STL源码剖析读书笔记

The Annotated STL Sources(using SGI STL) 侯捷

第1章 STL 概论与版本简介

跳过吧

第2章 空间配置器(allocator)

在STL中，容器的定义中都带有一个模板参数，如vector

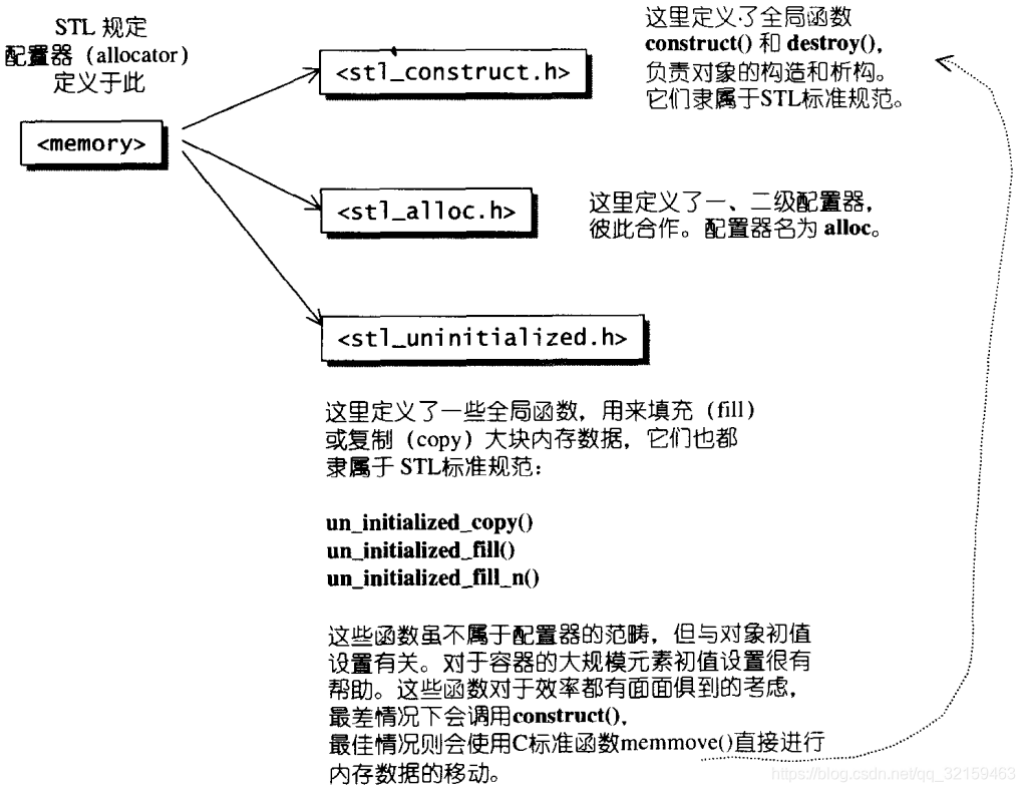
```
template <class T, class Alloc=alloc>     //默认使用alloc空间配置器
class vector{...}
```

SGI标准空间配置器， std::allocator
SGI中定义了一个符合部分标准、名为allocator的配置器，但从不从不使用它。主要效率不佳，只是简单包装了下::*operator new* 和 ::*operator delete*

```
template <class T>
inline T* allocate(ptrdiff_t, T*){
    set_new_handler(0);
    T* tmp = (T*)(::operator new((size_T)(size*sizeof(T))));
    if(tmp==0){
        cerr<<"out of memory"<<endl;
        exit(1);
    }
    return tmp;
}

template <class T>
inline void deallocate(T* buffer){
    ::operator delete(buffer);
}

template <class T>
class allocator{
    //内部allocate和deallocate的实现采用上面的模板函数
};

```

SGI特殊的空间配置器， std::alloc

```
class Foo {...};
Foo* pf = new FOO();     //先配置空间、后构造对象
delete pf;               //先对象析构、后释放空间
```



1. 构造和析构 construct()和destroy()

```
template <class T1, class T2>
inline void construct(T1* p, const T2& value){
    new (p) T1(value);       //placement new; 调用T1::T1(value);
}
// 第一个版本，接受一个指针
template <class T>
inline void destory(T* pointer){
    pointer->~T();
}
// 第二个版本，接受了个迭代器， 根据元素的类型进行删除
template <class ForwardIterator, class T>
inline void destory(ForwardIterator first, ForwardIterator last, T*){
    _destory_aux(first, last, value_type(first));
}

//数值类型有non-trivial destructor
template<class ForwardIterator>
inline void _destroy_aux(ForwardIterator first, ForwardIterator last,
__false_type){
    for(; first<last; ++first)
        destroy(&*first);
}
//数值类型有trival destructor
template<class ForwardIterator>
inline void _destroy_aux(ForwardIterator first, ForwardIterator last,
__true_type){
    // 针对迭代器为char*和wchar*的特化版
    inline void destroy(char*, char*){}
    inline void destroy(wchar_t*, wchar_t*){}
}
```

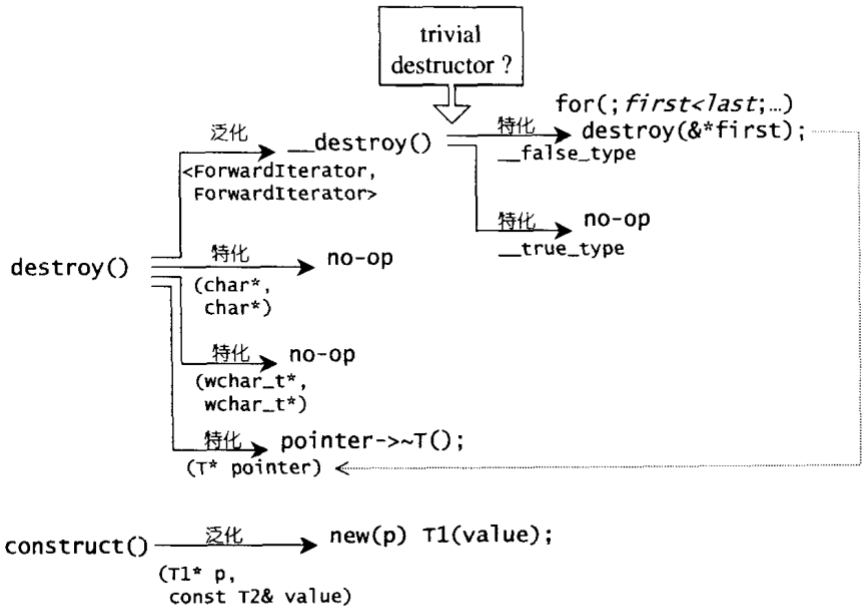


图 2-1 construct() 和 destroy() 示意

2. 空间配置与释放

设计理念

- 向system heap要求空间
- 考虑多线程我状态
- 考虑内存不足的应对措施
- 考虑过多“小型区块”可能造成的内存碎片问题

alloc定义了两级的空间配置器

第一级是对malloc/free简单的封装。 C++内存配置基本操作::*operator new* ,::*operator delete*相当于C的malloc()和free(),SGI正是使用malloc()和free()完成空间配置的。

而为了解决小型区块可能造成的内存破碎问题，alloc采用了第二级空间配置器。第二级空间配置器在分配大块内存(大于128bytes)时，会直接调用第一级空间配置器，而分配小于128bytes的内存时，则使用内存池跟free_list进行内存分配/管理。

第一级空间配置器核心代码实现

```
template<int inst>
class _malloc_alloc_template {
private:
    //以下函数处理内存不足的情况， oom：out of memory
    static void *oom_malloc(size_t);
    static void *oom_realloc(void *,  size_t);
    static void (* __malloc_alloc_oom_handler)();
public:
    // 只是对malloc/free的简单封装
    static void* allocate(size_t n)
    {
        void* res = malloc(n);
        if (0 == res) res = oom_malloc(n);
        return res;
    }
    static void* reallocate(void* p, size_t new_sz)
    {
```

```
        void* res = realloc(p, new_sz);
        if (0 == res) res = oom_realloc(p, new_sz);
        return res;
    }
    static void deallocate(void* p)
    {
        free(p);
    }
    // 用来设置内存不足时的处理函数 该函数参数跟返回值都是一个函数指针
    // 一般会抛出异常/尝试回收内存
    static void(*set_handler(void(*f)()))()
    {
        void(*old)() = __malloc_alloc_oom_handler;
        __malloc_alloc_oom_handler = f;
        return old;
    }
private:
    // 用来处理内存不足的情况
    static void* oom_malloc(size_t n)
    {
        void(*my_handler)();
        void* res;

        for (;;)
        {
            my_handler = _oom_handler;
            if (0 == my_handler) { return NULL; }
            (*my_handler)();
            if (res = malloc(n)) return res;
        }
    }
    // 用来处理内存不足的情况
    static void* oom_realloc(void* p, size_t n)
    {
        void(*my_handler)();
        void* res;

        for (;;)
        {
            my_handler = _oom_handler;
            if (0 == my_handler) { return NULL; }
            (*my_handler)();
            if (res = reallocate(p, n)) return res;
        }
    }
    // 由用户设置，在内存不足的时候进行处理，由上面两个函数调用
    static void(*_oom_handler)();
};

// 处理函数默认为0
typedef _malloc_alloc_template<0> malloc_alloc;

```

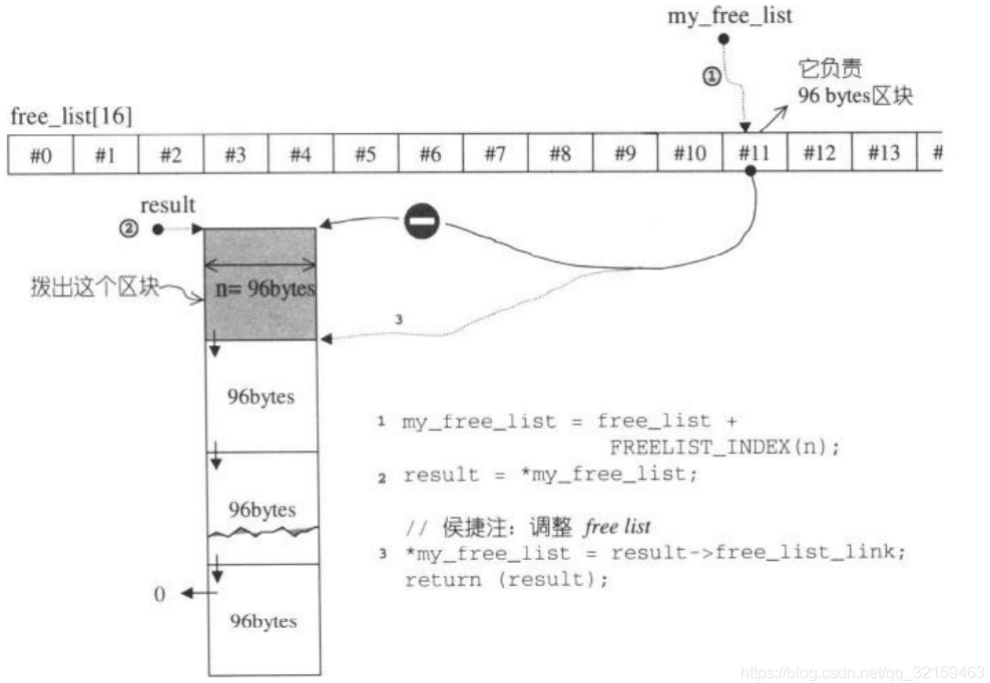
第二级空间配置器

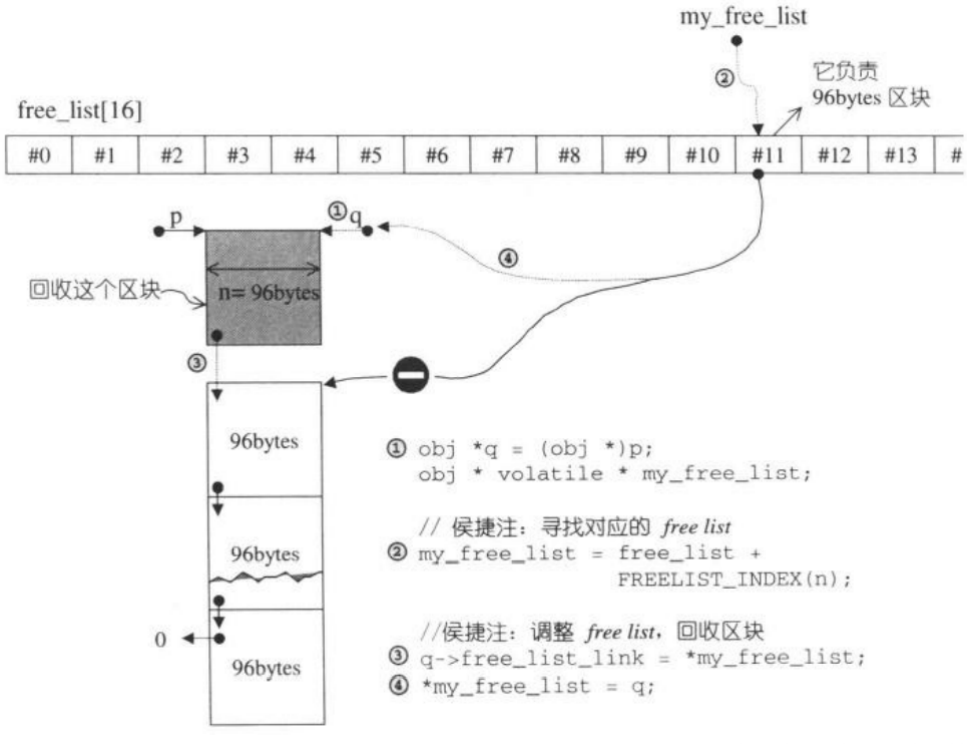
该配置器维护一个free_list，这是一个指针数组。节点结构如下

```
union obj{  // 联合体表示，成员变量不会同时出现，任何时候内存中只有一个
    union obj *free_list_link;
    char client_date[1];
}
```

在分配内存的时候，补足8bytes的倍数，free_list数组中每个指针分别管理分配大小为8、16、24、32...128bytes的内存。下图表示从二级空间配置器中分配内存时是如何维护free_list的(建议参考下面源码阅读)。

开始所有指针都为0，没有可分配的区块时(就是free_list[i]==0)会从内存池中分配内存(默认分配20个区块)插入到free_list[i]中。然后改变free_list[i]的指向，指向下一个区块(free_list_link指向下一个区块，如果没有则为0)。





```cpp
enum { _ALIGN = 8 };     // 对齐
enum { _MAX_BYTES = 128 }; // 区块大小上限
enum { _NFREELISTS = _MAX_BYTES / _ALIGN }; // free-list个数

class default_alloc {
private:
    // 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];
    };
private:
    // 16个free-lists 各自管理分别为8,16,24...的小额区块
    static obj* free_list[_NFREELISTS];
    // 根据区块大小，决定使用第n号free-list
    static size_t FREELIST_INDEX(size_t bytes)
    {
        return (bytes + _ALIGN - 1) / _ALIGN - 1;
    }
    // 分配内存，返回一个大小为n的区块，可能将大小为n的其他区块加入到free_list
    static void* refill(size_t n)
    {
        // 默认分配20个区块
        int nobjs = 20;
        char* chunk = chunk_alloc(n, nobjs);
        obj** my_free_list;
        obj* result, *current_obj, *next_obj;

        // 如果只分配了一个区块，直接返回
        if (1 == nobjs) return chunk;
        // 否则将其他区块插入到free list
        my_free_list = free_list + FREELIST_INDEX(n);
        result = (obj*)chunk;
        // 第一个区块返回 后面的区块插入到free list
        *my_free_list = next_obj = (obj*)(chunk + n);
        for (int i = 1;; ++i)
        {
            current_obj = next_obj;
            next_obj = (obj*)((char*)next_obj + n);
            // 最后一个next的free_list_link为0
            if (nobjs - 1 == i)
            {
                current_obj->free_list_link = 0;
                break;
            }
            current_obj->free_list_link = next_obj;
        }
        return result;
    }
    // 分配内存
    // 在内存池容量足够时，只调整start_free跟end_free指针
    // 在内存池容量不足时，调用malloc分配内存(2 * size * nobjs +
ROUND_UP(heap_size >> 4)，每次调整heap_size += 本次分配内存的大小)
```

```
    static char* chunk_alloc(size_t size, int& nobjs);

    static char* start_free; //内存池的起始位置
    static char* end_free;     //内存池的结束位置
    static size_t heap_size;          //分配内存时的附加量
public:
    static void* allocate(size_t n)
    {
        obj** my_free_list;
        obj* result;
        // 大于128就调用第一级空间配置器
        if (n > (size_t)_MAX_BYTES)
            return base_alloc::allocate(n);

        // 寻找适当的free-list
        my_free_list = free_list + FREELIST_INDEX(n);
        result = *my_free_list;
        // 没有可用的free list
        if (result == 0)
            return refill(ROUND_UP(n));

        // 调整free list 移除free list
        *my_free_list = result->free_list_link;
        return result;
    }
    static void deallocate(void* p, size_t n)
    {
        obj* q = (obj*)p;
        obj** my_free_list;

        // 大于128就调用第一级空间配置器
        if (n > (size_t)_MAX_BYTES)
        {
            base_alloc::deallocate(p);
            return;
        }

        // 寻找对应的free list
        my_free_list = free_list + FREELIST_INDEX(n);
        // 调整free list 回收区块(将区块插入到my_free_list)
        q->free_list_link = *my_free_list;
        *my_free_list = q;
    }
    static void reallocate(void* p, size_t old_sz, size_t new_sz);
};

char* default_alloc::start_free = 0;
char* default_alloc::end_free = 0;
size_t default_alloc::heap_size = 0;
default_alloc::obj* default_alloc::free_list[_NFREELISTS] = { 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

```

第3章 迭代器(iterators)概念与traits编程技法

迭代器是一种行为类似指针的对象，用于连接容器和算法。实现对于一个容器不必知道它的类型，直接获得它的迭代器就可以用于算法的执行。因此，在迭代器要传递所指对象类型或者可以获取到对象的类型。

在算法执行中可能要用到迭代器的相应型别（迭代器的所指之物）——function template的参数推导机制。

```
template <class I, class T>
void func_impl(I iter, T t){   // 在这里可以推导出*iter的类型。
    T tmp;  // T 就是迭代器的相应型别
    // ... 这里做原本func()应该做的工作
};
template <class I>
inline void func(I iter){
    func_impl(iter, *iter); // func的全部工作移往func_impl
}
int main()
{
    int i;
    func(&i);
}
```

1. Traits编程技法

迭代器所指对象的型别，称为该迭代器的value type。即上面例子的T。template参数推导机制推导的只是参数，无法推导返回值类型。**声明内嵌型别**可以解决这个问题。

```
template <class T>
struct MyIter{
    typedef T value_type;   //内嵌型别声明
    T* ptr;
    MyIter(T* p=0):ptr(p) {}
    T& operator*() const {return *ptr;}
};

template <class I>
typename I::value_type   //这一行是返回值类型，typename指明这是一个类型
func(I ite){
    return *ite;
}
MyIter<int> ite(new int(8));
cout<<func(ite);

```

Partial Specialization(偏特化)的意义
原生指针不是class，无法定义内嵌型别，因此这里引入偏特化。且class template有一个以上的template参数时，可以针对其中若干个template参数进行特化处理。

什么是偏特化——针对template参数更进一步的条件限制所设计出来的一个特化版本。

```
template <typename T>
class C{...};   // 这个泛化版本允许接受T为任何型别。

template <typename T>
class C<T*>{...};  // 这个特化版本适用于T为原生指针的型别。
```

针对“迭代器的template参数为指针”者设计特化版的迭代器，使用class template来“萃取”迭代器的特性。

```
template <class T>
struct iterator_traits{
    //创建类型别名、指定时类型、类型别名
    typedef typename I::value_type value_type;
};
// 也就是说，如果I定义了自己的value_type，先前的func可以改写成如下
template <class I>
typename iterator_traits<I>::value_type   // 这一整行是函数返回值
func(I ite) { return *ite; }

// 在带来了一层间接性外，偏特化也带来了traits可以拥有特化版本的好处。
template <class T>
struct iterator_traits<T*>{  // 偏特化版 -- 迭代器是原生指针
    typedef T value_type;
};
//对于迭代器是const T*类型的偏特化，萃取类型为T
template <class T>
struct iterator_traits<const T*>{
    typedef T value_type;
};

```

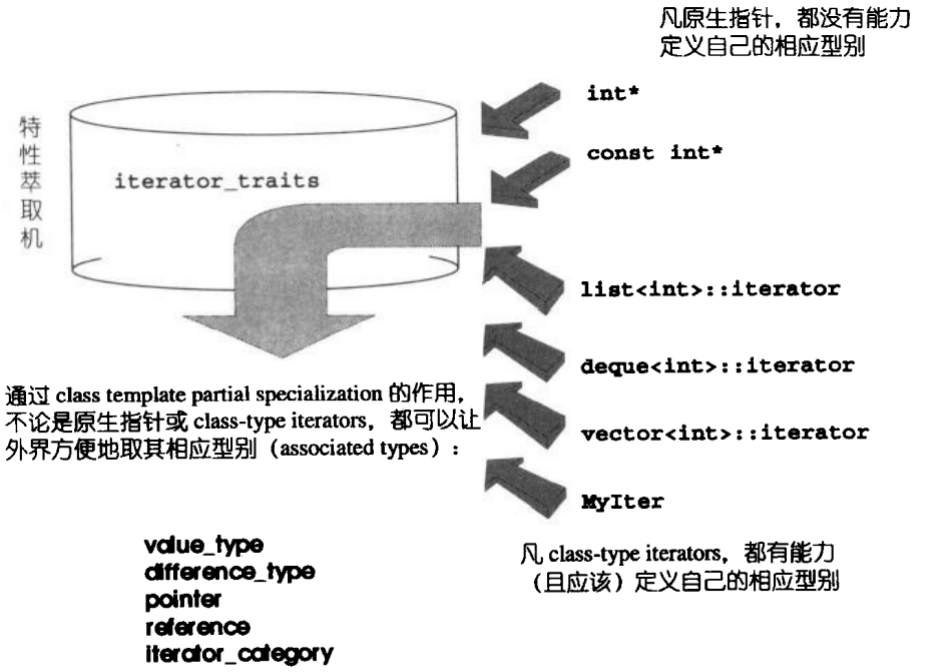


图 3-1　traits 就像一台 “特性萃取机”，榨取各个迭代器的特性（相应型别）

最常用到的迭代器相应类别有五种：***value_type, difference type, pointer, reference, iterator categoly***。为了使容器可以融入STL中，一定要为容器的迭代器定义着五种相应类型。“特性萃取机”traits会把这些特性萃取出来。

```
template <class I>
struct iterator_traits{
    typedef typename I::iterator_category iterator_category;
    typedef typename I::value_type        value_type;
    typedef typename I::difference_type   difference_type;
```

```
    typedef typename I::pointer           pointer;
    typedef typename I::reference         reference;
}
// 针对原生指针设计的特化版本
template <class T>
struct iterator_traits<T*>{
    typedef random_access_iterator_tag iterator_category;
    typedef T                          value_type;
    typedef ptrdiff_t                  difference_type;
    typedef T*                         pointer;
    typedef T&                         reference;
}
// 针对原生const指针设计的特化版本
template <class T>
struct iterator_traits<const T*>{
    typedef random_access_iterator_tag iterator_category;
    typedef const T                    value_type;
    typedef ptrdiff_t                  difference_type;
    typedef const T*                   pointer;
    typedef const T&                   reference;
}

```

2. 五种相应型别

- value type

迭代器所指对象的型别

- different type

两个迭代器之间的距离。STL中的count()返回值就是difference type

```
template <class I, class T>
typename iterator_traits<I>::difference_type  // 函数返回值
count(I first, I last, const T& value){
    typename iterator_traits<I>::difference_type n =0;
    for (; first != last; ++first)
        if (*first == value)
            ++n;
    return n;
}
```

- reference type

const int *p 和 int *p 。如果p是一个constant iterators 其value type是T，那么*p的型别应该是const T&。而不是const T。如果是mutable iterators 那么*p的型别应该是T&，而不是T。

```
int *pi = new int(5);
const int* pci = new int(9);
*pi = 4;
*pci = 1; // 不允许
```

- pointer type

迭代器所指之物（指针）

```
Item& operator*() const{return *ptr;}  // 是ListIter的reference type
Item* operator->() const{return ptr;}  // 是ListIter的point type
```

- iterator_category

迭代器的类型：*Input Iterator,Output Iterator, Forward Iterator, Bidirectional Iterator, Random Access Iterator.*获得迭代器的类型用于函数参数进行函数重载，在编译阶段即可确定使用针对不同迭代器的函数。而迭代器的种类用五个classes定义：（好处1. 可以用于函数重载，2. 可以利用继承关系而使用重载给父类的函数）

```
struct input_iterator_tag{};
struct output_iterator_tag{};
struct forward_iterator_tag: public Input_Iterator{};
struct bidirectional_iterator_tag: public Forward_Iterator{};
struct random_access_iterator_tag: public Bidirectional_Iterator{};
```

比如advance()有三个版本：1.依次++i（Input Iterator）；2. 能逆向移动++i或--i（Bidirectional Iterator）；3. 跳跃前进 i+=n（Random Access Iterator）；利用iterator_category可以重新设计可重载的__advance函数。（代码：P95）

```
template <class InputIterator, class Distance>
inline void advance(InputIterator& i, Distance n){
    __advance(i,n, iterator_traits<InputIterator>::iterator_category()
    );
}
```

为了符合规范，任何迭代器都应该提供五个内嵌相应型别，以利于traits萃取，否则无法与STL其他组件搭配。STL提供一个iterators class，新设计的迭代器可以继承它，可以保证符合STL的规范。

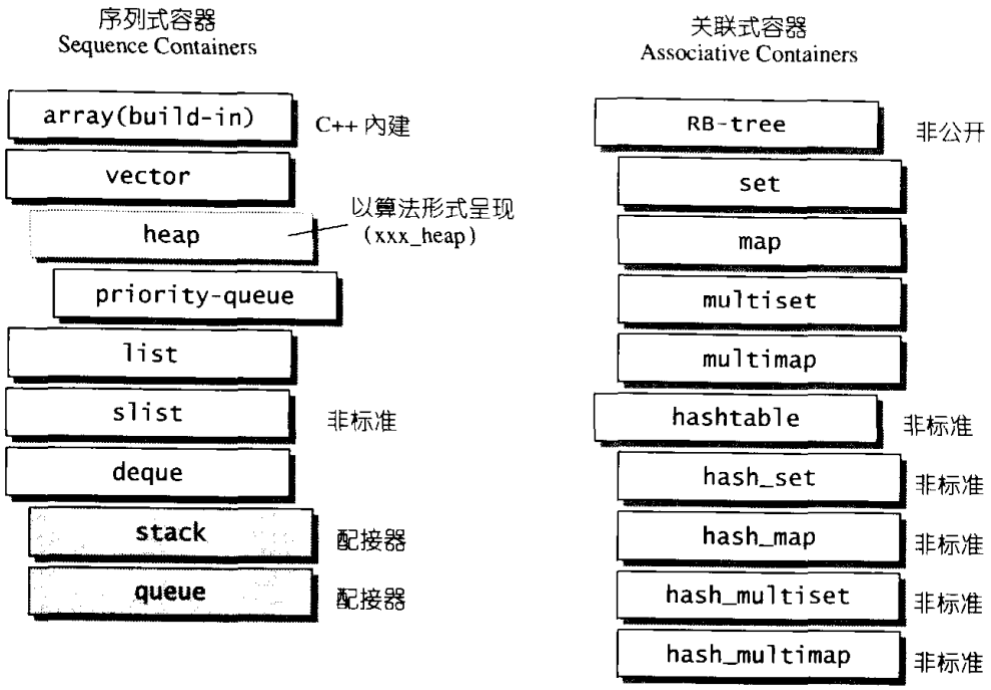
```
template  <class Category,
           class T, class Distance=ptrdiff_t, class Point = T*,class
Reference=&T>
struct iterator{
    typedef Category  iterator_category;
    typedef T         value_type;
    typedef Distance  difference_type;
    typedef Pointer   pointer;
    typedef Reference reference;
};
// 自定义的迭代器
template <class Item>
struct ListIter : public std::iterator<std::forward_iterator_tag, Item>
{...}
```

3. 总结

在使用迭代器去“撮合”算法和容器时，需要知道容器对象的类型。由此，首先引入了function template参数推导机制，该方法解决了函数形参的推导，但是无法推导出函数的返回值类型。接着有介绍了class的内嵌类型声明，在类（或结构体）中通过typedef T value_type取得参数类型，然后让返回值类型为typename T::value_type获得T的参数类型,但这种方法只在class对象中有效，在原生指针中无法定义内嵌类型。此时，本章最重要的traits特性萃取机就闪亮登场了。在iterator_traits中typedef重命名 I 中的对象类型，对原生指针特化的iterator_traits中重命名其对象类型，名称保持一致。这样实现了获取任何迭代器所指对象的类型。

第4章 序列式容器

此处的衍生关系并非继承而是内含关系。



1. vector

数据结构定义：

```
template <class T, class Alloc=alloc>
class vector{
private:
    //vector的嵌套型别定义
    typedef T value_type;
    typedef value_type* pointer;
    typedef value_type* iterator;
    typedef value_type& reference;
    typedef size_t      size_type;
    typedef ptrdiff_t   difference_type;
protect:
    // simple_alloc是SGI STL默认的空间配置器
    typedef simple_alloc<value_type, Alloc> data_allocator;
    iterator start;              // 表示目前使用空间的头
    iterator finish;             // 表示目前使用空间的尾
    iterator end_of_storage;     // 表示目前可用空间的尾
};

```

一个vector的容量永远大于等于其大小，当vector空间不够用时，容器的扩张必须经过“重新配置空间、元素移动、释放原空间”等过程。扩充空间的事件成本比较高，为避免多次扩充，我们会**将容量扩充两倍**。如果两倍还不够用，就扩充更大的容量。

vector提供的接口：包括得到vector的属性接口、vector的操作接口以及构造函数：

（1）构造函数：vector()、vector(size_type n、const T& value)、vector(size_type n);

（2）属性函数：begin、end、size、capacity、empty、operator[]、front和back

（3）操作函数：push_back()、pop_back()、erase()、resize()、clear()、insert()。

重点关注一下insert()：插入分为三种情况

（1）备用空间不够——开辟新空间，移动插入点之前的元素，插入新元素，移动插入点之后的元素，释放原空间。

（2）插入元素多于插入点之后的元素——将后面的元素移动到备用空间正确位置，插入新元素。

（3）插入元素少于插入点之后的元素——先在末尾插入一定数量新元素直到插入点之后的大小等于需要插入的元素数量，再将插入点之后的元素里的旧元素复制到备用空间，而后插入新元素。

2. list

list的插入和删除是常数时间，并且对空间的运用不浪费。list是双向链表。（SGI的list是双向环状链表）

```
template<class T>
struct __list_node{
    typedef void* void_pointer;
    void_pointer prev;
    void_pointer next;
    T data;
};
```

list的迭代器是双向迭代器，插入和删除不影响迭代器。而vector可能会影响（当备用空间不够重新申请空间时）

```
template<class T,class Ref,class Ptr>
struct __list_iterator{
    typedef __list_iterato<T,T&,T*>  iterator;
    typedef __list_iterato<T,Ref,Ptr>  self;
    typedef bidirectional_iterator_tag iterator_category;//双向迭代器
    tyepdef  __list_node<T>* link_type;
    link_type node;//包含了一个指向__list_node节点
    .....
};
```

list提供的接口：包括得到list的属性接口、list的操作接口以及构造函数：

（1）构造函数：list()、list(size_type n、const T& value)、list(size_type n)。

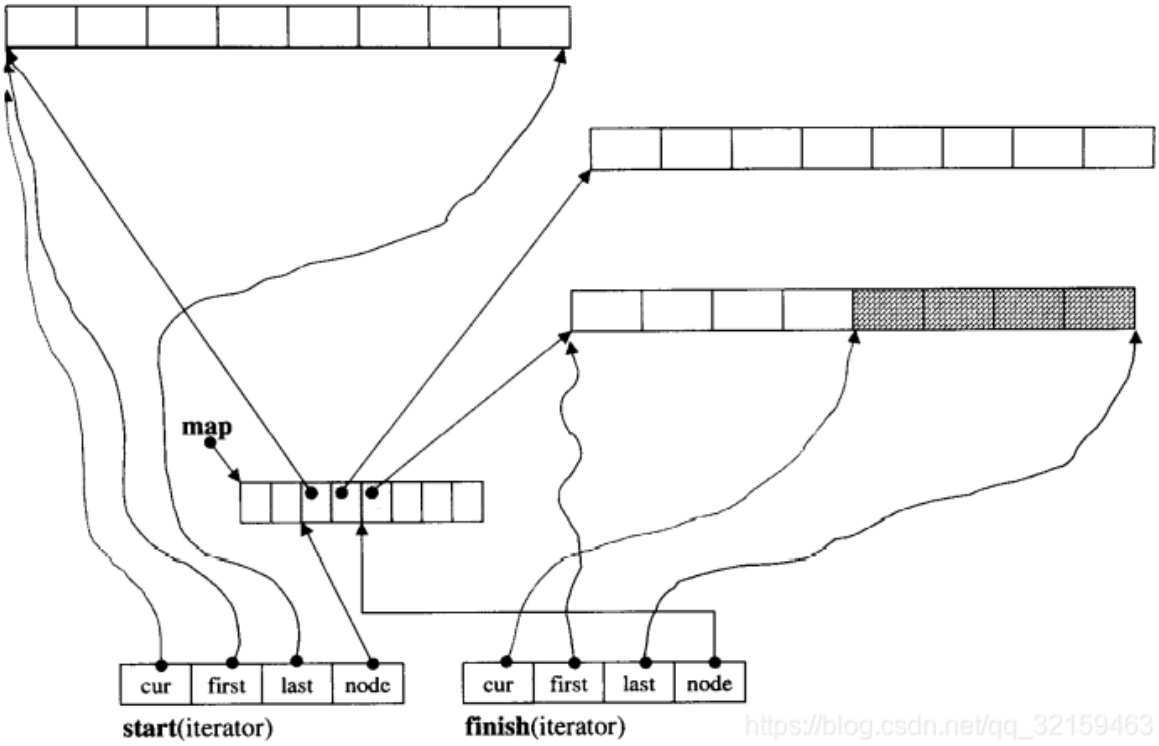
（2）属性函数：begin、end、empty、size、front和back。

（3）操作函数：push_back()、pop_back()、push_front、pop_front、erase()、resize()、clear()、unique、**splice**、merge、reverse、sort（list的sort是自己的快排sort，因为STL的算法sort要求是RamdonAccessIterator）、insert。（内部有一个transfer接口，将某连续范围元素迁移到某个特定位置之前）

3. deque

vector是单向开口，deque是双向开口的**连续线性空间**。能用vector不用deque。为高效排序deque，先将deque复制为vector，排完后复制回deque。但这种连续线性空间是逻辑上的连续，物理上是通过 *map* 中控器实现。

因而为实现逻辑上的连续，对迭代器要求很高，deque维护两个迭代器——start和finish。迭代器设计如下图。

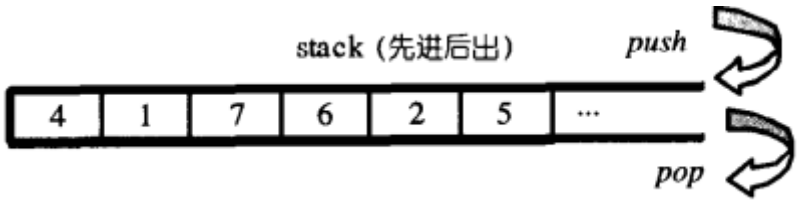


构造：一个map最少管理8个结点，最多管理“所需结点数“+2（前后各预留一个）所需结点=元素个数/缓冲区大小+1。当正好整除则多分配一个。然后设置使用的缓冲区正好处于map管理的正中央（确保前后预留差不多大），即设置nstart和nfinish指针。为正在使用的结点分配缓冲区，填充数据。为两个迭代器（start和finish）设置内容。

push_back和push_front的操作可以自行想象了。有关删除的操作，当缓冲区无元素时会及时释放。但当deque全空时（即clear（）等），保有一个空缓冲区。插入操作——根据插入点前后元素数量谁多谁少，决定移动哪边。

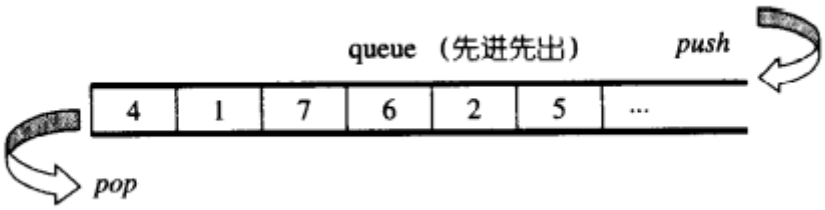
4. stack和queue

栈：先进后出，由deque的缺省实现——因而属于容器配接器。stack没有迭代器。




```
template<class T,class sequece=deque<T> >
class stack{
protected:
   Sequence c;//所有的接口转到调用C的接口但是只操作一端
    ...
}
```

队列——先进先出。也没有迭代器。




```
template<class T,class sequece=deque<T>>
class queue{
protected:
    Sequence c;//这个跟上面不同的是两端不封死
}
```

5. heap

堆并不是STL提供的容器，而是作为优先队列的底层实现。heap组成为：vector+一些算法组成以完全二叉树的结构。对于在vector中位于`i`下标的结点，其左孩子下标为`2i`，右孩子为`2i+1`其父节点为`i/2`。

堆分为大根堆和小根堆。即每个节点的key都大于等于（小于等于）其子结点。STL提供的为大根堆。heap没有迭代器，其算法主要为：push_heap, pop_heap, sort_heap。

- push_heap()

插入新节点在vector.end()处，依次向上调整（上溯），维持大根堆。

- pop_heap()

将vector首个元素置于覆盖尾部，然后进行下溯——将较大节点提上来直至叶节点。而后将原来尾部的节点填充空出来的位置，从该路径的叶节点进行一次上溯。

- sort_heap()

即做多次pop_heap(). 每次都把最大的放最后，最终得到一个增序序列。

6. priority_queue

缺省情况下是用一个max-heap实现。

```
template<class T, class Sequence = vector<T>,
        class Compare = less<typename Sequence::value_type>
class priority_queue{
protected:
    Sequence c;   // 底层容器
    Compare comp;   // 元素大小比较标准
}
```

7. slist

slist是一种单链表结构，slist和list的差别：list提供的Bidirectional iterator迭代器，而slist提供的是Forward Iterator。slist和list共同具有的特色是他们的插入、移除、结合等操作并不会造成迭代器失效。插入操作会将新元素插入于指定位置之前，而非之后。这样slist每次插入都要从头遍历找到前一个节点。

```
struct __slist_node_base{
```

```
    __slist_node_base *next;//可以作为头节点
}
template<class T>
struct __slist_node:public  __slist_node_base{  // 继承而来
    T data;
}

struct __slist_iterator_base{
     typedef forward_iterator_tag iterator_category;
     __slist_node_base *node;
    ...
}
template<class T,class Ref,class Ptr>
struct  __slist_iterator: public __slist_iterator_base{
     typedef  __slist_iterator<T,T&,T*> iterator;
    ...
}

template<class T,class  Alloc=alloc>
class slist{
     typedef  __slist_node<T> list_node;
     typedef  __slist_iterator<T,T&,T*> iterator;
    ...
}

```

对于它的迭代器，只能实现++，不能实现--。

第5章 关联式容器

关联式容器：每笔数据(每个元素) 都有一个键值(key)和一个实值(value) 。当元素被插 人到关联式容器中时，容器内部结构(可能是RB-tree,也可能是hash- table)便依照其键值大小，以某种特定规则将这个元素放置于适当位置。

一般而言，关联式容器的内部绐构是一个平衡二叉树（包括AVL-tree平衡二叉搜索树、RB-tree、AA-tree）。最广泛用于STL的RB-tree（红黑树）。

1. 树（RB-tree）

- 二叉搜索树：左孩子 < 结点 < 右孩子。不是平衡二叉树。
- AVL树：插入和删除的调整。这东西不如去看数据结构，没必要在STL里面看。
- RB树：因为STL用到了而我又不熟悉，所以记录一下

  是平衡二叉搜索树，根据规则3,4,得新增节点必须为红，新增节点的之父节点必须为黑。否则需要调整树形。

  1. 每个节点不是黑的就是红的
  2. 根节点是黑的
  3. 如果节点为红，则其子节点必须为黑。（不能连红）
  4. 任一节点到尾端的所有路径上所含黑节点数相同。

2. set

set的特性是所有元素的键值自动被排序，set的不允许有两个相同的键值，其中的元素不能被改变，以RB-tree为底层机制。对于set，STL提供了一组算法：交并差等。

```
template <class Key, class Compare=less<Key>, class Alloc=alloc> //默认采用递增排序
class set{
public:
    typedef Key key_type;
    typedef Key value_type;

    typedef Compare key_compare;
    typedef Compare value_compare;
private:
    template <class T> struct identity:public unary_function<T, T>{
        const T& operator()(const T& x) const { return x; }
    }
    typedef rb_tree<key_type, value_type, identity<value_type>, key_compare, Alloc> rep_type;
    rep_type t;     //采用RB-tree来表现set
public:
    typedef typename rep_type::const_pointer pointer;
    typedef typename rep_type::const_pointer const_pointer;
    typedef typename rep_type::const_reference reference;
    typedef typename rep_type::const_reference const_reference;
    typedef typename rep_type::const_iterator iterator;    // set的迭代器无法执行写入操作
    ....

    // set一定使用rb-tree的insert_unique(),multiset才使用insert_equal()
    // set不允许相同键值存在， multiset允许相同键值存在
    template <class InputIterator>
    set(InputIterator first, InputIterator last):t(comp){
        t.insert_unique(first, last); // 调用
    }
};

```

3. 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的元素由pair组成，同时拥有key和value。不允许由相同的key。

pair的定义：

```
template <class T1, class T2>
struct pair{
    typedef T1 first_type;  // 是public的!
    typedef T2 second_type;

    T1 first;
    T2 second;

    pair() : first(T1()), second(T2()) {}
    pair(const T1& a, const T2& b) : first(a), second(b){}
};

```

map的核心代码:

```
template <class Key, class T, class Compare=less<Key>, class Alloc=alloc> //
默认采用递增排序
class map{
public:
    typedef Key key_type;                    // 键值
    typedef T   data_type;                   // 实值
    typedef T mapped_type;
    typedef pair<const Key, T> value_type;  //元素型别（键值/实值）
    typedef Compare key_comapre;   //键值比较函数

    class value_compare: public binary_functioon<value_type, value_type,
bool>{
      friend class map<Key, T, Compare, Alloc>;
      protected:
        Compare comp;
        value_compare(Comapre c):comp(c){}
      public:
        bool operator()(const value_type& x, const value_type& y) const{
            return com(x.first, y.first);
        }
    };

private:
    typedef rb_tree<key_type, value_type, selectlst<value_type>,
key_compare, Alloc> rep_type;
    rep_type t;   //使用rb_tree;

public:
    typedef typename rep_type::pointer pointer;
    typedef typename rep_type::const_pointer const_pointer;
    typedef typename rep_type::reference reference;
    typedef typename rep_type::const_reference const_reference;
    typedef typename rep_type::iterator iterator;   // map的iterator可以修改元
素的值(value)
    typedef typename rep_type::const_iterator const_iterator;

    ....
    // map使用insert_unique插入
    template <class InputIterator>
    map(InputIterator first, InputIterator last, const Comapre&
comp):t(comp){

```

```
        t.insert_unique(first, last);
    }
}

```

multiset/multimap 于set/map唯一差别在于插入时使用`insert_equal()`

4.hashtable

hashtable 使用hash function将元素映射的某一位置上，但这无法避免的会产生碰撞解决方法有线性探测、二次探测、开链等。

- **线性探测**

使用hash function计算出某个元素的插入位置，如果该位置上的空间不可用时，继续向下寻找可用空间。

存在的问题：可能过去的元素集中在某一区域，导致需要不断的解决碰撞问题。

- **二次探测**

采用F(i)=i^2映射函数，当发生碰撞时，每次向下寻找第1、4、9、16...位置上的空间是否可用

- **开链**——STL采用此方法

每一个表格元素维护一个list，hash function会分配某一个list

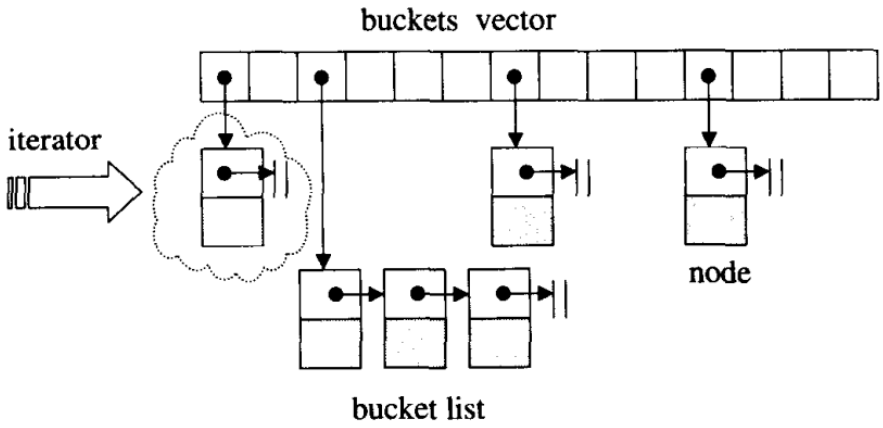


图 5-24 以开链（separate chaining）法完成的 hash table。SGI 即采此法。

hash_table 节点定义

```
template <class Value>
struct __hashtable_node
{
    __hashtable_node* next;
    Value val;
}
```

bucket所维护的linked list并不采用STL的list或slist，而是自行维护。而buckets聚合体以vector完成。vector表格的大小使用提前存储好的**28个质数**中最接近的一个。

hashtable数据结构

```
template <class Value, class Key,
    class HashFcn,  //hash function的函数类别
```

```
    class ExtractKey,    //取出键值的方法(函数或仿函数)
    class EqualKey,      //判断键值是否相同的方法(函数或仿函数)
    class Alloc>    // Alloc默认使用alloc
class hashtable{
public:
    typedef HashFcn hasher;
    typedef EqualKey key_value;
    typedef size_t size_type;
private:
    hasher hash;
    key_value equals;
    ExtractKey get_key;

    typedef __hashtable_node<Value> node;
    typedef Simple_alloc<node, Alloc> node_allocator;

    vector<node*, Alloc> buckets; // 以vector表示
    size_type num_elements;
public:
    // bucket个数代表bucket vector的大小
    size_type bucket_count() const { return bucket.size(); }
    ...
};

```

hash_map/hash_set/hash_multiset/hash_multimap

均采用hashtable为底层机制，它们的使用方式与采用br-tree的map/set类似。

第6章 算法

分为：质变算法和非质变算法。其中质变算法分为就地算法（如replace()）和另地算法（replace_copy()）。可以通过有无copy来区分。非质变算法分为默认算法和带有仿函数为参数的算法，如find() 和 find_if()。

算法总览：(没放全，查书吧)

算法名称	算法用途	质变?	所在文件
accumulate	元素累计	否	<stl_numeric.h>
adjacent_difference	相邻元素的差额	是 if in-place	<stl_numeric.h>
adjacent_find	查找相邻而重复（或符合某条件）的元素	否	<stl_algo.h>
binary_search	二分查找	否	<stl_algo.h>
Copy	复制	是 if in-place	<stl_algobase.h>
Copy_backward	逆向复制	是 if in-place	<stl_algobase.h>
Copy_n *	复制 n 个元素	是 if in-place	<stl_algobase.h>
count	计数	否	<stl_algo.h>
count_if	在特定条件下计数	否	<stl_algo.h>
equal	判断两个区间相等与否	否	<stl_algobase.h>
equal_range	试图在有序区间中寻找某值（返回一个上下限区间）	否	<stl_algo.h>
fill	改填元素值	是	<stl_algobase.h>
fill_n	改填元素值，n 次	是	<stl_algobase.h>
find	循序查找	否	<stl_algo.h>
find_if	循序查找符合特定条件者	否	<stl_algo.h>
find_end	查找某个子序列的最后一次出现点	否	<stl_algo.h>
find_first_of	查找某些元素的首次出现点	否	<stl_algo.h>
for_each	对区间内的每一个元素施行某操作	否	<stl_algo.h>

- remove() —— 移除但没有删除
- unique() —— 去重但没删
- STL 的排序算法：数据量大时采用快排，分段递归排序。一旦小于某个门槛，则使用插入排序，如果递归层次过深就使用堆排。

```
template<class RandomAccessIterator>
inline void sort (RandomAccessIterator first, RandomAccessIterator last){
    if (first != last){
        __introsort_loop(first, last, value_type(first), __lg(last-first)*2);
        __final_insertion_sort(first, last);
    }
}
// 其中 __lg() 用来控制分割恶化情况
// 找出 2^k <= n 的最大k
template <class Size>
inline Size __lg(Size n){
    Size k;
    for(k = 0; n>1; n>>=1) ++k;
    return k;
}

// 当元素个数为40时，__lg(last-first)*2 = 5*2;  2^5=32<=40。最多允许分割10层。
template < class RandomAccessIterator, class T, class Size>
void __introsort_loop(RandomAccessIterator first, RandomAccessIterator last,
```

```
                    T*, Size depth_limit){
    while(last - first > __stl_threshold) { // __stl_threshold = 16 是个全局常数
        if(depth_limit == 0) {   // 至此分割恶化
            partial_sort(first, last, last); // 改用heapsort
            return;
        }
    }
    --depth_limit;
    // 以下是 median-of-3 partion ，选择首，中，尾的中位数作为枢轴
    RandomAccessIterator cut = __unguarded_partition(first, last,
                                        T(__median(*first,
                                                   *(first + (last-first)/2),
                                                   *(last-1)
                                          ));
     // 对右半进行递归
     __introsort_loop(cut, last, value_type(first), depth_limit);
     last = cut; // 这里再次回到while循环，对左半段进行递归
}
```

第7张 仿函数

仿函数从参数类型上可以分为一元和二元仿函数。从功能上可以分为：逻辑运算、算术运算、关系运算。

STL内置的仿函数在头文件内。

```
// 定义一元函数的参数类型和返回值类型，任何Adaptation Unary Function都应该继承此类别
template <class Arg, class Result>
struct unary_function{
    typedef Arg argument_type;
    typedef Result result_type;
};
// nagete继承unary_function
template <class T>
struct negate:public unary_function<T, T>{
    T operator()(const T& x) const { return -x; }
};

```

```
// 定义二元函数的第一参数类型、第二参数类型，以及返回值类型
template <class Arg1, class Arg2, class Result>
struct binary_function{
  typedef Arg1 first_argument_type;
  typedef Arg2 second_argumet_type;
  typedef Result result_type;
};
// 以下仿函数继承binary_function
template <class T>
struct plus: public binary_function<T, T, T>{
    T operator()(const T& x, const T& y) const {return x+y;}
};

```