，每一次删除和插入，指针都有可能失效，调用push\_back在尾部插入也是如此。因为为了保证内部数据的连续存放，iterator指向的那块内存在删除和插入过程中可能已经被其他内存覆盖或者内存已经被释放了。即使时push\_back的时候，容器内部空间可能不够，需要一块新的更大的内存，只有把以前的内存释放，申请新的更大的内存，复制已有的数据元素到新的内存，最后把需要插入的元素放到最后，那么以前的内存指针自然就不可用了。特别时在和find等算法在一起使用的时候，牢记这个原则：不要使用过期的iterator。

#### 问题2：当数据元素增多时，set的插入和搜索速度变化如何？:

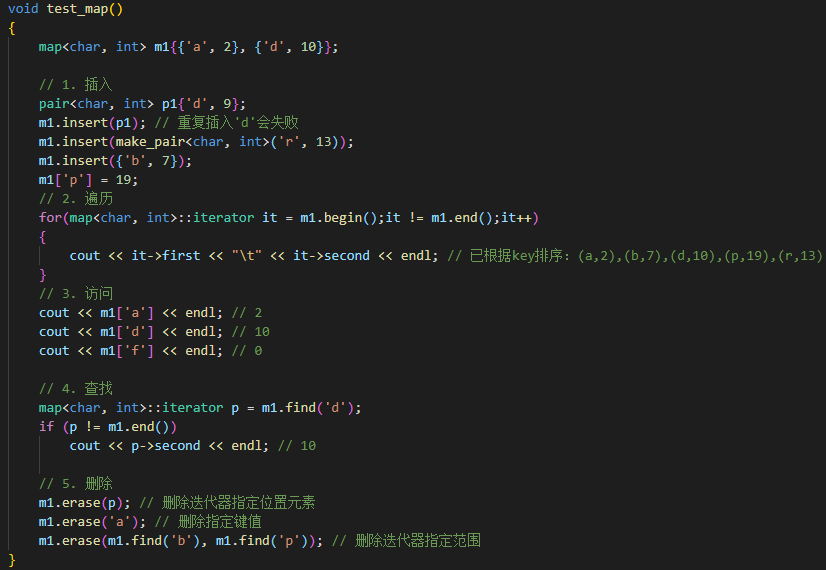
在set中查找是使用二分查找，也就是说，如果有16个元素，最多需要比较4次就能找到结果，有32个元素，最多比较5次。那么有10000个呢？最多比较的次数为log10000，最多为14次，如果是20000个元素呢？最多不过15次。所以，当数据量增大一倍的时候，搜索次数只不过多了1次。

## 5.4. map

map的特性是，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value和键值key。pair的第一个元素被视为键值key，第二元素被视为实值value。map不允许两个元素拥有相同的键值。

与list一样，当客户端对map进行新增insert或删除erase操作后，操作之前的迭代器在操作完成后依然有效，被删除的那个元素的迭代器除外。

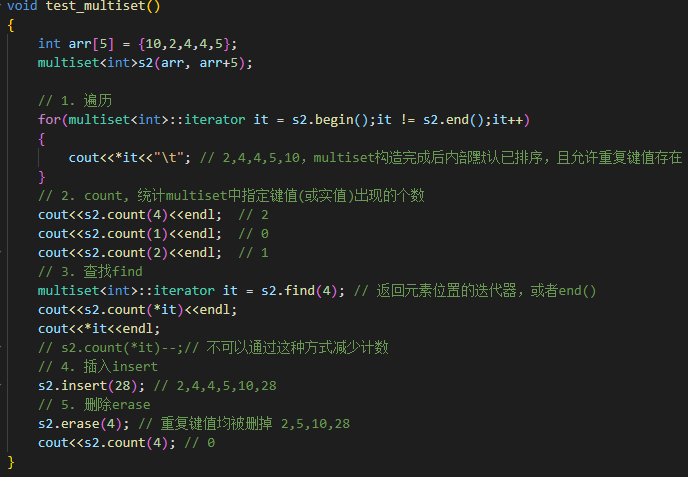
### map使用示例



## 5.5. multiset

multiset的特性和用法与set完全相同。唯一的差异在于它允许键值重复，因此它的插入操作采用的是底层机制RB-Tree的insert\_equal()而不是insert\_unique()。

### multiset使用示例



## 5.6. multimap

multimap的特性和用法与set完全相同。唯一的差异在于它允许键值重复，因此它的插入操作采用的是底层机制RB-Tree的insert\_equal()而不是insert\_unique()。

**注意：关联容器的额外的类型别名**

key\_type 此容器类型的关键字类型

mapped\_type 每个关键字关联的类型，只适用于map

value\_type 对于set，与key\_type相同；对于map，为pair<const key\_type,mapped\_type>

## 5.7. hash table

### 5.7.1. 哈希表和散列函数

哈希表（Hash table，也叫散列表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，**它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度**。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

**记录的存储位置=f(关键字)**

这里的对应关系f称为散列函数，又称为哈希（Hash函数），采用散列技术将记录存储在一块连续的存储空间中，这块连续存储空间称为散列表或哈希表（Hash table）。

通常散列表的大小是素数，以尽可能避免不同的关键字散列到同一个位置。

### 5.7.2. 哈希冲突及解决方法

不同的关键字经过散列函数后映射到哈希表相同的位置，称为散列冲突。

#### 分离连接法

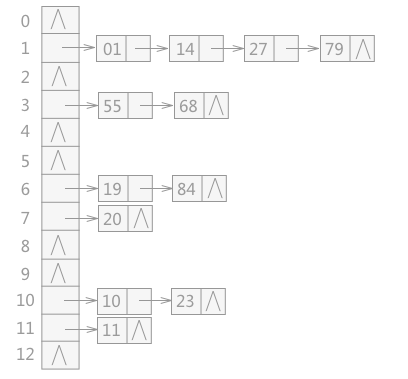
将散列到同一个值的所有元素保留到一个链表中，这些表都有表头，如下图。

find()：使用散列函数来确定考察哪个表，并返回所找到的被查找项所在的位置。

insert()：我们遍历一个相应的表，以检查该元素是否已经处在适当的位置。如果允许重复插入，则需要设计一个额外的计数；如果是新插入的元素，则放在链表头或尾。

**装填因子**（load factor）是指散列表中元素个数与散列表大小的比值。分离连接法的一般法则是使得表的大小尽量与元素个数差不多，即使得load factor的值约为1。所以尽可能使散列表大小为素数。

分离连接法的不足之处是需要使用指针，在新分配空间的时候会耗时。



散列函数H(x) = x mod 13

#### 开放定址法

开放定址法是指，在散列冲突发生以后，尝试算则另外的单元，直到找出空的单元为止。

##### 线性探测法

从发生哈希冲突位置开始，依次向后探测，直到找到下一个空位置为止。

**缺点：**依次往后找，导致花费时间较多；而且即使表相对较空，这样占据的单元会形成一些区块（一次聚集），之后加剧查找空位置的时间。

##### 平方探测法

为消除线性探测法中的一次聚集问题，平凡探测的冲突函数为二次函数。

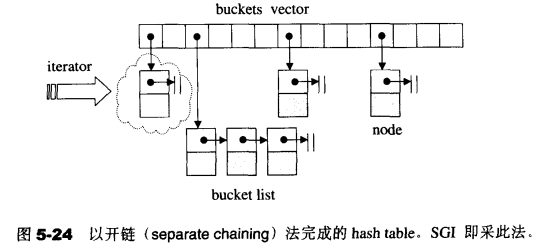
**缺点：**如果表大小不是素数，一旦表被填满超过一半，就不能保证一次找到一个空单元了，因为最多有表的一半可以用作解决冲突的备选位置。

#### 再散列

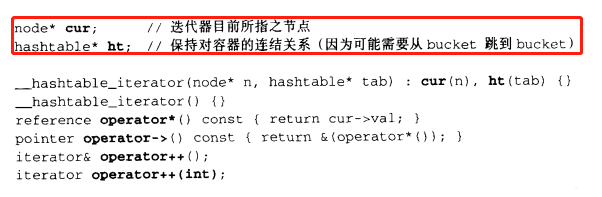
在使用哈希函数去散列一个输入的时候，输出是同一个位置就再次哈希，直至不发生冲突位置。

## 5.8. STL中的hash table

STL使用分离连接法完成hash table的实现。Hash table表格内的元素为桶bucket。bucket维护了一个链表，链表中的节点node除了保存元素值value以外，还有一个指向下一个node的指针next。

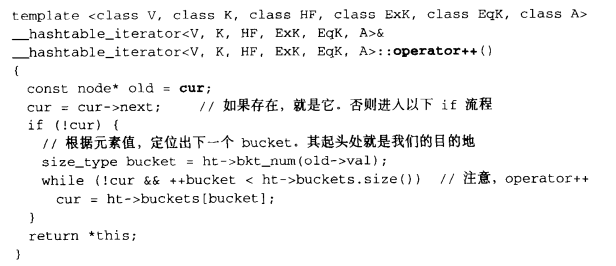


### 5.8.1. hash table的迭代器

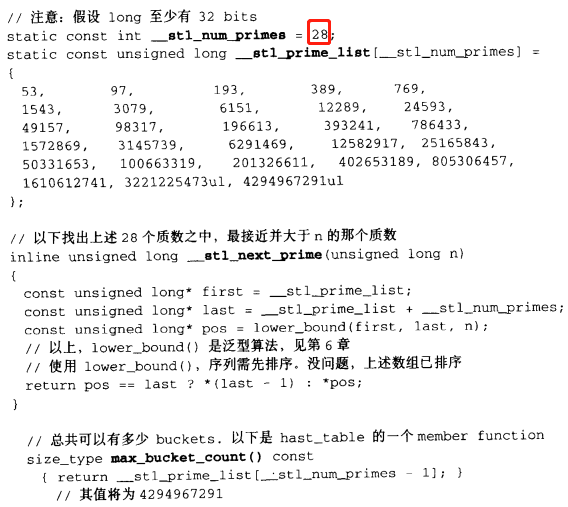


**hash table的迭代器是forwared\_iterator，没有后退操作（operator()--），也没有逆向迭代器**。

hash table迭代器必须永远维系着与整个bucket vector的关系，并记录目前所指的节点。其前进操作首先尝试从目前所指的节点出发，前进一个位置（利用node的next指针）。如果正好是当前bucket中list的尾端，则跳至下一个bucket的list的头部。



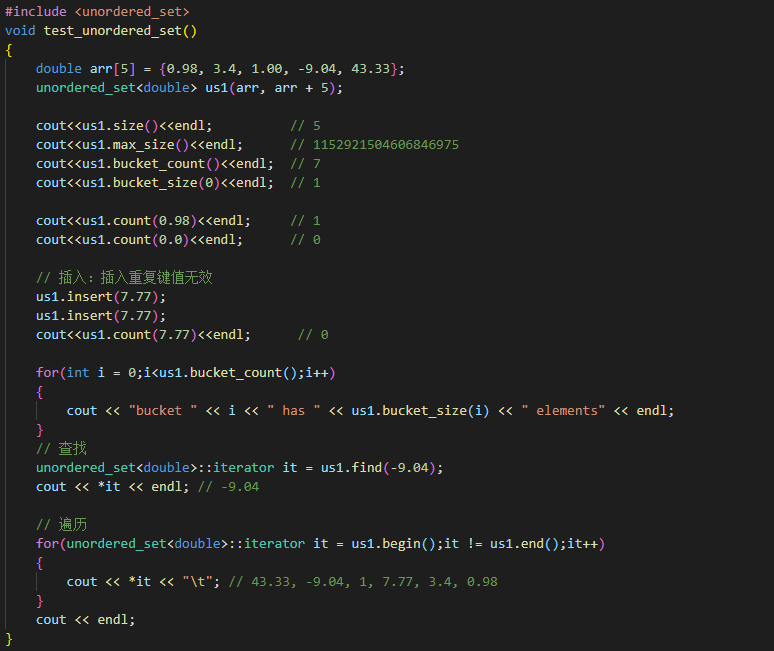
STL以质数来设计hash table大小，并且先将28个质数（逐渐呈现约两倍的关系）计算好，以备随时访问，同时提供一个函数，来查询在这28个质数中，最接近某数并大于某数的质数。



当客户插入元素时，hash table会判断是否需要重建表格。如果需要（元素个数大于table size），则从已上28个质数中找出下一个质数，作为新的hash table的size，并设立新的buckets。

## 5.9. unordered\_set

unordered\_set底层是hashtable，键值就是实值，实值就是键值。与set不同的是，set的元素会自动排序，而hashtable则不是。



# 第七章 仿函数

仿函数是一个行为类似函数的对象，其类别定义中必须重载function call运算符(operator ())。STL的仿函数，以操作数个数分类，可分为一元和二元仿函数。以功能划分，可以分为算术运算、关系运算和逻辑运算三大类。

STL所提供的各种算法，往往有两个版本，其中一个版本表现出最常用（或最直观的）的某种运算，第二个版本则表现出最泛化的演算流程，允许用户“以template参数来指定所要采行的策略”。拿accumulate来说，他的一般行为是将指定范围内的所有元素相加，第二版本则允许你指定某种“操作”，取代第一版本中的“相加”行为。要将某种“操作”当做算法的参数，唯一办法就是先将该“操作”（可能拥有数条以上的指令）设计为一个函数，在将函数当做算法的一个参数；或是将该“操作”设计为一个所谓的仿函数（就语言层面而言是个class），再以该仿函数产生一个对象，并以此对象作为算法的一个参数。

而为什么函数指针可以达到“将数组操作当做算法的参数”，那又何必有所谓的仿函数呢？原因在于函数指针毕竟不能满足STL对抽象性的要求，也不能满足软件积木的要求---->函数指针无法和STL其他组件搭配,产生更灵活的变化。

我自认为之所以要需要使用仿函数而不是函数指针的一个原因就是，在基于泛化编程的基础上，仿函数拥有函数体，而函数指针就是一个空指针，因为泛化编程只支持传入一个类型，必须保证泛化的类仅靠类型（int ,class（代表具体的类）) 就可以达到某种“操作”。

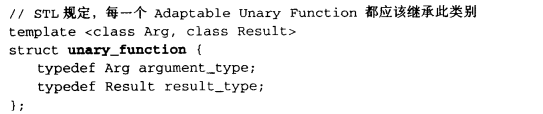
## 7.1. 可配接的关键

STL规定，每一个仿函数必须定义自己的相应型别。这些型别是为了让配接器能够取出、获得仿函数的某些信息。相应型别只是一些typedef，所有必要的操作在编译器就已全部完成，对程序的执行效率无任何影响。

仿函数的相应型别主要用来表现参数型别和传回值型别。<stl\_function.h>定义了两个类，分别代表一元仿函数和二元仿函数，其中没有任何数据成员和成员函数，唯有一些型别定义。任何仿函数，只要依个人需求继承其中一个class，便自动拥有了配接能力。

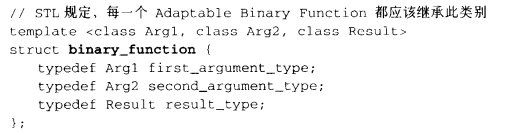
### 7.1.1. unary\_function

unafy\_funtion用来呈现一元函数的参数型别和返回值型别：



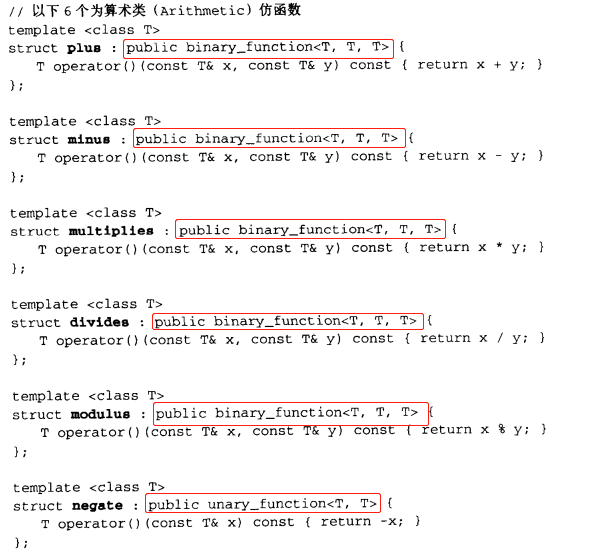
### 7.1.2. binary\_funciton

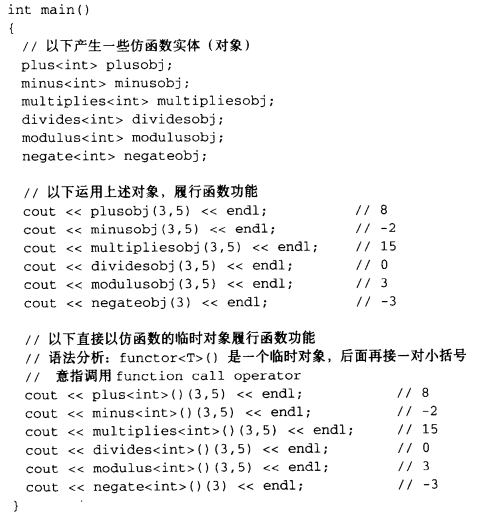
binary\_funtion用来呈现二元函数的第一参数型别、第二参数型别和返回值型别：



## 7.2. 算数类仿函数

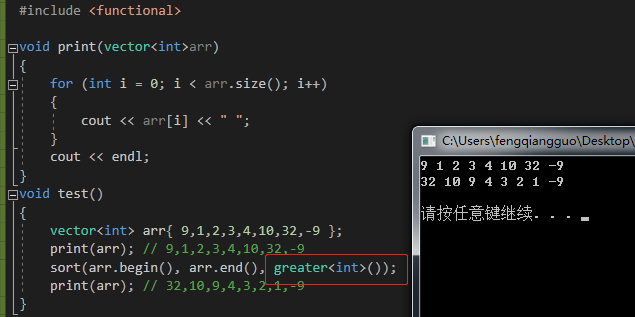
STL内建的算数类仿函数支持加、减、乘、除、模数、否定运算，否定运算是一元运算，其他为二元运算。





## 7.3. 关系运算类仿函数

支持等于、不等于、大于、大于等于、小于、小于等于六种运算。



## 7.4. 逻辑运算类仿函数

支持逻辑与、逻辑或和逻辑非。

## 7.5. 证同identity、选择select、投射project