- STL
  - 概述
  - 迭代器
    - 设计迭代器的原因
    - 迭代器和指针的区别
    - 迭代器的分类
      - 输入迭代器
      - 输出迭代器
      - 前向迭代器
      - 双向迭代器
      - 随机迭代器
  - 容器
    - 共性
    - 顺序容器
      - vector
        - vector的底层实现
        - vector可以存放任意类型的数据结构吗
        - vector使用的注意点及其原因，频繁对vector调用push_back()对性能的影响和原因
      - list
      - deque
        - deque的实现原理
      - stack、queue、priority_queue
    - 关联容器
      - set
      - multiset
      - map
        - map用find和[]的区别
      - multimap
      - hashtable
      - hash_map
        - 什么时候需要用hash_map，什么时候需要用map?
      - hash_set
      - unordered_map
    - 自动扩容
      - vector扩容原理
      - hash_map扩容
      - deque自动扩容

STL

概述

STL有容器，空间配置器，适配器，迭代器，仿函数以及算法这6个组件，它们六者关系大概如下：容器通过配置器取得数据存储空间,算法通过迭代器获取容器内容,仿函数可以协助算法完成不同的策略变化,适配器是通过组合特定的容器实现的一种新的数据结构。

迭代器

STL将数据和算法分开。 数据存储在容器里，使用算法进行操作。迭代器是这两者之间的粘合剂，让算法与容器可以进行交互。

Iterator（迭代器）模式又称Cursor（游标）模式，用于提供一种方法顺序访问一个聚合对象中各个元素,而又不需暴露该对象的内部表示。或者这样说可能更容易理解：Iterator模式是运用于聚合对象的一种模式，通过运用该模式，使得我们可以在不知道对象内部表示的情况下，按照一定顺序（由iterator提供的方法）访问聚合对象中的各个元素。

由于Iterator模式的以上特性：与聚合对象耦合，在一定程度上限制了它的广泛运用，一般仅用于底层聚合支持类，如STL的list、vector、stack等容器类及ostream_iterator等扩展iterator。

设计迭代器的原因

1、通过迭代器访问容器，可以避免许多错误，同时还能隐藏容器的具体实现。

2、迭代器可以保证对所有容器的基本遍历方式，都是一样的，实现算法时若需要遍历，则使用迭代器，则可以不用关注容器的具体类型，实现数据结构和算法的分离。

3、迭代器本身有很多优点，可以弥补C++语言的不足，比如它的iterator_category，可以得到迭代器所指向的类别，这样可以根据不同的类别的特性，提供不同的算法。

迭代器和指针的区别

迭代器不是指针，是类模板，表现的像指针。他只是模拟了指针的一些功能，重载了指针的一些操作符，->、*、++、--等。迭代器封装了指针，是一个“可遍历STL（ Standard Template Library）容器内全部或部分元素”的对象，本质是封装了原生指针，是指针概念的一种提升（lift），提供了比指针更高级的行为，相当于一种智能指针，他可以根据不同类型的数据结构来实现不同的++，--等操作。迭代器返回的是对象引用而不是对象的值，所以cout只能输出迭代器使用*取值后的值而不能直接输出其自身。

迭代器的分类

STL中的迭代器可分为类：随机存取迭代器（random-access-iterator），双向存取迭代器(bidirectional-access-iterator)，前向迭代器(forward iterator)，输入迭代器(input-iterator)，输出迭代器(output-iterator)。它们之间的继承关系如下：input-iterator
output-iteartor
forward-iterator:output-iteartor , input-iterator
bidirectional-access-iterator : forward-iterator
random-access-iterator:bidirectional-access-iterator

输入迭代器

输入迭代器也可以称之为前向的只读访问器，首先它提供了对容器的只读访问，其次它只能在容器中进行前向迭代（即只提供++操作）。所有的容器的迭代器都具有输入迭代器的特征。通过输入迭代器你可以进行下面三种操作：
1.V = *X++
2.V = *X,X++
3.V = *X,++X
注：V为值，X为迭代器
*p 复引用迭代器，作为右值
p=p1 将一个迭代器赋给另一个迭代器

p==p1 比较迭代器的相等性
p!=p1 比较迭代器的不等性

输出迭代器

输出迭代器也可以称之为前向的只写访问器，首先它提供了对容器的只写访问，其次它只能在容器中进行前向迭代（即只提供++操作）。通过输出迭代器你可以进行下面三种操作：
1.*X++ = V
2.*X = V, X++
3.*X = V, ++X
注：V为值，X为迭代器

前向迭代器

前向迭代器继承自输入和输出迭代器，因此具有输入和输出迭代器的所有特征，也即提供对容器数据结构的读写访问但也只具有前向迭代的能力（即只提供++操作）。因此，你可以对前向迭代器进行操作：R == S /++R == ++S。

双向迭代器

双向存取迭代器从前向迭代器继承过来，因而具有前向迭代器的所有特征，双向存取迭代器还具有后向访问能力（即只提供--操作）。

随机迭代器

随机存取迭代器从双向存取迭代器继承过来，因而具有双向存取迭代器的所有特征。所不同的是，利用随机存取迭代器你可以对容器数据结构进行随机访问。
p+=i 将迭代器递增i位
p-=i 将迭代器递减i位
p+i 在p位加i位后的迭代器
p-i 在p位减i位后的迭代器
p[i] 返回p位元素偏离i位的元素引用
p<p1 如果迭代器p的位置在p1前，返回true，否则返回false
p<=p1 p的位置在p1的前面或同一位置时返回true，否则返回false
p>p1 如果迭代器p的位置在p1后，返回true，否则返回false
p>=p1 p的位置在p1的后面或同一位置时返回true，否则返回false

容器

STL中的容器可分为顺序容器(Sequence Container)和关联容器(Associative Container)。
顺序容器：vector、deque、list、stack、queue、priority_queue（后三个为非STL标准）
关联容器：set、multiset、map、multimap（底层都是红黑树）、hashtable、hash_set、hash_map、hash_multiset、hash_multimap（后五个为非STL标准，后四个底层为hashtable）

共性

所有的容器都是物之所在，这就决定了它们必然存在很多共性，这些共性包括迭代器、大小等属性。容器与容器之间的主要区别体现在对数据的操作上。
每类容器都包含四个迭代器：iterator(正向迭代器)、const_iterator(常正向迭代器)、reverse_iterator(反向迭代器)、const_reverse_iterator(常反向迭代器)。
因此你可以按照下面的方式获取每个容器的相应的迭代器：
获取正向迭代器
C::iterator it = c.begin();
C ::iterator it = c.end();
获取反向迭代器
C ::reverse_iterator it = c.rbegin();
C :: reverse_iterator it = c.rend()
获取常正向迭代器
C::const_iterator it = c.begin();
C :: const_iterator it = c.end();
获取常反向迭代器
C :: const_ reverse_iterator it = c.rbegin();
C :: const_ reverse_iterator it = c.rend()
注：C为容器类型，c为C的实例
所有容器是数据的存在之处，可以看作的是数据的集合，因此它们都会有大小、是否为空等属性，因此你可以按照下面的方式获取所有的容器的公共属性：
获取容器的大小
c.size();
判断容器是否为空
c.empty();

顺序容器

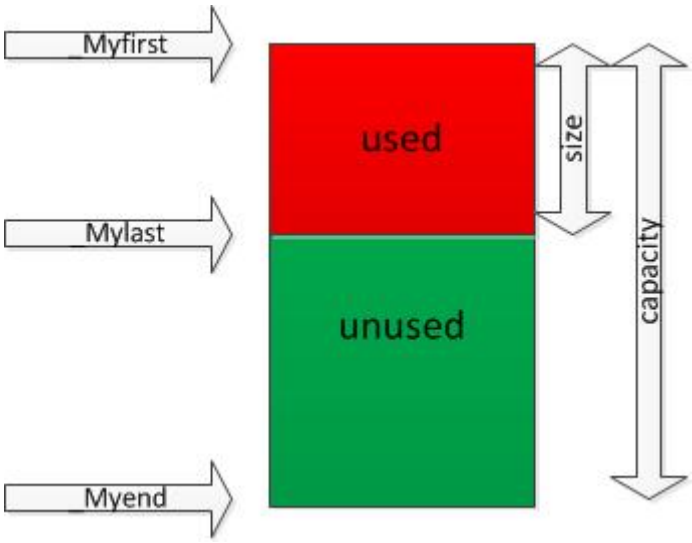
顺序容器中所有的元素在容器中的物理位置都是按照特定的次序进行存放的，区别于关联容器的是顺序容器中的元素的位置都是既定的。被纳入STL标准的顺序容器包括vector、list、deque。
序列容器之间的共性除了容器之间应有的共性之外，还有对数据操作的接口(非实现)上：
c.push_back
c.pop_back
c.push_front
c.pop_front
c.back
c.front
c.erase
c.remove

vector

vector和数组具有同样的内存处理方式。不同于数组的是：数组是静态空间，一旦分配了就不能被改变，因而空间的分配非常地不灵活；vector是动态空间，即空间可以被动态分配，因而空间的分配很灵活。
vector中的迭代器的种类为随机存取迭代器。

vector的底层实现

简单理解，就是vector是利用三个指针来表示的，基本示意图如下：



两个关键大小：
大小：size=_Mylast - _Myfirst；
容量：capacity=_Myend - _Myfirst；
分别对应于resize()、reserve()两个函数。
size表示vector中已有元素的个数，容量表示vector最多可存储的元素的个数；为了降低二次分配时的成本，vector实际配置的大小可能比客户需求的更大一些，以备将来扩充，这就是容量的概念。即capacity>=size，当等于时，容器此时已满，若再要加入新的元素时，就要重新进行内存分配，整个vector的数据都要移动到新内存,释放之前的内存，再插入新增的元素。二次分配成本较高，在实际操作时，应尽量预留一定空间，避免二次分配。

vector可以存放任意类型的数据结构吗

vector可以存放自定义的结构体，方法有：放入这个结构体类型变量的副本或者放入指向这个结构体类型变量的指针。

vector使用的注意点及其原因，频繁对vector调用push_back()对性能的影响和原因

在一个vector的尾部之外的任何位置添加元素，都需要重新移动元素。而且，向一个vector添加元素可能引起整个对象存储空间的重新分配。重新分配一个对象的存储空间需要分配新的内存，并将元素从旧的空间移到新的空间

list

List由双向链表（doubly linked list）实现而成，元素也存放在堆中，每个元素都是放在一块内存中，他的内存空间可以是不连续的，通过指针来进行数据的访问，这个特点使得它的随机存取变得非常没有效率，因此它没有提供[]操作符的重载。但是由于链表的特点，它可以很有效率的支持任意地方的插入和删除操作。相对vector，list没有capaciy属性。
list中的迭代器的种类为双向存取迭代器。

deque

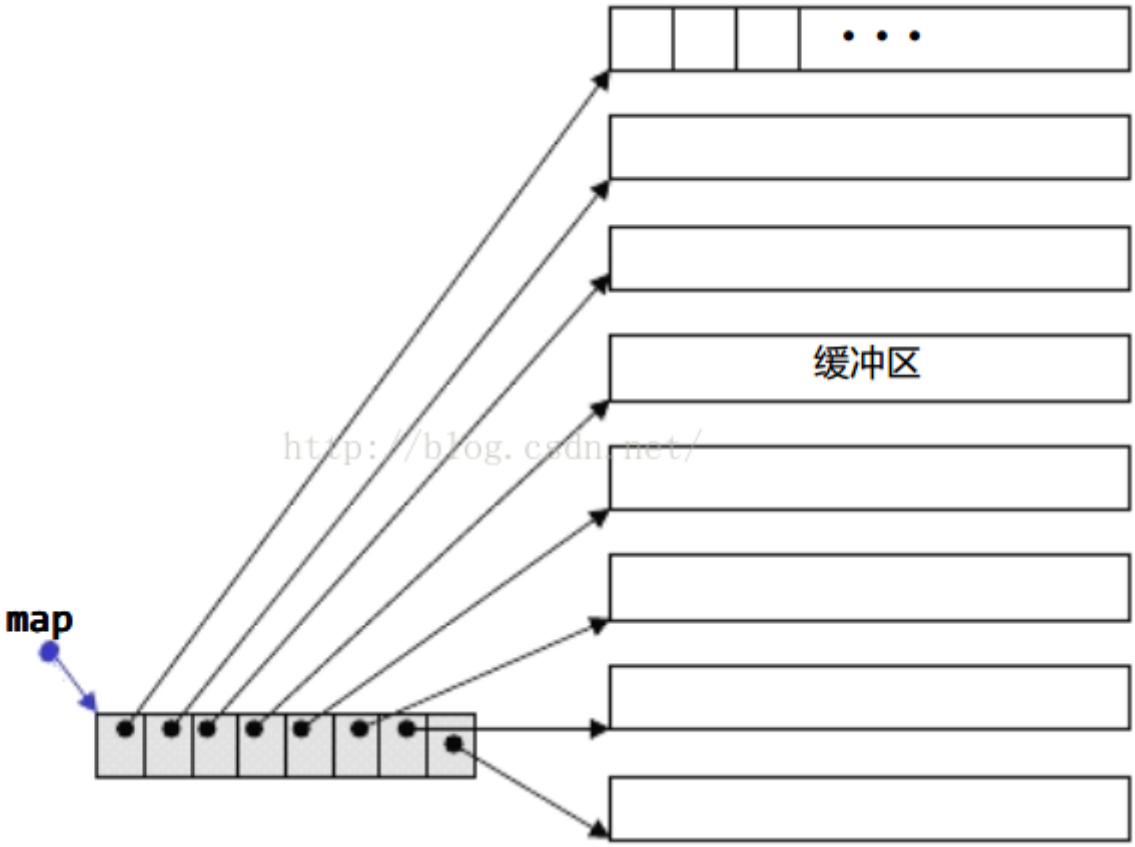
deque，即双向链表。相对vector，deque也是连续空间，但不是vector的连续线性空间。它允许较为快速地随机访问但它不像vector一样把所有对象保存在一个连续的内存块，而是多个连续的内存块。并且在一个映射结构中保存对这些块以及顺序的跟踪。
deque中的迭代器的种类为随机存取迭代器。

deque的实现原理

deque是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的最大任务，便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。避开了“重新配置、复制、释放”的轮回，代价则是复杂的迭代器架构。

受到分段连续线性空间的字面影响，我们可能以为deque的实现复杂度和vector相比虽不中亦不远矣，其实不然。主要因为，既是分段连续线性空间，就必须有中央控制，而为了维持整体连续的假象，数据结构的设计及迭代器前进后退等操作都颇为繁琐。deque的实现代码分量远比vector或list都多得多。

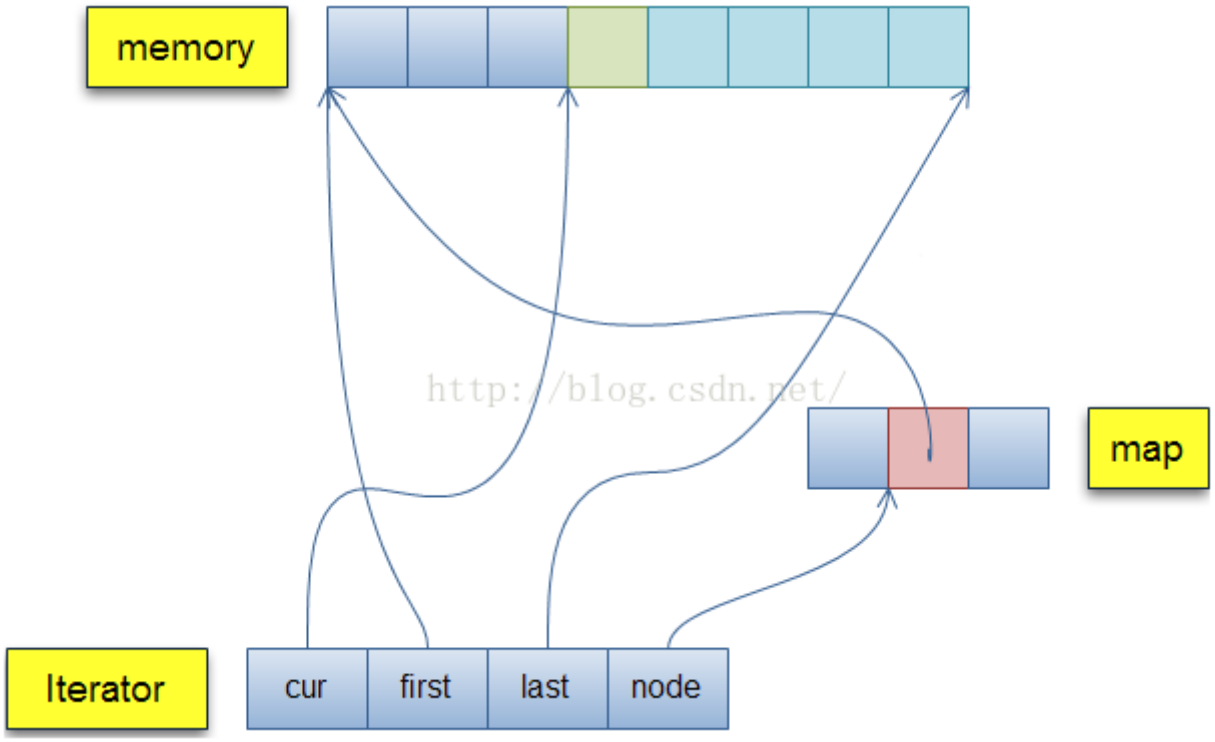
deque采用一块所谓的map（注意，不是STL的map容器）作为主控。这里所谓map是一小块连续空间，其中每个元素（此处称为一个节点，node）都是指针，指向另一段（较大的）连续线性空间，称为缓冲区。缓冲区才是deque的储存空间主体。SGI STL 允许我们指定缓冲区大小，默认值0表示将使用 512 bytes 缓冲区。
deque的整体架构如下图所示：



deque 的迭代器

让我们思考一下，deque的迭代器应该具备什么结构，首先，它必须能够指出分段连续空间（亦即缓冲区）在哪里，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，如果是，一旦前进或后退就必须跳跃至下一个或上一个缓冲区。为了能够正确跳跃，deque必须随时掌握管控中心（map）。所以在迭代器中需要定义：当前元素的指针，当前元素所在缓冲区的起始指针，当前元素所在缓冲区的尾指针，指向map中指向所在缓区地址的指针，分别为cur, first, last, node。

指针结构如下图所示：



stack、queue、priority_queue

Stack(栈)是一种后进先出的数据结构，也就是LIFO(last in first out)，最后加入栈的元素将最先被取出来，在栈的同一端进行数据的插入与取出，这一段叫做“栈顶”。
template <class T, class Container = deque > class stack;
如上，这对尖括号中有两个参数，第一个是T，表示栈中存放的数据的类型，比如int，double，或者结构体之类。
第二个参数指明底层实现的容器类型，也就是指明这个栈的内部实现方式，比如vector，deque，list。如果不指明它，默认使用deque(双端队列)。当然一般情况下不需要指定这一项参数。

queue与stack模版非常类似，queue模版也需要定义两个模版参数，一个是元素类型，一个是容器类型，元素类型是必要的，容器类型是可选的，默认为dqueue类型。

priority_queue模版类有三个模版参数，元素类型，容器类型，比较算子。其中后两个都可以省略，默认容器为vector，默认算子为less，即小的往前排，大的往后排（出队时序列尾的元素出队）。

关联容器

关联容器(Associative Container)提供了根据key快速检索数据的能力。在关联容器(Associative Container)中，key和元素都是成对(pair)存在的,你可以调用std::make_pair使用key和元素值来构建一个pair。
STL提供的关联容器包括set、multiset、map、multimap。
set和map只支持唯一键(unique key)，即对个key最多只保存一个元素。multiset和multimap则支持多个key，一个key可以对应多个元素。
set和map的区别在于，在set里面key和元素是同一个值，而在map里面key和元素分开存储。

set

set是集合的抽象数据结构(ADT)。不同于数学意义上的集合，STL中的set的所有的元素都是有序的而且set中所有的元素都是唯一的，而且系统能根据元素的值自动进行排序。底层采用红黑树来实现。
set中的迭代器的种类为双向存取迭代器。

multiset

multiset和set基本相同，所不同的是，一个multiset中的元素是可以重复的。
multiset中的迭代器的种类为双向存取迭代器。

map

map是字典的抽象数据结构(ADT)。map中的所有的元素都会根据key自动进行排序，而且所有的元素都是唯一的。map中的所有的元素都是pair，即键(key)和值(value)组成的序列(pair中的第一个元素为key，第二个元素为value)。底层使用红黑树来实现。时间复杂度是O(logN)
map中的迭代器的种类为双向存取迭代器。

map用find和[]的区别

最大的区别是当待查找的 KEY 不存在时：
1.map::find 返回一个空迭代器（map::end）。
2.map::operator[] 将用 VALUE 默认的构造函数创建一个对象并插入到 map 中，将其返回。

multimap

multimap和map基本相同，所不同的是，一个multimap中的key可以重复。
multimap中的迭代器的种类为双向存取迭代器。

hashtable

其基本原理是：使用一个下标范围比较大的数组来存储元素。把关键字Key通过一个固定的算法函数即所谓的哈希函数（散列函数）转换成一个整型数字，然后就将该数字对数组长度进行取余，取余结果就当作数组的下标，将value存储在以该数字为下标的list空间里。也可以简单的理解为，按照关键字key为每一个元素“分类”，然后将这个元素存储在相应“类”所对应的地方，称为桶。
为了避免哈希表太大，需要用到某种映射函数，将大数映射为小数，这种函数称为散列函数hash function。但hash function会带来碰撞问题，即不同的元素被映射到相同位置。
为了解决碰撞问题有三种方法：线性探测、二次探测、开链。
hashtable就是用的开链法。

hash_map

hash_map基于哈希表，哈希表的最大优点就是把数据的存储和查询消耗的时间大大降低，几乎可以看成是常数时间；而代价仅仅是消耗比较多的内存。

hash_map，首先分配一大块内存，形成许多桶，利用hash函数，对key进行映射到不同区域（桶）进行保存。其插入过程是：
1、得到key
2、通过hash函数得到hash值
3、得到桶号（一般都为hash值对桶数求模）
4、存放key与value在桶内。

其取值过程是：
1、得到key
2、通过hash函数得到hash值
3、得到桶号
4、比较桶的内部元素是否与key相等，若不想等，则没有找到
5、取出相等记录的value。

什么时候需要用**hash_map**，什么时候需要用**map?**

总体来说，hash_map查找速度会比map快，而且查找速度基本和数据量大小相关，属于常数级别；而map的查找速度是log(n)级别。并不一定常数就比log(n)小，hash还有hash函数的耗时。如果你考虑效率，特别是元素达到一个数量级时，考虑考虑hash_map。但若你对内存使用特别严格，希望程序尽可能少消耗内存，那么一定要小心。
选择基于三个因素：查找速度，数据量，内存使用。

hash_set

hash_set是以哈希表（Hash table）作为底层数据结构的。

unordered_map

类似于hash_map，但是hash_map不是标准库的

自动扩容

vector扩容原理

1、新增元素：Vector通过一个连续的数组存放元素，如果集合已满，在新增数据的时候，就要分配一块更大的内存，将原来的数据复制过来，释放之前的内存，在插入新增的元素；
2、对vector的任何操作，一旦引起空间重新配置，指向原vector的所有迭代器就都失效了；
3、初始时刻vector的capacity为0，塞入第一个元素后capacity增加为1；
4、不同的编译器实现的扩容方式不一样，VS2015中以1.5倍扩容，GCC以2倍扩容。

为什么：
1、vector在push_back以成倍增长可以在均摊后达到O(1)的事件复杂度，相对于增长指定大小的O(n)时间复杂度更好。

2、为了防止申请内存的浪费，现在使用较多的有2倍与1.5倍的增长方式，而1.5倍的增长方式可以更好的实现对内存的重复利用，因为更好。

hash_map扩容

扩容时需要满足两个条件：存放新值的时候当前hash_map所有元素的个数大于等于阈值；存放新值的时候当前存放数据发生hash碰撞。
STL会默认指定初始桶大小为16，负载因子是0.75，因此阈值就是16*0.75 = 12，所以当新插入元素时，当前的元素个数超过12，并且发生了冲突，就会扩容hash桶。扩容的大小是给之前的数组翻倍。
在hashmap数组扩容之后，最消耗性能的点就出现了：原数组中的数据必须重新计算其在新数组中的位置，并放进去，这就是resize。 所以如果已经预知hashmap中元素的个数，那么预设元素的个数能够有效的提高hashmap的性能，例如最多有1000个元素，则创建时：new HashMap(2048)（1024是不够的，要考虑负载因子：1024*0.75 = 768）。
另外桶的大小为2的幂次方时，hash_map的效率最高。这是因为：正常的取index方法为hash%length，但是由于位运算比取余快，所以代码中实现为index = hash & (length - 1)，所以只有length大小为2的次幂时：hash % length == hash & (length - 1)。

deque自动扩容

deque由一段一段构成 ,他是分段连续,而不是内存连续 当走向段的尾端时候自动跳到下一段 所以支持迭代器++ 操作,自动跳到下一段的方法由operator++实现。deque每次扩充 申请一个段。

迭代器失效

1. 对于序列式容器(如vector,deque)，序列式容器就是数组式容器，删除当前的iterator会使后面所有元素的iterator都失效。这是因为vetor,deque使用了连续分配的内存，删除一个元素导致后面所有的元素会向前移动一个位置。所以不能使用erase(iter++)的方式，还好erase方法可以返回下一个有效的iterator。

```
for (iter = cont.begin(); iter != cont.end();)
{
   (*it)->doSomething();
   if (shouldDelete(*iter))
      iter = cont.erase(iter);  //erase删除元素，返回下一个迭代器
   else
      ++iter;
}
```

对于序列式容器，比如vector，删除当前的iterator会使后面所有元素的iterator都失效。这是因为顺序容器内存是连续分配（分配一个数组作为内存），删除一个元素导致后面所有的元素会向前移动一个位置。（删除了一个元素，该元素后面的所有元素都要挪位置，所以，iter++，已经指向的是未知内存）。

2. 对于关联容器(如map, set,multimap,multiset)，删除当前的iterator，仅仅会使当前的iterator失效，只要在erase时，递增当前iterator即可。这是因为map之类的容器，使用了红黑树来实现，插入、删除一个结点不会对其他结点造成影响。erase迭代器只是被删元素的迭代器失效，但是返回值为void，所以要采用erase(iter++)的方式删除迭代器。
   map是关联容器，以红黑树或者平衡二叉树组织数据，虽然删除了一个元素，整棵树也会调整，以符合红黑树或者二叉树的规范，但是单个节点在内存中的地址没有变化，变化的是各节点之间的指向关系。
   所以在map中为了防止迭代器失效，在有删除操作时，常用如下方法：

```
for (iter = dataMap.begin(); iter != dataMap.end(); )
{
    int nKey = iter->first;
    string strValue = iter->second;
    if (nKey % 2 == 0){
        map<int, string>::iterator tmpIter = iter;
        iter++;
        dataMap.erase(tmpIter);
        //dataMap.erase(iter++) 这样也行

    }
    else{
        iter++;
    }
}
```

数组型数据结构：该数据结构的元素是分配在连续的内存中，insert和erase操作，都会使得删除点和插入点之后的元素挪位置，所以，插入点和删除掉之后的迭代器全部失效，也就是说insert(*iter)(或erase(*iter))，然后在iter++，是没有意义的。解决方法：erase(*iter)的返回值是下一个有效迭代器的值。 iter =cont.erase(iter);

链表型数据结构：对于list型的数据结构，使用了不连续分配的内存，删除运算使指向删除位置的迭代器失效，但是不会失效其他迭代器.解决办法两种，erase(*iter)会返回下一个有效迭代器的值，或者erase(iter++).

树形数据结构： 使用红黑树来存储数据，插入不会使得任何迭代器失效；删除运算使指向删除位置的迭代器失效，但是不会失效其他迭代器.erase迭代器只是被删元素的迭代器失效，但是返回值为void，所以要采用erase(iter++)的方式删除迭代器。

vector迭代器失效

1.当插入（push_back）一个元素后，end操作返回的迭代器肯定失效。

2.当插入(push_back)一个元素后，capacity返回值与没有插入元素之前相比有改变，则需要重新加载整个容器，此时begin和end操作返回的迭代器都会失效。

3.当进行删除操作（erase，pop_back）后，指向删除点的迭代器全部失效；指向删除点后面的元素的迭代器也将全部失效。

deque迭代器失效

1.在deque容器首部或者尾部插入元素不会使得任何迭代器失效。

2.在其首部或尾部删除元素则只会使指向被删除元素的迭代器失效。

3.在deque容器的任何其他位置的插入和删除操作将使指向该容器元素的所有迭代器失效。

list、set、map迭代器失效

删除时，指向该删除节点的迭代器失效

红黑树

红黑树是一种二叉查找树，二叉搜索树的规则是：任何节点的键值一定大于其左子树的每一个节点值，并小于右子树的每一个节点值。

红黑树的规则

1. 节点是红色或黑色。；

2. 根节点是黑色；

3. 每个叶节点（NILL节点，空节点）是黑色的；

4. 每个红色节点的两个子节点都是黑色（从每个叶子到根的所有路径上不能有两个连续的红色节点）；

5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点

由性质4可以推出：一条路径上不能有两个毗连的红色节点。最短的可能路径都是黑色节点，最长的可能路径是交替的红色和黑色节点。又根据性质5，所有最长的路径都有相同数目的黑色节点，这就表明了没有路径能多于任何其他路径的两倍长。

红黑树的插入

红黑树的删除

红黑树的旋转

算法

STL中的算法以迭代器为参数，通过作用于迭代器而达到处理容器数据的目的。所有的算法都前两个参数都是以对迭代器（iterator），通常称为first和last。事实上，算法所处理的迭代器范围为[first,last)，如果first==last则表明此范围为空。按照处理问题的不同，STL算法可分为四组，分别是：

1.不改变序列的算法(None-mutating sequence Algorithm)
2.改变序列的算法(Mutating sequence Algorithm)
3.排序以及相关算法(Sorting and related Algorithm)
4.常用数字算法(Generalized numeric Algorithm)

不改变序列算法

adjacent_find 查找两个相邻（Adjacent）的等价（Identical）元素
all_of (C++11) 检测在给定范围中是否所有元素都满足给定的条件
any_of (C++11) 检测在给定范围中是否存在元素满足给定条件
count 返回值等价于给定值的元素的个数
count_if 返回值满足给定条件的元素的个数
equal 返回两个范围是否相等
find 返回第一个值等价于给定值的元素
find_end 查找范围 A 中与范围 B 等价的子范围最后出现的位置
find_first_of 查找范围 A 中第一个与范围 B 中任一元素等价的元素的位置
find_if 返回第一个值满足给定条件的元素
find_if_not (C++11) 返回第一个值不满足给定条件的元素
for_each 对范围中的每个元素调用指定函数
mismatch 返回两个范围中第一个元素不等价的位置
none_of (C++11) 检测在给定范围中是否不存在元素满足给定的条件
search 在范围 A 中查找第一个与范围 B 等价的子范围的位置
search_n 在给定范围中查找第一个连续 n 个元素都等价于给定值的子范围的位置

修改序列算法

copy 将一个范围中的元素拷贝到新的位置处
copy_backward 将一个范围中的元素按逆序拷贝到新的位置处
copy_if (C++11) 将一个范围中满足给定条件的元素拷贝到新的位置处
copy_n (C++11) 拷贝 n 个元素到新的位置处
fill 将一个范围的元素赋值为给定值
fill_n 将某个位置开始的 n 个元素赋值为给定值
generate 将一个函数的执行结果保存到指定范围的元素中，用于批量赋值范围中的元素
generate_n 将一个函数的执行结果保存到指定位置开始的 n 个元素中
iter_swap 交换两个迭代器（Iterator）指向的元素
move (C++11) 将一个范围中的元素移动到新的位置处
move_backward (C++11) 将一个范围中的元素按逆序移动到新的位置处
random_shuffle 随机打乱指定范围中的元素的位置
remove 将一个范围中值等价于给定值的元素删除
remove_if 将一个范围中值满足给定条件的元素删除

remove_copy 拷贝一个范围的元素，将其中值等价于给定值的元素删除
remove_copy_if 拷贝一个范围的元素，将其中值满足给定条件的元素删除
replace 将一个范围中值等价于给定值的元素赋值为新的值
replace_copy 拷贝一个范围的元素，将其中值等价于给定值的元素赋值为新的值
replace_copy_if 拷贝一个范围的元素，将其中值满足给定条件的元素赋值为新的值
replace_if 将一个范围中值满足给定条件的元素赋值为新的值
reverse 反转排序指定范围中的元素
reverse_copy 拷贝指定范围的反转排序结果
rotate 循环移动指定范围中的元素
rotate_copy 拷贝指定范围的循环移动结果
shuffle (C++11) 用指定的随机数引擎随机打乱指定范围中的元素的位置
swap 交换两个对象的值
swap_ranges 交换两个范围的元素
transform 对指定范围中的每个元素调用某个函数以改变元素的值
unique 删除指定范围中的所有连续重复元素，仅仅留下每组等值元素中的第一个元素。
unique_copy 拷贝指定范围的唯一化（参考上述的 unique）结果

排序及相关算法

排序：
is_sorted (C++11) 检测指定范围是否已排序
is_sorted_until (C++11) 返回最大已排序子范围
nth_element 部份排序指定范围中的元素，使得范围按给定位置处的元素划分
partial_sort 部份排序
partial_sort_copy 拷贝部分排序的结果
sort 排序（快速排序）
stable_sort 稳定排序

划分：
is_partitioned (C++11) 检测某个范围是否按指定谓词（Predicate）划分过
partition 将某个范围划分为两组
partition_copy (C++11) 拷贝指定范围的划分结果
partition_point (C++11) 返回被划分范围的划分点
stable_partition 稳定划分，两组元素各维持相对顺序

二分查找：
binary_search 判断范围中是否存在值等价于给定值的元素
equal_range 返回范围中值等于给定值的元素组成的子范围
lower_bound 返回指向范围中第一个值大于或等于给定值的元素的迭代器
upper_bound 返回指向范围中第一个值大于给定值的元素的迭代器

合并：
includes 判断一个集合是否是另一个集合的子集
inplace_merge 就绪合并
merge 合并
set_difference 获得两个集合的差集
set_intersection 获得两个集合的交集
set_symmetric_difference 获得两个集合的对称差
set_union 获得两个集合的并集

堆：
is_heap 检测给定范围是否满足堆结构
is_heap_until (C++11) 检测给定范围中满足堆结构的最大子范围
make_heap 用给定范围构造出一个堆
pop_heap 从一个堆中删除最大的元素
push_heap 向堆中增加一个元素
sort_heap 将满足堆结构的范围排序

常用数字算法

is_permutation (C++11) 判断一个序列是否是另一个序列的一种排序
max 返回两个元素中值最大的元素
max_element 返回给定范围中值最大的元素
min 返回两个元素中值最小的元素
min_element 返回给定范围中值最小的元素
minmax (C++11) 返回两个元素中值最大及最小的元素
minmax_element (C++11) 返回给定范围中值最大及最小的元素

lexicographical_compare 比较两个序列的字典序
next_permutation 返回给定范围中的元素组成的下一个按字典序的排列
prev_permutation 返回给定范围中的元素组成的上一个按字典序的排列

空间配置器

为什么？

1.小块内存会带来内存碎片问题
如果任由STL中的容器自行通过malloc分配内存，那么频繁的分配和释放内存会导致堆中有很多的外部碎片。可能堆中的所有空闲空间之和很大，但当申请新的内存的请求到来时，没有足够大的连续内存可以分配，这将导致内存分配失败。因此这样会导致内存浪费。

2.小块内存的频繁申请释放会带来性能问题
开辟空间的时候，分配器需要时间去寻找空闲块，找到空闲块之后才能分配给用户。而如果分配器找不到足够大的空闲块可能还需要考虑处理加碎片现象（释放的小块空间没有合并），这时候需要花时间去合并已经释放了的内存空间块。
而且malloc在开辟内存空间的时候，还会附带附加的额外信息，因为系统需要靠多出来的额外信息管理内存。特别是区块越小，额外负担所占的比例就越大，更加显得浪费。

空间配置器分为两级：一级空间配置器（__malloc_alloc_template）和二级空间配置器(__default_alloc_template)。在STL中如果你申请的内存大于128个字节，那么直接调用一级空间配置器向内存申请内存，如果你申请的内存小于等于128个字节，将被认为是小内存，那么将会调用二级空间配置器直接去内存池中申请。

一级空间配置器，就是直接封装了malloc（）和free（）
二级空间配置器，根据具体情况采用不同的方法分配空间
小于128k，有一个内存池和一个自由链表来实现内存的管理
大于128k，调用一级空间配置器

默认情况下使用二级空间配置器，源码中有一个模板类——simple_alloc。其中有一个宏——__USE_MALLOC，如果这个宏被定义了，那么就是用一级空间适配器。如果没被定义，那么就是用二级空间适配器。

空间配置器的使用

1.源码中存在一个枚举变量__MAX_BYTES（128）来区分大内存块和小内存块，凡是大于__MAX_BYTES的内存块都是属于一级空间配置器的管理范畴。

2.二级空间配置器的底层是由一个free list（哈希桶）来管理的。有一个枚举变量__ALIGN的值（8）就是表示哈希桶划分的间隔。

3.二级空间配置器中会有对应算法来计算出最合适的内存块和内存块个数，然后查询哈希桶是否有对应的内存块或者说内存块是否足够，如果没有，向系统询问调整free list。

```
if (result == 0) {
    // 沒找到可用的 free list，準備重新填充 free list
    void *r = refill(ROUND_UP(n));      // 下節詳述
    return r;
}
```

4.在allocate()函数调用时发现free list中没有空间是，便会呼叫refill()函数重新填充free list。——去找内存池要。

```
int nobjs = 20;
// 呼叫 chunk_alloc()，嘗試取得 nobjs 個區塊做爲 free list 的新節點。
// 注意參數 nobjs 是 pass by reference。
char * chunk = chunk_alloc(n, nobjs); // 下節詳述
```

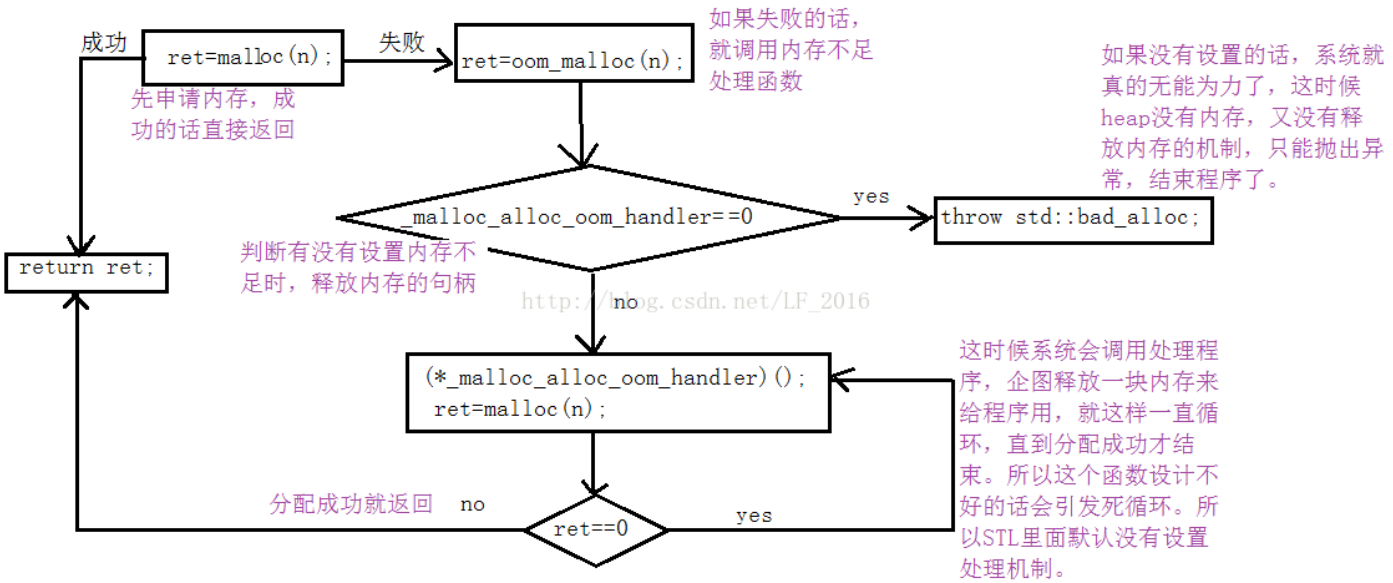
新的空间来自哪？——内存池。这部分工作交给chunk_alloc()。当然也有可能内存池（memory pool）中内存也满足不了要求。
类中利用两个静态指针 start_free 和 end_free来标识内存池。
chunk_alloc()会利用end_free - start_free来计算判断内存池的内存量：
如果存在内存但是不够，就先把能给的给出去。
如果存在内存而且够了，直接给不废话。
如果内存是一点都没了，只能重新去heap中malloc。

5.山穷水尽——如果真的二级空间配置器一点内存都分配不出来了，那么只能从一级空间配置器中来寻找。
为什么二级都没有而一级会有呢？
因为一级中有空间不足时的应对措施，所以就算是失败，也是一级空间配置器来报的错。

二级空间配置器的释放？
首先先判断是否大于128，如果大于则用一级空间配置器的释放，如果小于则挂回free list。

一级空间配置器

一级空间配置器只是简单的封装了一下malloc和free实现的。在allocate函数中如果通过malloc申请内存失败（失败返回0）就改用oom_malloc(size_t n)函数尝试分配内存，如果oom发现没有指定new-handler函数的话，那就直接调用__THROW_BAD_ALLOC，丢出bad_alloc或是直接通过exit(1)中止程序。

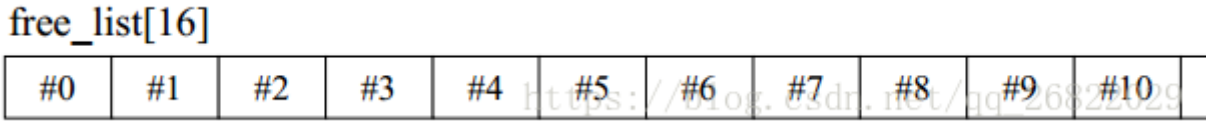


二级空间配置器

二级空间配置器使用内存池+自由链表的形式避免了小块内存带来的碎片化，提高了分配的效率，提高了利用率。SGI的做法是先判断要开辟的大小是不是大于128，如果大于128则就认为是一块大块内存，调用一级空间配置器直接分配。否则的话就通过内存池来分配，假设要分配8个字节大小的空间，那么他就会去内存池中分配多个8个字节大小的内存块，将多余的挂在自由链表上，下一次再需要8个字节时就去自由链表上取就可以了，如果回收这8个字节的话，直接将它挂在自由链表上就可以了。
如果不大于就将任何小额区块的内存__n上调至8的倍数（即使你请求1个字节，那也会申请8个字节）。这是因为在二级空间配置器中内存是通过free_list管理的。

*注意这里出现了关键字voliate：确保本条指令不会因为编译器的优化而省略，而且要求每次从内存中直接读取值

free_list如下图所示，为了便于管理，二级空间配置器在分配的时候都是以8的倍数对齐。因此free_list含有16个结点，分别管理大小为8，16，24，32，40，48，56，64，72，80，88，96，104，112，120，128字节大小的内存。

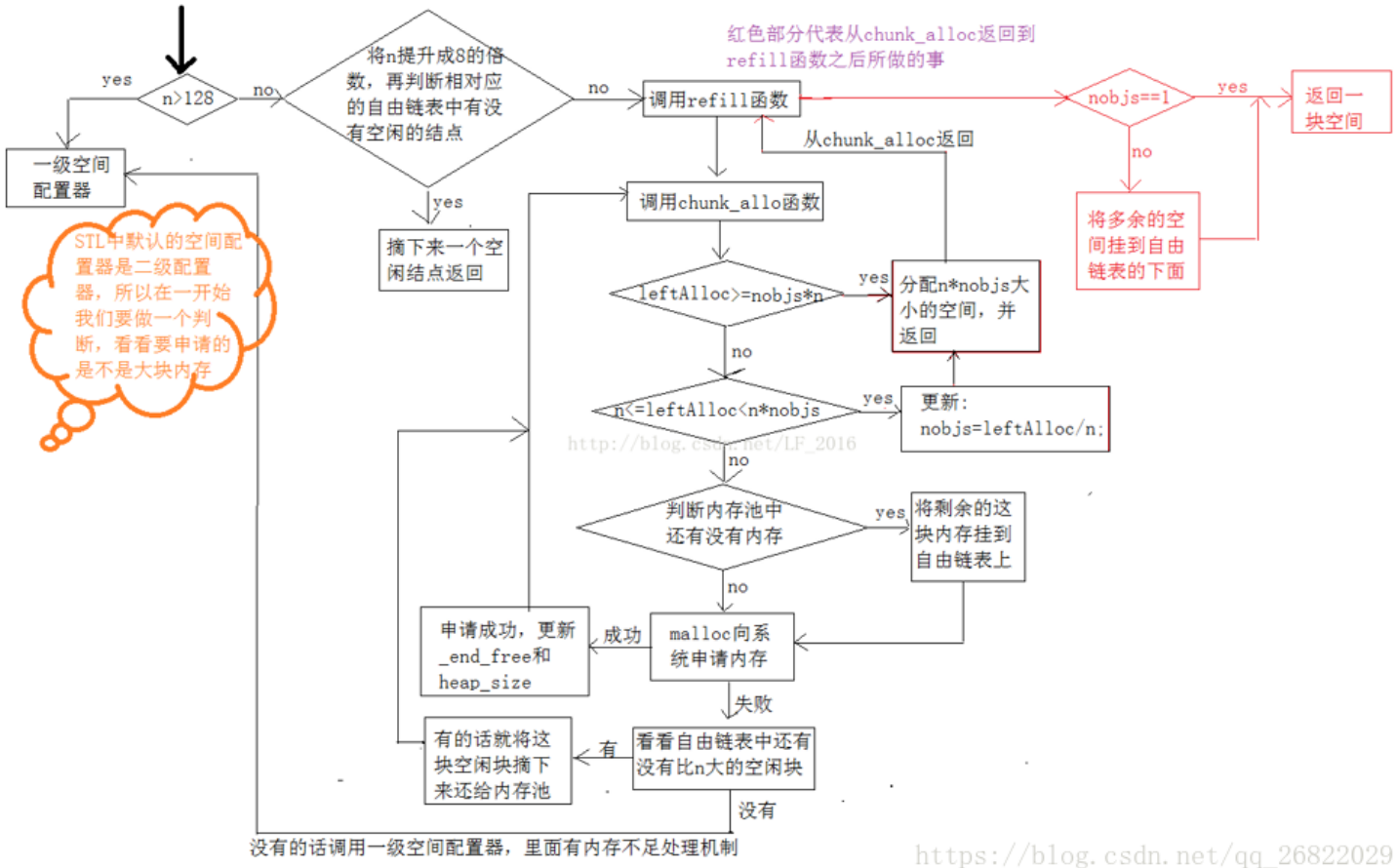


二级空间配置器的逻辑步骤：
假如现在申请n个字节：
1、判断n是否大于128，如果大于128则直接调用一级空间配置器。如果不大于，则将n上调至8的倍数处，然后再去自由链表中相应的结点下面找，如果该结点下面挂有未使用的内存，则摘下来直接返回这块空间的地址。否则的话我们就要调用refill(size_t n)函数去内存池中申请。
2、向内存池申请的时候可以多申请几个，STL默认一次申请nobjs=20个，将多余的挂在自由链表上，这样能够提高效率。
进入refill函数后，先调chunk_alloc(size_t n,size_t& nobjs)函数去内存池中申请，如果申请成功的话，再回到refill函数。
这时候就有两种情况，如果nobjs=1的话则表示内存池只够分配一个，这时候只需要返回这个地址就可以了。否则就表示nobjs大于1，则将多余的内存块挂到自由链表上。
如果chunk_alloc失败的话，在他内部有处理机制。
3、进入chunk_alloc(size_t n,size_t& nobjs)向内存池申请空间的话有三种情况：
3.1、内存池剩余的空间足够nobjs*n这么大的空间，则直接分配好返回就可以了。*
*3.2、内存池剩余的空间leftAlloc的范围是n<=leftAlloc<nobjs*n,则这时候就分配nobjs=(leftAlloc)/n这么多个的空间返回。
3.3、内存池中剩余的空间连一个n都不够了，这时候就要向heap申请内存，不过在申请之前先要将内存池中剩余的内存挂到自由链表上，之后再向heap申请。
3.3.1、如果申请成功的话，则就再调一次chunk_alloc重新分配。
3.3.2、如果不成功的话，这时候再去自由链表中看看有没有比n大的空间，如果有就将这块空间还给

内存池，然后再调一次chunk_alloc重新分配。

3.3.3、如果没有的话，则就调用一级空间配置器分配，看看内存不足处理机制能否处理。

申请的流程图：



空间配置器的其他问题：

1、在空间配置器中所有的函数和变量都是静态的，所以他们在程序结束的时候才会被释放发。二级空间配置器中没有将申请的内存还给操作系统，只是将他们挂在自由链表上。所以说只有当你的程序结束了之后才会将开辟的内存还给操作系统。

2、由于它没有将内存还给操作系统，所以就会出现二种极端的情况。

2.1、假如我不断的开辟小块内存，最后将整个heap上的内存都挂在了自由链表上，但是都没有用这些空间，再想要开辟一个大块内存的话会开辟失败。

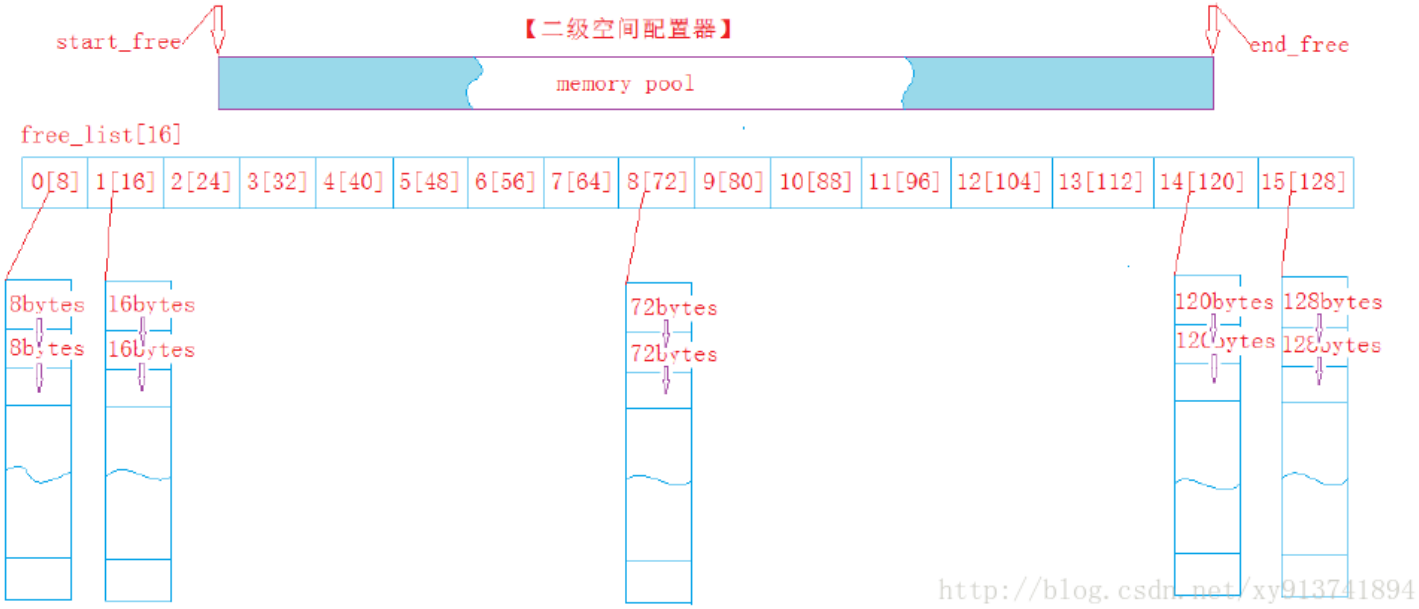
2.2、再比如我不断的开辟char,最后将整个heap内存全挂在了自由链表的第一个结点的后面，这时候我再想开辟一个16个字节的内存，也会失败。

或许我比较杞人忧天吧，总的来说上面的情况只是小概率情况。如果非得想办法解决的话，我想的是：针对2.1我们可以引入释放二级空间配置器的方法，但是这个释放比较麻烦。针对2.2我们可以合并自由链表上的连续的小的内存块。

3、二级空间配置器会造成内碎片问题，极端的情况下一直申请char，则就会浪费7/8的空间。但是整体来说，空间配置器的性能还是蛮高的。

内存池技术

如图所示，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。



当自由链表中没有对应的内存块，系统会执行以下策略：
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n块，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。
1、如果内存池大于 nobjs * n，那么直接从内存池中取出
2、如果内存池小于nobjs * n，但是比一块大小n要大，那么此时将内存最大可分配的块数给自由链表，并且更新nobjs为最大分配块数x （x < nobjs）
3、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申请空间，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

仿函数

新名称叫函数对象。
仿函数(functor)，就是使一个类的使用看上去象一个函数。其实现就是类中实现一个operator()，这个类就有了类似函数的行为，就是一个仿函数类了。实际上仿函数对象仅仅占用1字节，因为内部没有数据成员，仅仅是一个重载的方法而已。
仿函数产生的原因：
由于函数指针毕竟不能满足STL对抽象对象的需求，也不能满足软件积木的需求——函数指针无法和STL其它组件（如配接器adapter）搭配使用，产生更灵活的变化。

```
struct MyPlus{
    int operator()(const int &a , const int &b) const{
        return a + b;
    }
};

int main()
{
    MyPlus a;
    cout << MyPlus()(1,2) << endl;//1、通过产生临时对象调用重载运算符
    cout << a.operator()(1,2) << endl;//2、通过对象显示调用重载运算符
    cout << a(1,2) << endl;//3、通过对象类似函数调用  隐示地调用重载运算符
    return 0;
}
```