STL容器的实现无疑伴随着内存的管理，内存分配算法实现的好坏对STL特别是容器的效率有着很大的影响。最底层的C运行库中的malloc系列函数对内存管理的单位为字节，这对于适用于所有类型的容器而言非常不好，因此，需要实现针对STL的内存分配器。对于此，我们可以使用new和delete这些内置的C++操作符，但是，这些操作符会影响STL的效率，因此，SGI的实现版本中，并没有使用new和delete，而是使用了更加底层的malloc系列，对其进行了封装。实现了自己的内存管理器。

为了后边分析代码时容易描述，我们做好如下定义：

一级内存分配器：以字节为单位的内存分配器，例如malloc(size_t n)，标识分配n个字节。

二级内存分配器：以某种类型的对象大小为单位的内存分配器，例如当类型为整数（4字节）时，alloc(size_t n)表示分配存放n个整数的内存块，那么大小为4*n字节。这一点可以用C++的模版来实现泛型。

现在对STL的内存分配器进行分析：

（1）一级内存分配器

STL的一级内存分配器就是简单的对malloc系列函数的分装。不同之处就是针对maolloc系列函数调用失败的情况进行了处理（调用了一个函数），该函数由用户实现。可以进行设置。一级内存分配器对外提供了四个接口：

allocate：分配n个字节的内存块。

deallocate：释放由allocate分配的内存块。

reallocate：分配新的内存块，并将原来内存块中的数据拷贝到新内存块中。

set_malloc_handler：设置当allocate和reallocate实现时调用更加底层的函数分配内存失败后时需要调用的函数。

该类的声明如下：

```
template <int inst>
class __malloc_alloc_template
{
private:
    //调用C库malloc和realloc失败之后、调用的函数。这两个函数会调用函数__malloc_alloc_oom_handler
    //处理了内存分配失败的情况后，会重新分配内存。如果__malloc_alloc_oom_handler为0，则抛出异常。
    static void *oom_malloc(size_t);
    static void *oom_realloc(void *, size_t);
    //指向内存分配失败时要调用的函数
    static void (* __malloc_alloc_oom_handler)();

public:
    //分配n个字节的函数。使用了标准C库的malloc函数
    static void * allocate(size_t n)
    {
        void *result = malloc(n);
        if (0 == result)
            result = oom_malloc(n);
        return result;
    }
    //释放由allocate分配的内存，该函数使用了free，由此可以看到，oom_malloc函数也会返回由malloc分配的内存。
    static void deallocate(void *p, size_t /* n */)
    {
        free(p);
    }

    static void * reallocate(void *p, size_t /* old_sz */, size_t new_sz)
    {
        void * result = realloc(p, new_sz);
        if (0 == result)
        {
            result = oom_realloc(p, new_sz);
        }
        return result;
    }

    //操作符的优先级 函数调用() > *(解引用),因此set_malloc_handler为一个函数
    //set_malloc_handler返回值为一个函数指针，该指针指向一个参数为空的函数，
    //set_malloc_handler的参数为一个函数指针，该指针指向一个参数为空的函数。
    static void (* set_malloc_handler(void (*f)()))()
    {
        void (* old)() = __malloc_alloc_oom_handler;
        __malloc_alloc_oom_handler = f;
        return(old);
    }

};
```

接下来就是剩下的成员函数的实现。

```
template <int inst>
void (* __malloc_alloc_template<inst>::__malloc_alloc_oom_handler)() = 0;
```

如上图：可见默认下，当内存分配失败（例如内存不足时）时该采取的动作交由用户决定。换句话说：“内存不足处理例程”的设计是客户（使用该类者）的责任。
接下来看成员函数oom_malloc和oom_realloc的实现。这两个函数用于处理内存分配失败时的容错逻辑。

```
template <int inst>
void * __malloc_alloc_template<inst>::oom_malloc(size_t n)
{
    void (* my_malloc_handler)();
    void *result;
    for (;;)
    {
        my_malloc_handler = __malloc_alloc_oom_handler;
        if (0 == my_malloc_handler)
        {
            __THROW_BAD_ALLOC;
        }
        (*my_malloc_handler)();
        result = malloc(n);
        if (result)
        {
            return(result);
        }
    }
}
```

如上所示：该函数首先定义一个函数指针，然后进入无限循环；如果用户没有设置自己的内存错误处理函数，则直接抛出异常。否则一直调用内存错误处理函数并分配内存，直到分配成功。

```
template <int inst>
void * __malloc_alloc_template<inst>::oom_realloc(void *p, size_t n)
{
    void (* my_malloc_handler)();
    void *result;

    for (;;)
    {
        my_malloc_handler = __malloc_alloc_oom_handler;
        if (0 == my_malloc_handler)
        {
            __THROW_BAD_ALLOC;
        }
        (*my_malloc_handler)();
        result = realloc(p, n);
        if (result)
        {
            return(result);
        }
    }
}
typedef __malloc_alloc_template<0> malloc_alloc;
```

如上所示：oom_realloc与oom_malloc的实现逻辑一样。
由上边的实现，我们也看到模版参数inst也没有用到。
（2）二级内存分配器
二级内存分配器也很简单，但是要能够分配各种大小的对象，为此将要分配的类型应该应模版参数传递，因此需要类模版，定义如下：

```
template<class T, class Alloc>
class simple_alloc {
public:
    static T *allocate(size_t n)
    {
        return 0 == n? 0 : (T*) Alloc::allocate(n * sizeof (T));
    }
```

```
    static T *allocate(void)
    {
        return (T*) Alloc::allocate(sizeof (T));
    }
    static void deallocate(T *p, size_t n)
    {
        if (0 != n)
            Alloc::deallocate(p, n * sizeof (T));
    }
    static void deallocate(T *p)
    {
        Alloc::deallocate(p, sizeof (T));
    }
};
```

可以看到，十分简单，就是简单的调用Alloc的函数（这些函数为一级内存分配器-----以字节为单位）。Alloc该模版参数可以为上边的malloc_alloc类。

（3）复杂的一级内存分配器

为了优化内存的分配速度，SGI的STL并未简单的使用上述的malloc_alloc作为一级内存分配器，而是自己实现了更加复杂的内存分配器，为了加快速度，实现了内存池的思想。该类模版的定义如下：

```
#ifdef __SUNPRO_CC
enum {__ALIGN = 8};
enum {__MAX_BYTES = 128};
enum {__NFREELISTS = __MAX_BYTES/__ALIGN};
#endif

template <bool threads, int inst>
class __default_alloc_template
{
private:
# ifndef __SUNPRO_CC
    enum {__ALIGN = 8};             //8字节对齐
    enum {__MAX_BYTES = 128};       //超过128字节，直接使用malloc和free。
    enum {__NFREELISTS = __MAX_BYTES/__ALIGN};
# endif
    //将bytes上调到8的倍数
    static size_t ROUND_UP(size_t bytes)
    {
        return (((bytes) + __ALIGN-1) & ~(__ALIGN - 1));
    }
__PRIVATE:
    union obj
    {
        union obj * free_list_link;
        char client_data[1];    /* The client sees this.        */
    };
private:
    //维护的自由链表数组，可以看作简单的内存池，维护的内存块链表的种类有16种（128/8）
# ifdef __SUNPRO_CC
    static obj * __VOLATILE free_list[];
    // Specifying a size results in duplicate def for 4.1
# else
    static obj * __VOLATILE free_list[__NFREELISTS];
# endif
    //依据要申请的区块的大小，计算出free_list数组的索引值
    static  size_t FREELIST_INDEX(size_t bytes)
    {
        return (((bytes) + __ALIGN-1)/__ALIGN - 1);
    }
    //返回一个大小为n的对象，同时可能加入大小为n的其他内存块到自由链表free_list中。
    static void *refill(size_t n);
    //分配一大块内存，该块内存可容纳nobjs个大小为size的对象。
    //如果难以分配内存，那么nobjs可能会被减小。
    static char *chunk_alloc(size_t size, int &nobjs);

    static char *start_free;
    static char *end_free;
    static size_t heap_size;
    ////////////////////////////////如果支持多线程//////////////////////////
# ifdef __STL_SGI_THREADS
```

```
51.     static volatile unsigned long __node_allocator_lock;
52.     static void __lock(volatile unsigned long *);
53.     static inline void __unlock(volatile unsigned long *);
54. # endif
55. 
56. # ifdef _PTHREADS
57.     static pthread_mutex_t __node_allocator_lock;
58. # endif
59. 
60. # ifdef __STL_WIN32THREADS
61.     static CRITICAL_SECTION __node_allocator_lock;
62.     static bool __node_allocator_lock_initialized;
63. 
64. public:
65.     __default_alloc_template()
66.     {
67.         // This assumes the first constructor is called before threads are started.
68.         if (!__node_allocator_lock_initialized)
69.         {
70.             InitializeCriticalSection(&__node_allocator_lock);
71.             __node_allocator_lock_initialized = true;
72.         }
73.     }
74. private:
75. # endif
76.     class lock
77.     {
78.     public:
79.         lock() { __NODE_ALLOCATOR_LOCK; }
80.         ~lock() { __NODE_ALLOCATOR_UNLOCK; }
81.     };
82.     friend class lock;
83. public:
84. 
85.     /*
86.     分配一个大小为n字节的内存块
87.     分配逻辑如下：
88.     先判断n是否大于__MAX_BYTES(128字节)，如果是，则直接用maolloc分配；如果不大于128字节，
89.     则从维护的内存缓冲池链表中取一块内存，当然了，有可能内存缓冲池中没有对      应大小的内存
90.     块，此时就需要调用函数refill，该函数返回对应的内存块，并可能增加新的块到内存缓冲池中（
91.     类似缓存没有命中的思维）。
92.     */
93.     static void * allocate(size_t n)
94.     {
95.         obj * __VOLATILE * my_free_list;
96.         obj * __RESTRICT result;
97.         //如果是很大一块内存（大于128字节，则直接使用malloc分配）
98.         if (n > (size_t) __MAX_BYTES)
99.         {
100.             return(malloc_alloc::allocate(n));
101.         }
102.         my_free_list = free_list + FREELIST_INDEX(n);
103. #       ifndef _NOTHREADS
104.         lock lock_instance;
105. #       endif
106.         result = *my_free_list;
107.         if (result == 0)
108.         {
109.             void *r = refill(ROUND_UP(n));
110.             return r;
111.         }
112.         *my_free_list = result -> free_list_link;
113.         return (result);
114.     };
115. 
116.     /*
117.     释放一个内存块，如果释放的内存块大于128，则直接调用free函数，否则：
118.     直接将内存放到内存缓冲池中。以备将来使用。这里要特别注意:对于块小于128
119.     的内存来说，不会调用free函数，而是放到自己维护的自由链表中，这样的话，内
120.     存的使用岂不是会随着使用时间的增加一直增加？？
121.     */
122.     static void deallocate(void *p, size_t n)
```

```
123.     {
124.         obj *q = (obj *)p;
125.         obj * __VOLATILE * my_free_list;
126.
127.         if (n > (size_t) __MAX_BYTES)
128.         {
129.             malloc_alloc::deallocate(p, n);
130.             return;
131.         }
132.         my_free_list = free_list + FREELIST_INDEX(n);
133. #       ifndef _NOTHREADS
134.         lock lock_instance;
135. #       endif
136.         //将内存块放入空前内存块链表中
137.         q -> free_list_link = *my_free_list;
138.         *my_free_list = q;
139.         // lock is released here
140.     }
141.
142.     static void * reallocate(void *p, size_t old_sz, size_t new_sz);
143.
144. } ;
```

该类的成员函数的实现有如下几个特点：

1、分配的内存块的大小至少为8字节，也即8字节对其；

2、当申请的内存块大小超过128字节时直接使用malloc系列函数；

3、内存块小于128字节时，该分配器会用自己维护的进程全局的内存池中的内存来分配；

4、为了效率，该内存分配器会维护一个自由链表数组（该数组大小为128/16，表示链表中内存块的大小）。

接下来详细分析该分配器的成员函数的实现。

该分配器对外的接口只有如下三个：

allocate：分配一个大小为n字节的内存块；

deallocate：释放由allocate分配的内存块；

reallocate：类似于C语言中realloc函数的语义。

初始状态下：链表数组中的空闲链表都是空的，在内存的逐渐分配中，会不断增加链表中的空闲内存块。初始状态如下：

free_list：

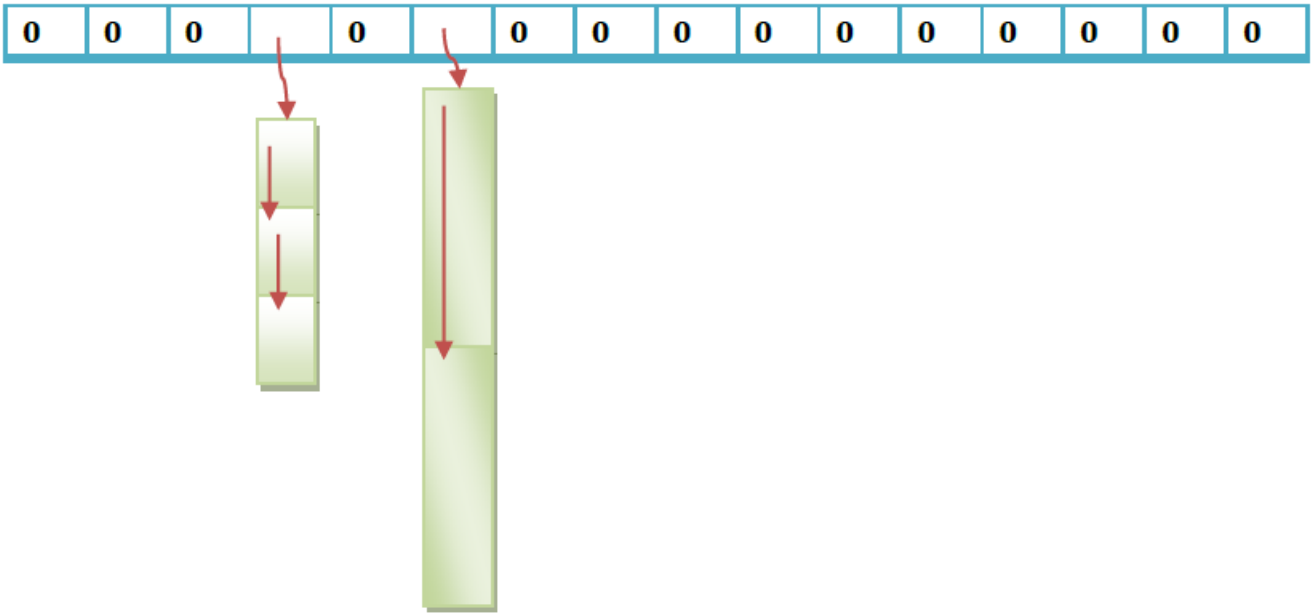
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

allocate函数的实现：

```
1.     static void * allocate(size_t n)
2.     {
3.         obj * __VOLATILE * my_free_list;
4.         obj * __RESTRICT result;
5.         //如果是很大一块内存（大于128字节，则直接使用malloc分配）
6.         if (n > (size_t) __MAX_BYTES)
7.         {
8.             return(malloc_alloc::allocate(n));
9.         }
10.         my_free_list = free_list + FREELIST_INDEX(n);
11. #       ifndef _NOTHREADS
12.         lock lock_instance;
13. #       endif
14.         result = *my_free_list;
15.         if (result == 0)
16.         {
17.             void *r = refill(ROUND_UP(n));
18.             return r;
19.         }
20.         *my_free_list = result -> free_list_link;
21.         return (result);
22.     };
```

该函数首次分配不大于128字节的内存块时，由于free_list这个链表数组中的链表为空，因此result = *my_free_list;语句一定为0，这时将调用函数refill，函数refill返回一个大小为n的内存块，同时可能加入大小为n的其他内存块到自由链表free_list中。可以看到，调用函数refill之前已经调用过了ROUND_UP(n)，即已经将n上调到8的倍数。具体如何扩展的自由链表，在后边进行详细分析。

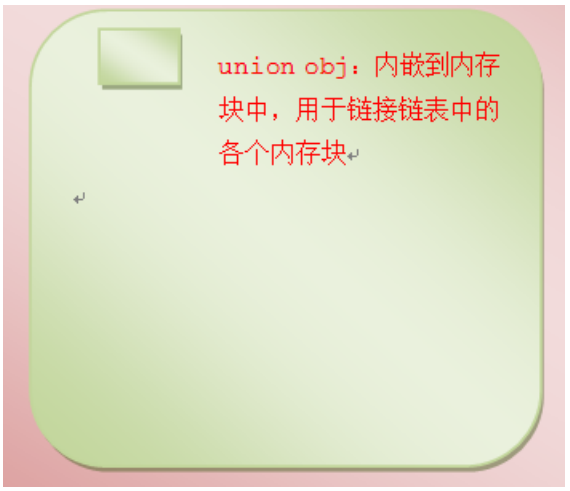
如果是运行过程中，则链表数组的状态如下：



此时如果申请大小为32字节的内存块，则会走到如下逻辑：

```
result = *my_free_list;
*my_free_list = result -> free_list_link;
```

即：将链表数组的某个链表的第一个节点返回。并让链表数组对应的元素指向该链表的下一个元素作为节点。
链表中每个内存节点的构成如下：



deallocate函数的实现：
该函数的实现相对简单，如果要释放的内存块大于128字节，则直接使用malloc系列的函数将其释放给C运行库维护的自由链表（是否释放给操作系统，这个就由C运行库来决定了），否则，将内存块放到由该分配器维护的自由链表数组中。
由此可见，对于内存块小于128的内存，并未真正释放给C运行库，更不会释放给操作系统，这当然是为了下次分配的时候更加快速（直接定位数组坐标，并取出内存块即可）。**这让会不会有内存问题呢，比如服务器程序运行了很久？？？？？这就要看怎么使用STL的容器了。后边分析完STL的容器再来分析这个问题。**
reallocate函数的实现
该函数实现也很简单：首先进行特殊条件（是否需要重新分配内存，并释放老内存）的判断，如果需要重新分配，就重新分配新内存，拷贝老内存的数据到新内存中，并释放老内存。具体代码如下：

```
template <bool threads, int inst>
void* __default_alloc_template<threads, inst>::reallocate(void *p,size_t old_sz,size_t new_sz)
{
    void * result;
    size_t copy_sz;

    if (old_sz > (size_t) __MAX_BYTES && new_sz > (size_t) __MAX_BYTES)
```

```
    {
        return(realloc(p, new_sz));
    }
    if (ROUND_UP(old_sz) == ROUND_UP(new_sz))
    {
        return(p);
    }
    result = allocate(new_sz);
    copy_sz = new_sz > old_sz? old_sz : new_sz;
    memcpy(result, p, copy_sz);    //采用底层的内存拷贝函数，为了提高效率
    deallocate(p, old_sz);
    return(result);
}
```

reallocate函数在使用时必须正确传递老内存块的大小。不然会出问题。

分析完对外提供的三个函数之外，就要分析支撑这些函数运行的内部函数了。还是由客户端的使用去驱动分析。

由之前的分析，我们已经知道：当没有空闲的内存时会调用refill来分配内存。现在就来看看该函数干了些什么。函数refill的正确使用建立在如下两个条件上：

1、多线程环境下，该函数假设调用方已经加了必要的互斥锁，我们的allocate函数在调用它时就是加了锁的，并且在调用完之后解的锁。

2、该函数假设函数参数n已经被调整过，在此就是8字节对其，我们的allocate函数在调用它时是做了这个处理的。

现在来分析该函数实现如下：

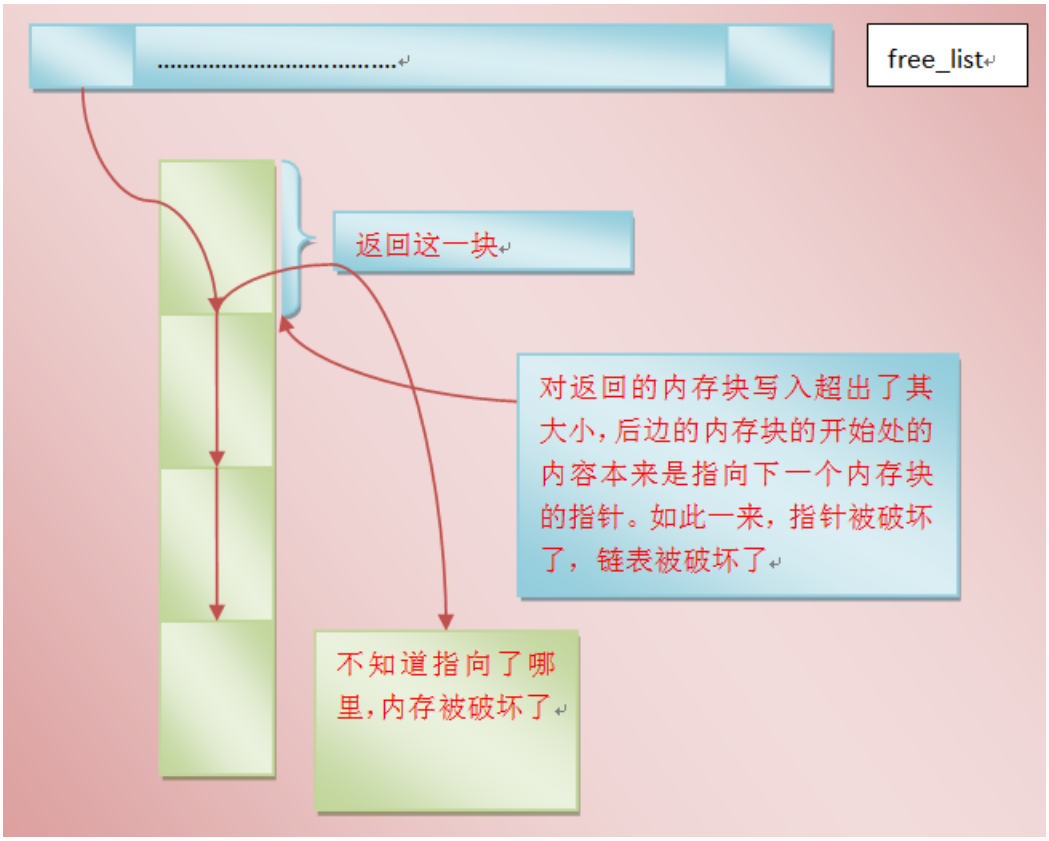
```
template <bool threads, int inst>
void* __default_alloc_template<threads, inst>::refill(size_t n)
{
    //内存块大小n已经是经过调整后的了(也就是说已经是8的整数倍了)
    int nobjs = 20;
    char * chunk = chunk_alloc(n, nobjs);
    obj * __VOLATILE * my_free_list;
    obj * result;
    obj * current_obj, * next_obj;
    int i;

    if (1 == nobjs)
    {
        return(chunk);
    }
    my_free_list = free_list + FREELIST_INDEX(n);

    /* Build free list in chunk */
    result = (obj *)chunk;
    *my_free_list = next_obj = (obj *)(chunk + n);
    for (i = 1; ; i++)
    {
        current_obj = next_obj;
        next_obj = (obj *)((char *)next_obj + n);
        if (nobjs - 1 == i)
        {
            current_obj -> free_list_link = 0;
            break;
        }
        else
        {
            current_obj -> free_list_link = next_obj;
        }
    }
    return(result);
}
```

该函数具体实现逻辑如下：

首先分配可以容纳20个大小为n的内存块的连续（从该函数后边建立自由链表的过程可以看到是连续的大块内存）大块内存，这里使用了函数chunk_alloc函数，chunk_alloc函数也假设n已经对其过了，该函数可能会比要求的内存小。取出第一块内存后作为结果返回给调用者。剩下的内存用于构建自由链表。这里我们可以看到：对于返回的内存块，我们对其写入的时候一定不能超出其大小，否则自由链表将会遭到破坏。如下图：



如上图：对于返回的内存块，我们对其写入的时候如果超出了其大小，那么自由链表第一个节点后的节点将断掉。自由链表被破坏了。因此对其写入的时候一定不能超出其大小。

由上看到函数refill也没有对内存进行真正的管理，真正的扩容由chunk_alloc来进行，chunk_alloc才是最为核心的对内存（不是操作系统层的内存、也不是C运行库的、而是本内存分配器维护的内存）管理的函数，该函数要详细分析，该函数实现如下：

```
template <bool threads, int inst>
char* __default_alloc_template<threads, inst>::chunk_alloc(size_t size, int& nobjs)
{
    //内存块大小n已经是经过调整后的了(也就是说已经是8的整数倍了)
    char * result;
    size_t total_bytes = size * nobjs;
    size_t bytes_left = end_free - start_free;

    if (bytes_left >= total_bytes)
    {
        result = start_free;
        start_free += total_bytes;
        return(result);
    }
    else if (bytes_left >= size)
    {
        nobjs = bytes_left/size;
        total_bytes = size * nobjs;
        result = start_free;
        start_free += total_bytes;
        return(result);
    }
    else
    {
        size_t bytes_to_get = 2 * total_bytes + ROUND_UP(heap_size >> 4);
        // Try to make use of the left-over piece.将内存链接到内存池中
        if (bytes_left > 0)
        {
            obj * __VOLATILE * my_free_list = free_list + FREELIST_INDEX(bytes_left);
            ((obj *)start_free) -> free_list_link = *my_free_list;
            *my_free_list = (obj *)start_free;
        }
        start_free = (char *)malloc(bytes_to_get);
        if (0 == start_free)
        {
```

```
            int i;
            obj * __VOLATILE * my_free_list, *p;
            for (i = size; i <= __MAX_BYTES; i += __ALIGN)
            {
                my_free_list = free_list + FREELIST_INDEX(i);
                p = *my_free_list;
                if (0 != p)
                {
                    *my_free_list = p -> free_list_link;
                    start_free = (char *)p;
                    end_free = start_free + i;
                    return(chunk_alloc(size, nobjs));
                }
            }
            end_free = 0;   // In case of exception.
            start_free = (char *)malloc_alloc::allocate(bytes_to_get);
        }
        heap_size += bytes_to_get;
        end_free = start_free + bytes_to_get;
        return(chunk_alloc(size, nobjs));
    }
}
```

该函数的使用也基于两点假设：

1、多线程环境下，该函数假设调用方已经加了必要的互斥锁，我们的refill函数在调用它时就是加了锁的，并且在调用完之后解的锁。

2、该函数假设函数参数n已经被调整过，在此就是8字节对其，我们的refill函数在调用它时是做了这个处理的。

满足这两点假设才能正确使用。

接下来详细分析该函数的实现逻辑。

该函数为了避免malloc系列的C函数内存碎片不会太多，总会申请大块内存，来维护一个内存缓冲池，来共空闲链表使用。

前面看到类定义中有如下几个成员：

```
    static char *start_free;
    static char *end_free;
    static size_t heap_size;
```

这几个成员就用来维护该分配器维护的内存缓冲池。

该函数首先判断end_free - start_free是否大于或等于调用者要求的内存块大小即：total_bytes；

如果大于等于total_bytes，则返回其中的内存给调用者，并修改start_free；

如果小于total_bytes但是大于等于单个内存块的大小，也就是refill中的n，也就是allocate中的n，则返回(end_free - start_free)/n块大小为n的内存，当然了，这些内存是连续的。注意该函数的参数nobjs是个引用，函数的调用者是知道chunk_alloc分配的这块内存可以容纳多少个大小为n的内存块的。

如果以上两个条件都不满足，则会用malloc向C运行库申请内存。维护内存池，并返回。

由上可以看到：STL的内存层次结构图如上：

接着分析第三种情况：此时空闲链表中没有空闲内存、而且要分配的内存块小于等于128字节，而且该分配器维护的内存缓冲池中也没有内存了，该考虑向C运行库申请了。具体代码如下：

```
        size_t bytes_to_get = 2 * total_bytes + ROUND_UP(heap_size >> 4);
        if (bytes_left > 0)
        {
            obj * __VOLATILE * my_free_list = free_list + FREELIST_INDEX(bytes_left);
            ((obj *)start_free) -> free_list_link = *my_free_list;
            *my_free_list = (obj *)start_free;
        }
        start_free = (char *)malloc(bytes_to_get);
        if (0 == start_free)
        {
            int i;
            obj * __VOLATILE * my_free_list, *p;
            for (i = size; i <= __MAX_BYTES; i += __ALIGN)
            {
                my_free_list = free_list + FREELIST_INDEX(i);
                p = *my_free_list;
                if (0 != p)
                {
                    *my_free_list = p -> free_list_link;
                    start_free = (char *)p;
                    end_free = start_free + i;
                    return(chunk_alloc(size, nobjs));
                }
            }
            end_free = 0;   // In case of exception.
            start_free = (char *)malloc_alloc::allocate(bytes_to_get);
        }
        heap_size += bytes_to_get;
        end_free = start_free + bytes_to_get;
        return(chunk_alloc(size, nobjs));
```

作者：许富博