

## 第1章 STL 概论

### 1.2 STL 六大组件

STL 提供六大组件，彼此可以组合套用：

- 1.容器 (container)：各种数据结构，如 vector,list,deque,set,map 等
- 2.算法 (algorithm)：各种常用算法如 sort,search,copy,erase.....
- 3.迭代器 (iterator)：扮演容器与算法之间的胶着剂。所以 STL 容器都附带有自己专属的迭代器。指针也是一种迭代器。
- 4.仿函式 (functors)：行为类似函数，可作为算法的某种策略，从实现的角度讲，仿函式是一种重载了 operator()的 class 或 class template。**仿函式是否就是 c++primer 中的函数对象？**
- 5.适配器 (adaptor)：一种用来修饰容器或仿函式或迭代器接口的东西，有 function adaptor, container adaptor, iterator adaptor。
- 6.分配器 (allocator)：负责空间分配与管理。

### 1.7 STLport 版本

STLport 版本是以 SGI STL 为蓝本的高度可移植性版本。

## 第 2 章 空间分配器

### 2.1 空间配置器的标准接口

根据 STL 规范，以下是 allocator 的必要接口：

第一组：各种 type 类型

//以下各种 type 的设计原由，第三章详述。

allocator::value\_type

allocator::pointer

allocator::const\_pointer

allocator::reference

allocator::const\_reference

allocator::size\_type

allocator::difference\_type

第二组：构造与析构函数

Allocator::rebind

一个嵌套的 (nested) class template。class rebind<U>拥有唯一成员 other，那是一个 typedef，代表 allocator<U>。

allocator::allocator()---默认构造函数

allocator::allocator(const allocator&)---拷贝构造函数

template <class U>allocator::allocator(const allocator<U>&) --- 泛化的拷贝构造函数

allocator::~~allocator()---默认的析构函数

第三组：取地址函数

pointer allocator::address(reference x) const --- 传回某个对象的地址，算式 a.address(x)等同于 &x。

const\_pointer allocator::address(const\_reference x) const --- 传回某个 const 对象的地址，算式 a.address(x)等同于 &x。

第四组：空间分配与释放

pointer allocator::allocate(size\_type n, const void\* = 0) --- 配置空间，足以储存 n 个 T 对象。第二自变量是个提示。实作上可能会利用它来增进区域性 (locality)，或完全忽略之。

void allocator::deallocate(pointer p, size\_type n) --- 归还先前分配的空间。

size\_type allocator::max\_size() const --- 传回可成功分配的最大量。

第五组：construct 和 destroy 函数

void allocator::construct(pointer p, const T& x) --- 等同于 new((const void\*) p) T(x)。

void allocator::destroy(pointer p) --- 等同于 p->~T()。

#### 2.1.1 设计一个空间配置器 JJ:: allocator

在书中，侯捷先生写了一个分配器 JJ:: allocator。代码见书。

### 2.2 具备 sub-allocation 的 SGI allocator

SGI STL 配置器与标准规范不同，其名称是 `alloc` 而非 `allocator`，而且不接受任何自变量。如果要在程序中使用 SGI STL 配置器，则不能采用标准写法：

```
Vector<int, std::allocator<int>> iv; //在 vc 或 c++ builder 中这么写
```

```
Vector<int, std::alloc> iv; //在 GCC 中这么写
```

但是令人欣慰的是，我们通常使用预设的空间配置器，很少需要自行指定配置器，而 SGI STL 的每一个容器都已经指定其预设的空间配置器为 `alloc`。

### 2.2.1 SGI 标准的空间配置器 `std::allocator`

该配置器符号部分标准，但效率差，SGI 自己不用，也不建议我们使用。

### 2.2.2 SGI 特殊的空间配置器 `std::alloc`

看下面的代码：

```
Class Foo{
    .....
}
```

```
Foo *of = new Foo; //配置内存，然后构造对象
```

```
Delete pf; //将对象析构，然后释放内存
```

为了精密分工，STL `allocator` 提供四个动作：`alloc::allocate()` 负责内存分配，`alloc::deallocate()` 负责内存释放，`::construct()` 负责对象构造，`::destroy()` 负责对象析构。

配置器定义于 `<memory>` 中，其中包括如下两个头文件：

```
#include <stl_alloc.h> //负责内存分配与释放
```

```
#include <stl_construct.h> //负责对象构造和析构
```

### 2.2.3 构造和析构函数：`construct` 和 `destroy`

下面代码是 `<stl_construct.h>` 中的部分内容：

```
#include<new.h>
```

```
//construct()接受一个指针 p 和一个初值 value
```

```
Template<class T1, class T2>
```

```
Inline void construct(T1* p, const T2& value){
```

```
    New(p) T1(value); //placement new 用法;调用构造函数 T1::T1(value)。这里用到了 placement new，它能实现在指定的内存地址上用指定类型的构造函数构造一个对象
}
```

```
//destroy()的第一个版本，接受一个指针
```

```
Template<class T>
```

```
Inline void destroy(T* pointer){
```

```
    Pointer->~T(); //调用析构函数 ~T()
}
```

在调用 `destroy()` 函数同时释放 `n` 个对象（假设类型为 `T`）时，SGI 提供了方法可以判定对象是否有 `non-trivial destructor` (trivial:琐碎的，无价值的，不重要的)，如果没有则不必要循

环为每个对象调用 `T::~T()`，以提高效率代码如下：

```
//以下是 destroy()的第二版本，接受两个迭代器，准备将[first, last)范围内的所有物件析
//构掉，因为不知道这个范围有多大，万一很大，但是每个物件的析构函数都是无关痛
//痒的（trivial destructor），那么一次次呼叫这些无关痛痒的析构函数，对效率是一种损
//害，所以此函数设法找出元素的数值类型，进而利用__type_traits<>选 //择适当措
//施
```

```
template <class ForwardIterator>
// __false_type 表明是具有 non trivial destructor，所以要循环调用 destroy
inline void __destroy_aux(ForwardIterator first, ForwardIterator last, __false_type) {
    for ( ; first < last; ++first)
        destroy(&*first);
}
```

```
template <class ForwardIterator>
//__true_type 表明是具有 trivial destructor 不需要调用 destroy
inline void __destroy_aux(ForwardIterator, ForwardIterator, __true_type) {} //空函数体
```

```
//判断元素的型别，是否有 trival destructor
template <class ForwardIterator, class T>
inline void __destroy(ForwardIterator first, ForwardIterator last, T*) {
    typedef typename __type_traits<T>::has_trivial_destructor trivial_destructor;
    __destroy_aux(first, last, trivial_destructor());
}
```

```
template <class ForwardIterator>
inline void destroy(ForwardIterator first, ForwardIterator last) {
    __destroy(first, last, value_type(first));
}
```

```
//以下是 destroy()第二版本针对迭代器为 char*和 wchar*的特化版
Inline void destroy(char*, char*){}
Inline void destroy(wchar_t*, wcht_t*){}
```

但是 c++本身并不直接支持对“指针所指之物”的型别判断，也不支持“对象析构函数是否是 trival”的判断，这需要借助于 Traits 编程技法来完成的：

```
typedef typename __type_traits<T>::has_trivial_destructor trivial_destructor;
首先使用 value_type()获取迭代器指向的物体类型，然后使用__type_traits<T>查看 T 是
否有 non-trivial destructor。
```

#### 2.2.4 空间的配置与释放 std::alloc

对象构造前的空间分配和对象析构之后的空间释放，由<stl\_alloc.h>负责，SGI 对此的设计哲学如下：

向 system heap 申请空间;  
考虑多线程情况;  
考虑内存不足时的应对措施;  
考虑过多小型区块可能造成的内存碎片 (fragment) 问题;

下面摘录的代码都暂时没有考虑多线程情况的处理。

C++内存分配基本动作是::operator new(),内存释放动作是::operator delete().这两个全域函数相当于c的 malloc()和 free(),SGI正是以 malloc()和 free()完成内存的分配与释放。

考虑小型区块可能造成的内存碎片问题,SGI设计了双层级配置器,低一级分配器直接使用 malloc()和 free(),第二级分配器则视情况采用不同策略:当分配区块超过 128bytes,则视之“足够大”,便使用低一级分配器;当分配区块小于 128bytes,则视之“过小”,便采用复杂的 mempool 方式。

究竟使用哪种分配器,取决于\_\_USE\_MALLOC是否被定义:

```
# ifdef __USE_MALLOC
...
typedef __malloc_alloc_template<0> malloc_alloc;
typedef malloc_alloc alloc; //令 alloc 为第一级配置器
# else
...
//令 alloc 为第二级配置器
typedef __default_alloc_template<__NODE_ALLOCATOR_THREADS, 0> alloc;
#endif /* !__USE_MALLOC */
```

其中,\_\_malloc\_alloc\_template就是第一级分配器,\_\_default\_alloc\_template就是第二级分配器。

下面的小节中,我想以自底向上的顺序介绍 STL 的 allocator,首先讨论 STL 内建两种分配器,然后介绍 STL 如何封装这两种分配器对外提供统一的接口,最后用一个 vector 的例子看看容器如何使用这个 allocator。

#### 2.2.5 第一级分配器\_\_malloc\_alloc\_template

该分配器是对 malloc、alloc、free 的封装,并作出类似 c++ new-handler 的机制(所谓 new-handler 机制是指,你可以要求系统在内存分配无法被满足时,唤起一个你所指定的函数,也就是说在::operator::new 无法完成任务,在丢出 std::bad\_alloc 异常之前,会先调用用户指定的处理例程,即 new-handler),这里作出类似 c++ new-handler 机制,而不能直接用 c++ new-handler 机制,是因为他不是使用::operator new 来分配内存:

```
#if 0
#   include<new>
#   define __THROW_BAD_ALLOC throw bad_alloc
#elif !defined(__THROW_BAD_ALLOC)
#   include <iostream.h>
#   define __THROW_BAD_ALLOC cerr<<"out of memory"<<endl;exit(1)
#endif
```

//注意，无「template 型别参数」。至于「非型别参数」inst，完全没派上用场。

```
template <int inst>
```

```
class __malloc_alloc_template {
```

```
private:
```

```
//以下都是函数指针，所代表的函式将用来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:
```

```
static void * allocate(size_t n)
```

```
{
    void *result = malloc(n); //第一级配置器直接使用 malloc()
    // 以下，无法满足需求时，改用 oom_malloc()
    if (0 == result) result = oom_malloc(n);
    return result;
}
```

```
static void deallocate(void *p, size_t /* n */)
{
    free(p); //第一级配置器直接使用 free()
}
```

```
static void * reallocate(void *p, size_t /* old_sz */, size_t new_sz)
```

```
{
    void * result = realloc(p, new_sz); //第一级配置器直接使用 realloc()
    // 以下，无法满足需求时，改用 oom_realloc()
    if (0 == result) result = oom_realloc(p, new_sz);
    return result;
}
```

//以下模拟 C++的 set\_new\_handler(). 换句话说，你可以透过它，

//指定你自己的 out-of-memory handler

```
static void (* set_malloc_handler(void (*f)()))() //蓝色部分作为参数，最后一个()和 void(*)
```

//一起组成 void(\*)()表示返回值是一个函数指针

```
{
    void (* old)() = __malloc_alloc_oom_handler;
    __malloc_alloc_oom_handler = f;
    return(old);
}
```

```

};

// malloc_alloc out-of-memory handling
//初值为 0。有待用户设定。 __malloc_alloc_oom_handler 是一个函数指针
template <int inst>
void (* __malloc_alloc_template<inst>::__malloc_alloc_oom_handler)() = 0;

template <int inst>
void * __malloc_alloc_template<inst>::oom_malloc(size_t n)
{
    void (* my_malloc_handler)();
    void *result;

    for (;;) {
        //不断尝试释放、配置、再释放、再配置...
        my_malloc_handler = __malloc_alloc_oom_handler;
        if (0 == my_malloc_handler) { __THROW_BAD_ALLOC; }
        (*my_malloc_handler)();//呼叫处理例程，企图释放内存。
        result = malloc(n); //再次尝试配置内存。
        if (result) return(result);
    }
}

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

    for (;;) {
        //不断尝试释放、配置、再释放、再配置...
        my_malloc_handler = __malloc_alloc_oom_handler;
        if (0 == my_malloc_handler) { __THROW_BAD_ALLOC; }
        (*my_malloc_handler)();//呼叫处理例程，企图释放内存。
        result = realloc(p, n);//再次尝试配置内存。
        if (result) return(result);
    }
}

//注意，以下直接将参数 inst 指定为 0。
typedef __malloc_alloc_template<0> malloc_alloc;

```

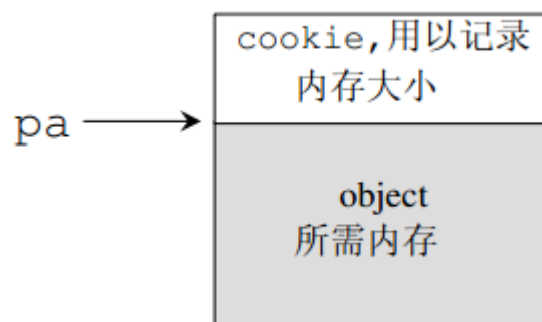
当调用 `malloc` 和 `realloc` 申请不到内存空间的时候，会改调用 `oom_malloc()` 和 `oom_realloc()`，这两个函数会反复调用用户传递过来的 out of memory handler 处理函数，直

到能用 malloc 或者 realloc 申请到内存为止。如果用户没有传递 \_\_malloc\_alloc\_oom\_handler, \_\_malloc\_alloc\_template 会抛出 \_\_THROW\_BAD\_ALLOC 异常。

所以，内存不足的处理任务就交给类客户去完成。

## 2.2.6 第二级分配器 \_\_default\_alloc\_template

第二级配置器多了一些机制，避免太多小额区块造成内存的碎片。小额区块带来的其实不仅是内存碎片而已，配置时的额外负担（overhead）也是一大问题。所谓的额外负担如下：



这个分配器采用了内存池的思想，有效地避免了内碎片的问题（顺便一句话介绍一下内碎片和外碎片：内碎片是已被分配出去但是用不到的内存空间，外碎片是由于大小太小而无法分配出去的空闲块）。

如果申请的内存块大于 128bytes，就将申请的操作移交 \_\_malloc\_alloc\_template 分配器去处理；如果申请的区块大小小于 128bytes 时，就从本分配器维护的内存池中分配内存。

分配器用空闲链表的方式维护内存池中的空闲空间，为了方便管理，SGI 第二级配置器会主动将任何小额区块的内存需求量上调至 8 的倍数（例如客端要求 30 bytes，就自动调整为 32bytes），并维护 16 个 free-lists，各自管理大小分别为 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120, 128 bytes 的小额区块。free-lists 的节点结构如下：

```
union obj {
    union obj * free_list_link;
    char client_data[1]; /* The client sees this. */
};
```

我们可能会想，为了维护链表，每个节点需要额外的指针（指向下一节点），但是这里用的是 union，从第一个字段看，obj 可被视为指针，指向相同形式的另一个 obj，从第二个字段看，指向实际的区块。一物二用，不会为了维护串行所必须的指针而造成内存浪费。

第二级分配器的部分实现：

```
enum {__ALIGN = 8}; //小型区块的上调边界
enum {__MAX_BYTES = 128}; //小型区块的上限
enum {__NFREELISTS = __MAX_BYTES / __ALIGN}; //free-lists 个数

//以下是第二级配置器。
```



```

//注意，无「template 型别参数」，且第二参数完全没派上用场。
//第一参数用于多线程环境下。本书不讨论多线程环境。
template <bool threads, int inst>
class __default_alloc_template {

private:
    // ROUND_UP() 将 bytes 上调至 8 的倍数。
    static size_t ROUND_UP(size_t bytes) {
        return (((bytes) + __ALIGN-1) & ~(__ALIGN - 1));
    }
private:
    unionobj { //free-lists 的节点构造
        union obj * free_list_link;
        char client_data[1]; /* The client sees this. */
    };
private:
    // 16 个 free-lists
    static obj * volatile free_list[__NFREELISTS];
    // 以下函数根据区块大小，决定使用第 n 号 free-list。n 从 1 起算。
    static size_t FREELIST_INDEX(size_t bytes) {
        return (((bytes) + __ALIGN-1)/__ALIGN - 1);
    }

    // 传回一个大小为 n 的对象，并可能加入大小为 n 的其它区块到 free list.
    static void *refill(size_t n);
    // 配置一大块空间，可容纳 nobjs 个大小为 "size" 的区块。
    // 如果配置 nobjs 个区块有所不便，nobjs 可能会降低。
    static char *chunk_alloc(size_t size, int &nobjs);

    // Chunk allocation state.
    static char *start_free; //内存池起始位置。只在 chunk_alloc()中变化
    static char *end_free; //内存池结束位置。只在 chunk_alloc()中变化
    static size_t heap_size;

public:
    static void *allocate(size_t n) { /* 详述于后 */ }
    static void deallocate(void *p, size_t n) { /* 详述于后 */ }
    static void *reallocate(void *p, size_t old_sz, size_t new_sz);
};

//以下是 static data member 的定义与初值设定
template <bool threads, int inst>
char *__default_alloc_template<threads, inst>::start_free = 0;

```

### 2.2.7 空间分配函数 allocate()

### Allocate()的算法描述:

输出：若分配成功，则返回一个内存的地址，否则返回 NULL

### Allocate()的代码实现:

```
// n must be > 0
static void *allocate(size_t n)
{
    obj * volatile * my_free_list;
    obj * result;

    // 大于 128 就呼叫第一级配置器
```

```

If (n > (size_t) __MAX_BYTES)
    return(malloc_alloc::allocate(n));
}
// 寻找 16 个 free lists 中适当的一个
my_free_list = free_list + FREELIST_INDEX(n);
result = *my_free_list;
if (result == 0) {
    // 没找到可用的 free list, 准备重新填充 free list
    void *r = refill(ROUND_UP(n));
    return r;
}
// 调整 free list
*my_free_list = result->free_list_link;
return (result);
};

```

区块子 free\_list 中拔出, 示意图如下:

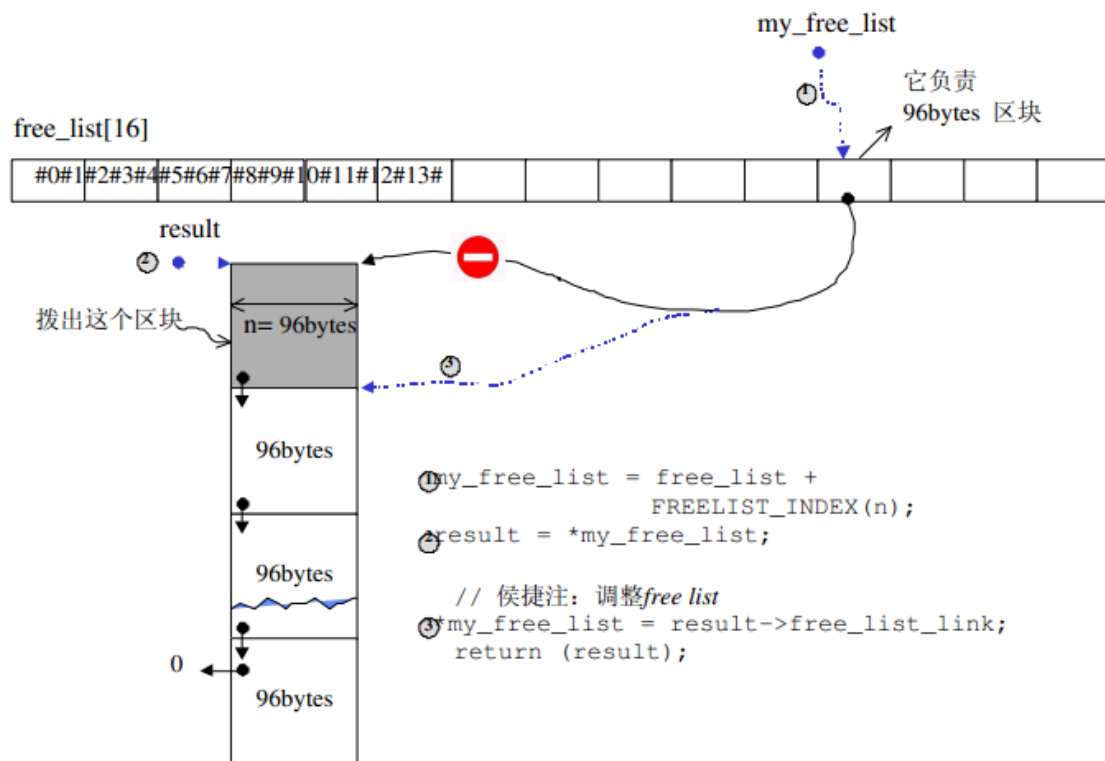


图 2-5 区块自 free list 拔出。阅读次序请循图中编号。

## 2.2.8 空间释放函数 deallocate()

此函数首先判断区块大小, 大于 128 bytes 就呼叫第一级配置器, 小于 128 bytes 就找出对应的 free list, 将区块回收。

### Deallocate() 算法描述:

算法: deallocate

输入: 需要释放的内存块地址 p 和大小 size

```

{
    if(size 大于 128 字节) 直接调用 free(p)释放;
    else{
        将 size 向上取 8 的倍数, 并据此获取对应的空闲列表表头指针 free_list_head;
        调整 free_list_head 将 p 链入空闲列表块中;
    }
}

```

### Deallocate()代码实现:

```

// p 不可以是 0
static void deallocate(void *p, size_t n)
{
    obj *q = (obj *)p;
    obj * volatile * my_free_list;

    // 大于 128 就呼叫第一级配置器
    if (n > (size_t) __MAX_BYTES) {
        malloc_alloc::deallocate(p, n);
        return;
    }
    // 寻找对应的 free list
    my_free_list = free_list + FREELIST_INDEX(n);
    // 调整 free list, 回收区块
    q-> free_list_link = *my_free_list;
    *my_free_list = q;
}

```

区块回收纳入 free list 的动作, 如图 2-6。

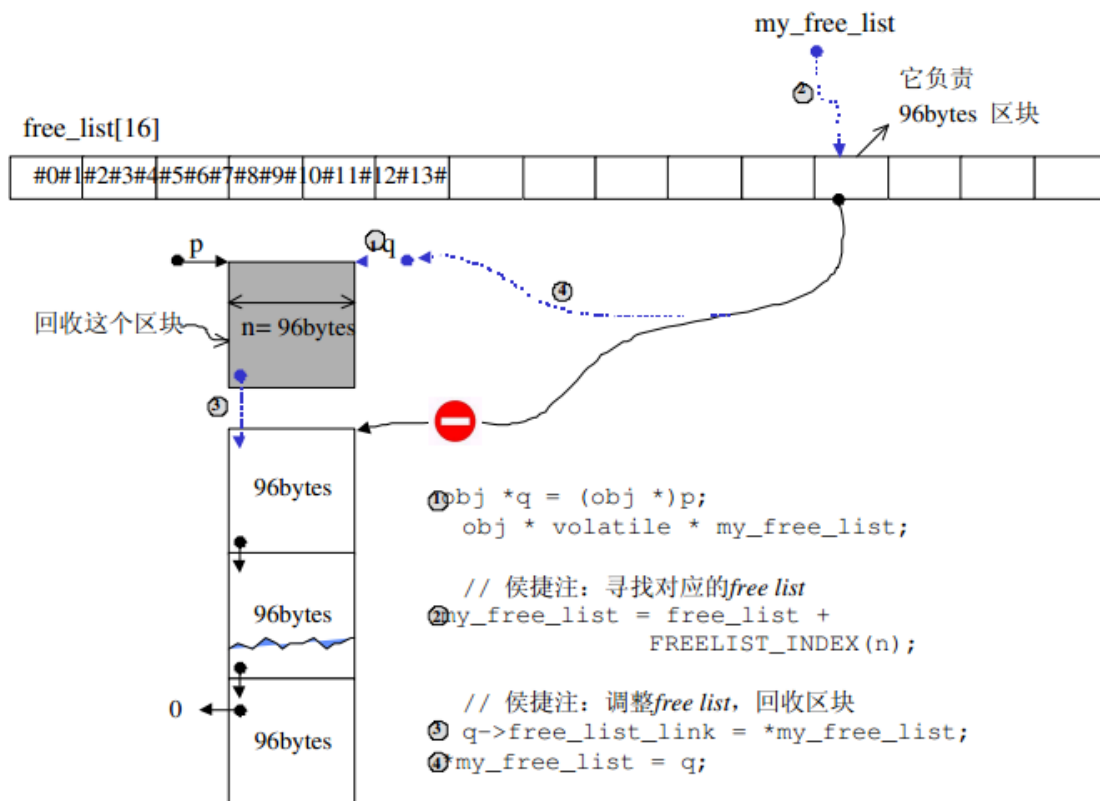


图 2-6 区块回收，纳入 free list。阅读次序请循图中编号。

## 2.2.9 重新填充 free lists

回头讨论先前说过的 `allocate()`。当它发现 free list 中没有可用区块了，就呼叫 `refill()` 准备为 free list 重新填充空间。新的空间将取自内存池（经由 `chunk_alloc()` 完成）。预设去的 20 个新节点（新区块），但万一存储池不足，获得的节点数（区块数）可能小于 20。

### Refill()的算法描述：

算法： `refill`

输入：内存块的大小 `size`

输出：建立空闲块链表并返回第一个可用的内存块地址

```

{
    调用 chunk_alloc 算法分配若干个大小为 size 的连续内存区域并返回起始地址
    chunk 和成功分配的块数 nobj;
    if(块数为 1)直接返回 chunk;
    否则
    {
        开始在 chunk 地址块中建立 free_list;
        根据 size 取 free_list 中对应的表头元素 free_list_head;
        将 free_list_head 指向 chunk 中偏移起始地址为 size 的地址处，即
        free_list_head=(obj*)(chunk+size);
        再将整个 chunk 中剩下的 nobj-1 个内存块串联起来构成一个空闲列表;
        返回 chunk，即 chunk 中第一块空闲的内存块;
    }
}

```

### Refill()的代码实现:

//传回一个大小为 n 的对象, 并且有时候会为适当的 freelist 增加节点.

//假设 n 已经适当上调至 8 的倍数。

template <bool threads, int inst>

void\* \_\_default\_alloc\_template<threads, inst>::refill(size\_t n)

```
{
    int nobjs = 20; //预设取得 20 个区块
    // 呼叫 chunk_alloc(), 尝试取得 nobjs 个区块做为 free list 的新节点。
    // 注意参数 nobjs 是 pass by reference。
    char * chunk = chunk_alloc(n, nobjs); //下节详述

    obj * volatile * my_free_list;
    obj * result;
    obj * current_obj, * next_obj;
    int i;

    // 如果只获得一个区块, 这个区块就直接返回给调用者, free list 无新节点。
    if (1 == nobjs) return(chunk);
    // 否则准备调整 free list, 纳入新节点。
    my_free_list = free_list + FREELIST_INDEX(n);

    // 以下在 chunk 空间内建立 freelist
    result = (obj *)chunk; //这一块 (第 0 块) 准备传回给客户端
    // 以下导引 free list 指向新配置的空间 (取自记忆池), 因为第 0 个将返回给 // 调用者, 所以这里 my_free_list 的第一个元素是 chunk+n
    *my_free_list = next_obj = (obj *) (chunk + n);

    // 以下将 free list 的各节点串接起来。
    for (i = 1; ; i++) { //从 1 开始, 因为第 0 个将传回给客户端
        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);
}
```

### 2.2.10 存储池 (memory pool)

从存储池中取出空间给 free list 使用, 是 chunk\_alloc 的工作。

### Chunk\_alloc 的算法描述:

算法: chunk\_alloc

输入: 内存块的大小 size, 预分配的内存块数 nobj(以引用传递)

输出: 一块连续的内存区域的地址和该区域内可以容纳的内存块的块数

```
{
    计算总共所需的内存大小 total_bytes;
    if(内存池中足以分配, 即 end_free - start_free >= total_bytes) {
        则更新 start_free;
        返回旧的 start_free;
    } else if(内存池中不够分配 nobj 个内存块, 但至少可以分配一个){
        计算可以分配的内存块数并修改 nobj;
        更新 start_free 并返回原来的 start_free;
    } else { //内存池连一块内存块都分配不了
        先将内存池的内存块链入到对应的 free_list 中后 (充分利用内存池中不够分配的
        剩余零头, 添加到相应的 free_list 中);
        调用 malloc 操作重新分配内存池, 大小为 2 倍的 total_bytes 加附加量, start_free
        指向返回的内存地址;
        if(分配不成功) {
            if(16 个空闲列表中尚有空闲块)
                尝试将 16 个空闲列表中空闲块回收到内存池中再调用 chunk_alloc(size,
                nobj);
            else {
                调用第一级分配器尝试 out of memory 机制是否还有用;
            }
        }
        更新 end_free 为 start_free+total_bytes, heap_size 为 2 倍的 total_bytes;
        调用 chunk_alloc(size,nobj);
    }
}
```

### Chunk\_alloc 的代码实现:

//假设 size 已经适当上调至 8 的倍数。

//注意参数 nobjs 是 pass by reference。

```
template <bool threads, int inst>
```

```
char*
```

```
__default_alloc_template<threads, inst>::
```

```
chunk_alloc(size_t size, int& nobjs)
```

```
{
```

```
    char * result;
```

```
    size_t total_bytes = size * nobjs;
```

```
    size_t bytes_left = end_free - start_free; // 记忆池剩余空间
```

```
    if (bytes_left >= total_bytes) {
```

```

// 记忆池剩余空间完全满足需求量。
    result = start_free;
    start_free += total_bytes;
    return(result);
} else if (bytes_left >= size) {
// 记忆池剩余空间不能完全满足需求量，但足够供应一个（含）以上的区块。
    nobjs = bytes_left/size;
    total_bytes = size * nobjs;
    result = start_free;
    start_free += total_bytes;
    return(result);
} else {
// 记忆池剩余空间连一个区块的大小都无法提供。
    size_t bytes_to_get = 2 * total_bytes + ROUND_UP(heap_size >> 4);
// 以下试着让记忆池中的残余零头还有利用价值。
    if (bytes_left > 0) {
// 记忆池内还有一些零头，先配给适当的 free list。
// 首先寻找适当的 free list。
        obj * volatile * my_free_list =
            free_list + FREELIST_INDEX(bytes_left);
// 调整 free list，将记忆池中的残余空间链入。
        ((obj *)start_free) -> free_list_link = *my_free_list;
        *my_free_list = (obj *)start_free;
    }

// 从 heap 分配空间，用来注入记忆池。
start_free = (char *)malloc(bytes_to_get);
if (0 == start_free) {
// heap 空间不足，malloc() 失败。
    int i;
    obj * volatile * my_free_list, *p;
// 试着检视我们手上拥有的东西。这不会造成伤害。我们打算尝试配置
// 较小的区块，因为那在多进程（multi-process）机器上容易导致灾难
// 以下搜寻适当的 free list，
// 所谓适当是指「尚有空闲区块，且区块够大」之 free list。
    for (i = size; i <= __MAX_BYTES; i += __ALIGN) {
        my_free_list = free_list + FREELIST_INDEX(i);
        p = *my_free_list;
        if (0 != p) { // free list 内尚有空闲区块。
            // 调整 free list 以释出空闲区块
            *my_free_list = p -> free_list_link;
            // 重新分配的内存区的起始地址和终止地址
            start_free = (char *)p;
            end_free = start_free + i;

```



```

        // 递归调用自己，为了修正 nobjs。
        return(chunk_alloc(size, nobjs));
        // 注意，任何残余零头终将被编入适当的 free-list 中备用。
    }
}

end_free = 0; // 如果出现意外（山穷水尽，到处都没内存可用了）
// 呼叫第一级配置器，看看 out-of-memory 机制能否尽点力
start_free = (char *)malloc_alloc::allocate(bytes_to_get);
// 第一级存储分配器有 out-of-memory 处理机制（类似 new-handler 机制），
或许能够使内存不足的情况获得改善，否则抛出 bad_alloc 异常（exception）。
}
//调用 malloc 从 heap 成功分配
heap_size += bytes_to_get;
end_free = start_free + bytes_to_get;
// 递归呼叫自己，为了修正 nobjs。
return(chunk_alloc(size, nobjs));
}
}

```

上述的 `chunk_alloc()` 函数以 `end_free - start_free` 来判断存储池的可用空间，如果可用空间充足，就拨出 20 个区块给 free list。否则，如果不够 20 个区块的空间，但是还足够供应 1 个以上的区块空间，就拨出这剩余的不足 20 个区块的空间，此时，`nobjs` 参数将被修改为实际能够供应的区块数。如果存储池中连一个区块空间都无法供应，则利用 `malloc()` 从 heap 中分配，为存储池注入活水源头以应付需求。如果从 heap 中分配失败，则查找 16 个 free list 中可用的空闲块作为存储区，并从中分配。

#### 2.2.11 对外提供的分配器接口

SGI STL 为了方便用户访问，为上面提到的两种分配器包装了一个接口，该接口如下：



```

1 template<class _Tp, class _Alloc>
2 class simple_alloc {
3
4 public:
5     static _Tp* allocate(size_t __n)
6         { return 0 == __n ? 0 : (_Tp*) _Alloc::allocate(__n * sizeof(_Tp)); }
7     static _Tp* allocate(void)
8         { return (_Tp*) _Alloc::allocate(sizeof(_Tp)); }
9     static void deallocate(_Tp* __p, size_t __n)
10        { if (0 != __n) _Alloc::deallocate(__p, __n * sizeof(_Tp)); }
11    static void deallocate(_Tp* __p)
12        { _Alloc::deallocate(__p, sizeof(_Tp)); }
13 };


```



用户调用分配器的时候，为 `simple_alloc` 的第二个模板参数传递要使用的分配器。


### 2.2.12 用户使用分配器的方式

下面是 vector 使用 STL 分配器的代码



```
1 template <class _Tp, class _Alloc>
2 class _Vector_base { //STL vector 的基类
3 public:
4     typedef _Alloc allocator_type;
5     allocator_type get_allocator() const { return allocator_type(); }
6
7     _Vector_base(const _Alloc&)
8         : _M_start(0), _M_finish(0), _M_end_of_storage(0) {}
9     _Vector_base(size_t __n, const _Alloc&)
10        : _M_start(0), _M_finish(0), _M_end_of_storage(0)
11    {
12        _M_start = _M_allocate(__n);
13        _M_finish = _M_start;
14        _M_end_of_storage = _M_start + __n;
15    }
16
17    ~_Vector_base() { _M_deallocate(_M_start, _M_end_of_storage -
18        _M_start); }
19
20 protected:
21     _Tp* _M_start;
22     _Tp* _M_finish;
23     _Tp* _M_end_of_storage;
24
25     typedef simple_alloc<_Tp, _Alloc> _M_data_allocator;
26     _Tp* _M_allocate(size_t __n)
27     { return _M_data_allocator::allocate(__n); }
28     void _M_deallocate(_Tp* __p, size_t __n)
29     { _M_data_allocator::deallocate(__p, __n); }
30 };

```



我们可以看到 vector 的基类调用 simple\_alloc 作为其分配器

### 2.3 内存基本处理工具

STL 中定义有五个全局域函数，前面两个是 construct() 和 destroy(), 另三个是 uninitialized\_copy(), uninitialized\_fill(), uninitialized\_fill\_n(), 分别对应于高阶函数 copy(), fill(), fill\_n() --- 这些都是 STL 算法, SGI 将他们定义在 <stl\_uninitialized>.

### 2.3.1 uninitialized\_copy

uninitialized\_copy()像下面的样子:

```
1 template <class _InputIterator, class _ForwardIterator>
2 inline _ForwardIterator
3   uninitialized_copy(_InputIterator__first, _InputIterator__last
4                     _ForwardIterator __result)
5 {
6   return __uninitialized_copy(__first, __last, __result,
7                               __VALUE_TYPE(__result));
8 }
```

首先萃取出迭代器 result 的 value type (详见第三章), 然后判断该型别是否为 POD 型别:

```
template <class InputIterator, class ForwardIterator, class T>
inline ForwardIterator
__uninitialized_copy(InputIterator first, InputIterator last,
                    ForwardIterator result, T*) {
    typedef typename __type_traits<T>::is_POD_type is_POD;
    return __uninitialized_copy_aux(first, last, result, is_POD());
    // 以上, 企图利用 is_POD() 所获得的结果, 让编译器做自变量推导。
}
```

//如果是 POD 型别, 执行流程就会转进到以下函式。这是藉由 function template //的自变量推导机制而得。

```
template <class InputIterator, class ForwardIterator>
inline ForwardIterator
__uninitialized_copy_aux(InputIterator first, InputIterator last,
                        ForwardIterator result, __true_type) {
    return copy(first, last, result); //呼叫 STL 算法 copy()
}
```

// 如果是 non-POD 型别, 执行流程就会转进到以下函式。这是藉由 function template

//的自变量推导机制而得。

```
template <class InputIterator, class ForwardIterator>
ForwardIterator
__uninitialized_copy_aux(InputIterator first, InputIterator last,
                        ForwardIterator result, __false_type) {
    ForwardIterator cur = result;
    // 为求阅读顺畅, 以下将原 A 该有的异常处理 (exception handling) 省略。
    for ( ; first != last; ++first, ++cur)
        construct(&*cur, *first); //必须一个一个元素地建构, 无法批量进行
    return cur;
}
```

```

}
}

```

针对 `char*` 和 `wchar_t*` 两种型别，可以最具效率的作法 `memmove`（直接搬移内存内容）来执行复制行为。因此 SGI 得以为这两种型别设计一份特化版 A。

//以下是针对 `const char*` 的特化版 A

```

inline char* uninitialized_copy(const char* first, const char* last,
                                char* result) {
    memmove(result, first, last - first);
    return result + (last - first);
}

```

//以下是针对 `const wchar_t*` 的特化版 A

```

inline wchar_t* uninitialized_copy(const wchar_t* first, const
wchar_t* last, wchar_t* result) {
    memmove(result, first, sizeof(wchar_t) * (last - first));
    return result + (last - first);
}

```

如果作为输出目的地的 `[result, result + (last - first))` 范围内的每一个迭代器都指向未初始化的区域，则 `uninitialized_copy()` 会使用 `copy constructor` 为身为输入来源之 `[__first, __last)` 范围内的每一个对象产生一分副本，放入输出范围中。即针对输入范围内的每一个迭代器 `i`，此函数会调用 `construct(&*(result + (i - first)), *i)`，产生 `*i` 的复制品，放置于输出范围的相对位置上。

它调 `__uninitialized_copy(__first, __last, __result, __VALUE_TYPE(__result))` 会判断迭代器 `_result` 所指的对象是否是 POD 类型（POD 类型是指拥有 `constructor`, `destructor`, `copy`, `assignment` 函数的类），如果是 POD 类型，则调用算法库的 `copy` 实现；否则遍历迭代器 `_first ~ _last` 之间的元素，在 `_result` 起始地址处一一构造新的元素。

使用方式：

分配足以包含范围内所有元素的内存区块；

使用 `uninitialized_copy()`，在该内存区块上构造元素；

### 2.3.2 Uninitialized\_fill

```

1 template <class _ForwardIter, class _Tp>
2 inline void uninitialized_fill(_ForwardIter __first,
3                               _ForwardIter __last,
4                               const _Tp& __x)
5 {
6     __uninitialized_fill(__first, __last, __x,
7                         __VALUE_TYPE(__first));
7 }

```

首先萃取出迭代器 `result` 的 `value type`（详见第三章），然后判断该型别是否为 POD

型别:

```
template <class ForwardIterator, class T, class T1>
inline void __uninitialized_fill(ForwardIterator first,
ForwardIterator last, const T& x, T1*) {
    typedef typename __type_traits<T1>::is_POD_type is_POD;
    __uninitialized_fill_aux(first, last, x, is_POD());
}

//如果是 POD 型别, 执行流程就会转进到以下函式。这是藉由 function template
//的自变量推导机制而得。
template <class ForwardIterator, class T>
inline void
__uninitialized_fill_aux(ForwardIterator first, ForwardIterator
                        last, const T& x, __true_type)
{
    fill(first, last, x); //呼叫 STL 算法 fill()
}

// 如果是 non-POD 型别, 执行流程就会转进到以下函式。这是藉由 function
//的自变量推导机制而得。
template <class ForwardIterator, class T>
void
__uninitialized_fill_aux(ForwardIterator first, ForwardIterator
                        last, const T& x, __false_type)
{
    ForwardIterator cur = first;
    // 为求阅读顺畅, 以下将原 A 该有的异常处理 (exception handling) 省略。
    for ( ; cur != last; ++cur)
        construct(&*cur, x); //必须一个一个元素地建构, 无法批量进行
}
```

`uninitialized_fill()` 会将迭代器 `_first` 和 `_last` 范围内的所有元素初始化为 `x`。`Uninitialized_fill()` 会针对操作范围内的每个迭代器 `i`, 调用 `construct(&*i, x)`, 在 `i` 所指之处产生 `x` 的复制品。它调用的 `__uninitialized_fill(__first, __last, __x, __VALUE_TYPE(__first))` 会判断迭代器 `_first` 所指的对象是否是 POD 类型的, 如果是 POD 类型, 则调用算法库的 `fill` 实现; 否则一一构造。

### 2.3.3 uninitialized\_fill\_n

`uninitialized_fill_n()` 会将迭代器 `_first` 开始处的 `n` 个元素初始化为 `x`。 `[__first, __first+n)` 范围内的每个迭代器 `i`, `uninitialized_fill_n()` 都会调用 `construct(&*i, x)`, 在对应位置处产生 `x` 的副本。它调用的 `__uninitialized_fill_n(__first, __n, __x, __VALUE_TYPE(__first))` 会判断迭代器 `_first` 所指对象是否是 POD 类型, 如果是, 则调用算法库的 `fill_n` 实现; 否则一一构造。

```

1 template <class _ForwardIter, class _Size, class _Tp>
2 inline _ForwardIter
3 uninitialized_fill_n(_ForwardIter __first, _Size __n, const _Tp&
__x)
4 {
5     return __uninitialized_fill_n(__first, __n, __x,
__VALUE_TYPE(__first));
6 }

```

首先萃取迭代器\_\_first的value\_type,然后判断该类型是否为POD类型(plain old data, 可以理解为和c兼容的类型):

```

template <class ForwardIterator, class Size, class T, class T1>
inline ForwardIterator __uninitialized_fill_n(ForwardIterator first
, Size n, const T& x, T1*)
{
    // 以下 __type_traits<> 技法, 详见 3.7 节
    typedef typename __type_traits<T1>::is_POD_type is_POD;
    return __uninitialized_fill_n_aux(first, n, x, is_POD());
}

```

对 POD 型别采取最有效率的初值填写手法, 而对 non-POD 型别采取最保险安全的作法:

//如果是 POD 型别, 执行流程就会转进到以下函式。这是藉由 function template //的自变量推导机制而得。

```

template <class ForwardIterator, class Size, class T>
inline ForwardIterator
__uninitialized_fill_n_aux(ForwardIterator first, Size n,
const T& x, __true_type) {
    return fill_n(first, n, x); //交由高阶函式执行。见 6.4.2节。
}

```

// 如果不是 POD 型别, 执行流程就会转进到以下函式。这是藉由 function template //的自变量推导机制而得。

```

template <class ForwardIterator, class Size, class T>
ForwardIterator
__uninitialized_fill_n_aux(ForwardIterator first, Size n,
const T& x, __false_type) {
    ForwardIterator cur = first;
    // 为求阅读顺畅, 以下将原本该有的异常处理 (exception handling) 省略
    for ( ; n > 0; --n, ++cur)
        construct(&*cur, x); //见 2.2.3节
    return cur;
}

```

Uninitialized\_copy(),uninitialized\_fill(),uninitialized\_fill\_n()都具有“commit

or rollback”语义：要不就产生所有必要的元素，否则就不产生任何元素。如果任何一个 copy constructor 丢出异常，他们都必须析构已产生的所有元素。

### 第 3 章 迭代器概念与 traits 编程技法

STL 的中心思想在于，将数据容器和算法分开，彼此独立设计，最后再以一粘着剂将他们撮合在一起。容器和算法的泛型化，可以分别依赖于 c++ 的 class templates 和 function templates. 如何设计出良好的粘着剂才是大难题。

下面以 find() 算法为例，展示容器，算法，迭代器的合作。

//摘自 SGI <stl\_algo.h>

```
template <class InputIterator, class T>
```

```

InputIterator find(InputIterator first,
                  InputIterator last,
                  const T& value) {
    while (first != last && *first != value)
        ++first;
    return first;
}

```

只要给予不同的迭代器，find()便能够对不同的容器做搜寻动作：

// file : 3find.cpp

```
#include <vector>
```

```
#include <list>
```

```
#include <deque>
```

```
#include <algorithm>
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    const int arraySize = 7;
```

```
    int ia[arraySize] = { 0,1,2,3,4,5,6 };
```

```
    vector<int> ivect(ia, ia+arraySize);
```

```
    list<int> ilist(ia, ia+arraySize);
```

```
    deque<int> ideque(ia, ia+arraySize); //注意：VC6[x], 符合标准
```

```
    vector<int>::iterator it1 =find(ivect.begin(), ivect.end(), 4);
```

```
    if (it1 == ivect.end())
```

```
        cout << "4 not found." << endl;
```

```
    else
```

```
        cout << "4 found. " << *it1 << endl;
```

```
// 执行结果： 4 found. 4
```

```
    list<int>::iterator it2 =find(ilist.begin(), ilist.end(), 6);
```

```
    if (it2 == ilist.end())
```

```
        cout << "6 not found." << endl;
```

```
    else
```

```
        cout << "6 found. " << *it2 << endl;
```

```
// 执行结果： 6 found. 6
```

```
    deque<int>::iterator it3 =find(ideque.begin(), ideque.end(), 8);
```

```
    if (it3 == ideque.end())
```

```
        cout << "8 not found." << endl;
```

```
    else
```



```

        cout << "8 found. " << *it3 << endl;
    // 执行结果: 8 not found.
}

```

### 3.2 迭代器 iterator 是一种 smart pointer

迭代器类似指针，指针中最重要的是\*和->操作符，因此迭代器最重要的编程工作就是对operator\*和operator->进行重载。

下面以 list 为例说明迭代器的原理。

//List 节点定义

```

Template<class T>
Struct __list_node {
    Typedef void *void_pointer;
    Void_pointer next;
    Void_pointer prev;
    T data;
};

```

//List 迭代器定义

```

Template<class T, class Ref, class Ptr>
Struct __list_iterator {
    //这三个 typedef 是为了简化后面的代码书写
    Typedef __list_iterator<T, T&, T*>    iterator;
    Typedef __list_iterator<T, const T&, const T*>    const_iterator;
    Typedef __list_iterator<T, Ref, Ptr>    self;?? ??

    Typedef bidirectional_iterator_tag    iterator_category;// 迭代器 属于 bidirectional
iterator
    Typedef T value_type;//值类型
    Typedef Ptr pointer;//指针类型
    Typedef Ref reference;//引用类型
    Typedef __list_node<T>* link_type;//节点指针类型
    Typedef size_t size_type;
    Typedef ptrdiff_t difference_type;

```

Link\_type node;//迭代器当前所指的节点,保持与容器之间的联系

//三种构造函数

```

__list_iterator(link_type x):node(x){}
__list_iterator(){}
__list_iterator(const iterator& x):node(x.node){}

```

//==和!=操作符重载

```

Bool operator==(const self& x) const {return node == x.node;}

```

```
Bool operator!=(const self& x) const{return node != x.node;}
```

**//不必提供 copy ctor 和 operator=, 因为编译器提供的预设行为已足够**

**//操作符, 获取所指节点中的数据**

```
Reference operator*() const {return (*node).data;}
```

```
Pointer operator->() const {return &(operator*());}
```

**//前置++操作符, 指向下一个节点**

```
Self& operator++(){  
    Node = (link_type)((*node).next);  
    Return *this;  
}
```

**//后置++操作符, 指向下一个节点**

```
Self operator++(int){  
    Self tmp = *this;  
    ++*this;  
    Return tmp;  
}
```

**//前置--操作符, 指向前一个节点**

```
Self& operaotr--(){  
    Node = (link_type)((*node).prev);  
    Return *this;  
}
```

**//后置--操作符, 指向前一个节点**

```
Self operator--(int){  
    Self tmp = *this;  
    --*this;  
    Return tmp;  
}  
}
```

```
Template<class T, class Alloc = alloc>
```

```
Class list {
```

```
...
```

```
...
```

```
Public:
```

```
    Typedef __list_iterator<T, T&, T*>    iterator;//
```

```
    Typedef __list_iterator<T, const T&, const T*> const_iterator;
```

```
...
```

```
...
```

Protected:

Link\_type node;//头结点, 该 list 其实就是一个带头结点的双向循环链表

Public:

List(){empty\_initialize();}

Iterator begin{return (link\_type)((node\*).next);}//返回头结点的下一个节点,即第一个节点的 iterator

Const iterator begin() const{return (link\_type)((node\*).next);}

Iterator end(){return node;}//返回头结点的 iterator, 其实就是链表的结尾

Const iterator end() const {return node;}

...

...

}

如果我们对 List 容器使用 find 算法, 这一过程中会发生什么?

```
int a[] = {1,2,3,4,5};
```

```
list<int> l(a, a+5);
```

```
list<int>::iterator it = find(l.begin(), l.end(), 3);
```

```
cout << *it << end;
```

先看看 find 函数的定义:

```
template <class InputIterator, class T>
```

```
InputIterator find(InputIterator first, InputIterator last, const T& value) {
```

```
    while (first != last && *first != value) ++first;
```

```
    return first;
```

```
}
```

我们所调用的 find 函数的特化版本其实是:

```
find<__list_iterator<int, int&, int*>, int>(__list_iterator<int, int&, int*> first, __list_iterator<int, int&, int*> last, const int& value)
```

从而 find 函数中所用到的!=、\*、++等操作符都作用在\_\_list\_iterator<int, int&, int\*>的身上, 这正是泛型的作用所在。

### 3.3 迭代器的相应型别

在算法之中用迭代器时, 很可能会用到其相应的型别。迭代器所指之物的型别便是其一。假设算法需要声明一个变量, 以“迭代器所指对象的型别”为类型, 该怎么办呢? C++不支持 typeid!

解决办法就是: 利用 function template 的自变量推导 (argument deduction) 机制, 例如:

```
template <class I, class T>
```

```
void func_impl(I iter, T t)
```

```
{
```

```
    T tmp; // 这里解决了问题。T 就是迭代器所指之物的型别, 本例为 int
```

```
    // ... 这里做原 A func()应该做的全部工作
```

```
};

template <class I>
inline
void func(I iter)
{
    func_impl(iter,*iter);// func 的工作全部移往 func_impl
}

int main()
{
    int i;
    func(&i);
}
```

由于 func\_impl()是一个 function template，一旦被呼叫，编译器会自动进行 template 自变量推导。于是导出型别 T，顺利解决了问题。

然而迭代器型别常用的有五种，并不是每一种都可以利用上述的 template 自变量推导机制获取，我们需要更全面的解法---traits。

### 3.4 traits 编程技法---STL 源码门钥

迭代器所指物件的型别，称为该迭代器的 value type。前面的 function template argument

Deduction 机制虽然可以用于 value type，但是并非万能：万一 value type 作为函数的返回值，就束手无策，因为 template 自变量推导机制推导的只有自变量，无法推导函数的返回值。

Partial specification（部分特化/偏特化）：

如果 class template 拥有一个以上的 template 参数，我们可以针对其中数个（一个或多个，但非全部）template 参数进行特化工作。也即我们可以在泛化设计中提供特化版本（将 template 参数做进一步的限制，所设计出来的一个特化版本）。

如：

```
Template<typename T>
Class C {...};//这个泛化版本允许接收 T 为任何型别
```

```
Template<typename T>
Class C<T*> {...};//这个特化版本，仅适用于 T 为原生指针（即 c 指针）的情况，T 为原生指针就是 T 为任何类型的更进一步条件限制
```

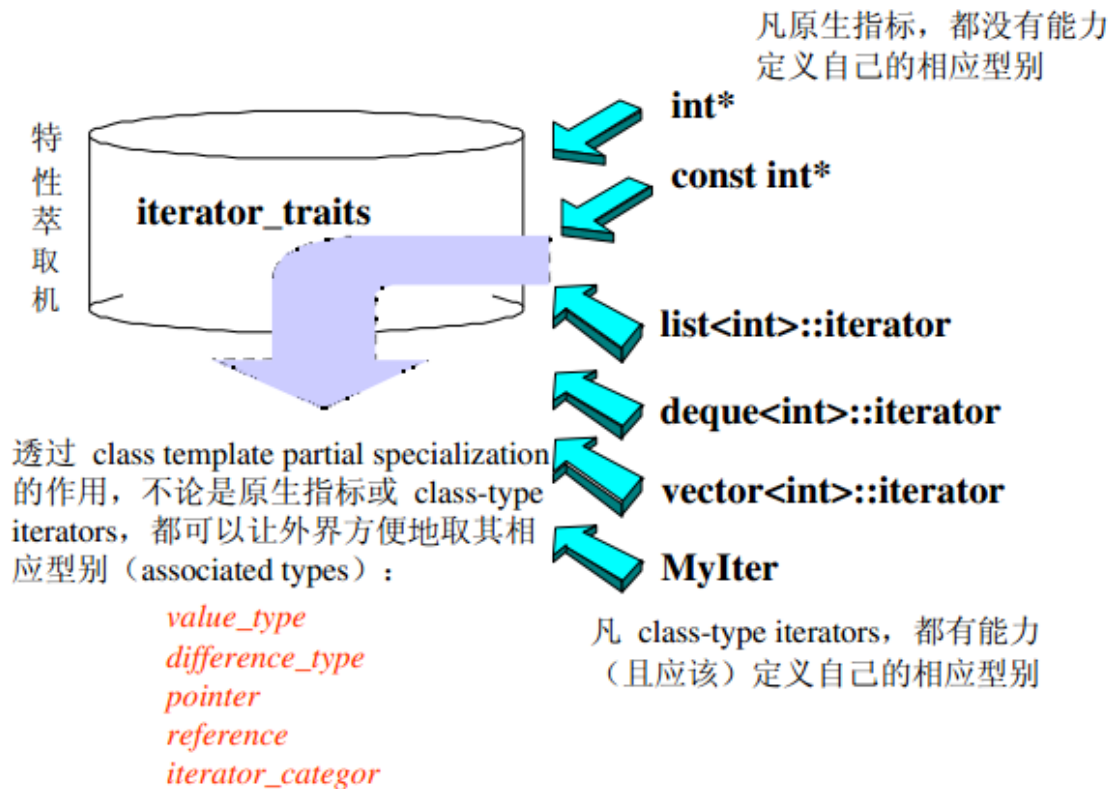


图 3-1 traits 就像一台“特性萃取机”，萃取各个迭代器的特性（相应型别）。注：图示中的原生指标即原生指针（它也是作为 iterator 的一种），他不能通过嵌套型别定义的方式取出相应型别；class-type iterators 可以通过嵌套型别定义(所谓的嵌套型别定义就是在相应的 iterator 类定义中对于模板参数 T 做形如：typedef T value\_type 的定义)的方式取出相应型别。但是经过 class template partial specification 所有类型的 iterator 都可以获取型别了。

STL 的迭代器在使用的时候需要了解各种迭代器的特性。主要特性包含以下几种：

- 1、iterator\_category：表示迭代器所属的类型
- 2、value\_type：表示迭代器所指对象类型的类型
- 3、difference\_type：表示两个迭代器之间的距离类型
- 4、pointer：表示迭代器所指数据的指针类型
- 5、reference：表示迭代器所指数据的引用类型

通常迭代器的几种特性被放在 iterator\_traits 中。

// 对所有 Iterator 的泛化

```
template <class Iterator>
```

```
struct iterator_traits {
```

```
    typedef typename Iterator::iterator_category iterator_category;
```

```
    typedef typename Iterator::value_type          value_type;
```

```
    typedef typename Iterator::difference_type     difference_type;
```

```
    typedef typename Iterator::pointer            pointer;
```

```
    typedef typename Iterator::reference          reference;
```

```

};

// 对指针类型的偏特化 (Partial Specialization)
template <class T>
struct iterator_traits<T*> {
    typedef random_access_iterator_tag iterator_category;    // 指针类型是可以随机访问的
    typedef T value_type;                                    // 值类型
    typedef ptrdiff_t difference_type;                      // 指针类型之间的距离
    // 一定是整型 (ptrdiff_t 被定义为 int 型)
    typedef T* pointer;
    typedef T& reference;
};

template <class T>
struct iterator_traits<const T*> {                          // 同上，只不过这里是常量指针
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef const T* pointer;
    typedef const T& reference;
};

```

### 3.4.1 value type

```

Template<class I>
Struct iterator_traits {
    ...
    Typedef typename I::value_type value_type;
}

```

//针对原生指针而设计的「偏特化 (partial specialization)」版

```

Template <class T>
Struct iterator_traits<T*>{//偏特化版
    ...
    Typedef T value_type;
}

```

### 3.4.2 difference type

**difference type** 用来表示两个迭代器之间的距离，也因此，它可以用来表示一个容器的最大容量，因为对于连续空间的容器而言，头尾之间的距离就是其最大容量。如果一个泛型算法提供计数功能，例如 STL 的 `count()`，其传回值就必须使用迭代器的 **difference type**：

```

template <class I, class T>
typename iterator_traits<I>::difference_type //这一整行是函数返回类型

```

```

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;
}

```

针对相应型别 difference type，traits 的两个（针对原生指针而写的）特化版本如下，以 C++ 内建的 ptrdiff\_t（定义于 <cstdlib> 头文件）做为原生指针的 difference type:

```

template <class I>
struct iterator_traits {
    ...
    typedef typename I::difference_type    difference_type;
};

```

//针对原生指标而设计的「偏特化（partial specialization）」版

```

template <class T>
struct iterator_traits<T*> {
    ...
    typedef ptrdiff_t    difference_type;
};

```

//针对原生的 pointer-to-const 而设计的「偏特化（partial specialization）」版

```

template <class T>
struct iterator_traits<const T*> {
    ...
    typedef ptrdiff_t    difference_type;
};

```

现在，任何时候当我们需要任何迭代器 I 的 difference type，可以这么写：

```

typename iterator_traits<I>::difference_type

```

### 3.4.3 reference type

### 3.4.4 pointer type

```

template <class I>
struct iterator_traits {
    ...
    typedef typename I::pointer    pointer;
    typedef typename I::reference    reference;
};

```

//针对原生指标而设计的「偏特化版（partial specialization）」

```
template <class T>
Struct iterator_traits<T*> {
    ...
    typedef T*    pointer;
    typedef T&    reference;
};
```

//针对原生的 **pointer-to-const** 而设计的「偏特化版 (partial specialization)」

```
template <class T>
Struct iterator_traits<const T*> {
    ...
    typedef const T*    pointer;
    typedef const T&    reference;
};
```

### 3.4.5 iterator\_category

先讨论迭代器的分类：

**Input Iterator**：这种迭代器所指对象，不允许外界改变。只读 (read only)。

**Output Iterator**：唯写 (write only)。

**Forward Iterator**：允许「写入型」算法 (例如 `replace()`) 在此种迭代器所形成的区间上做读写动作。

**Bidirectional Iterator**：可双向移动。某些算法需要逆向走访某个迭代器区间 (例如逆向拷贝某范围内的元素)，就可以使用 **Bidirectional Iterators**。

**Random Access Iterator**：前四种迭代器都只供应一部份指标算术能力 (前三种支持 `operator++`，第四种再加上 `operator--`)，第五种则涵盖所有指标算术能力，包括 `p+n`, `p-n`, `p[n]`, `p1-p2`, `p1<p2`。

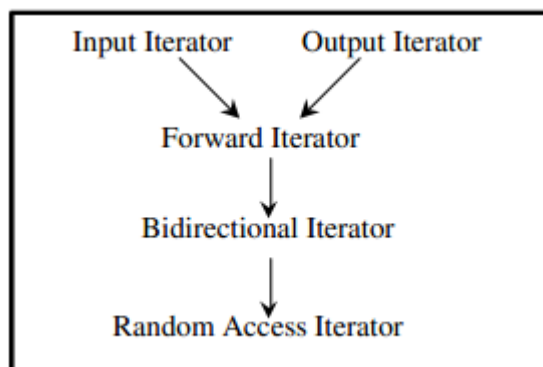


图 3-2 迭代器的分类与从属关系 (注：这里不是继承关系，二是概念与强化关系)

下面以 `advance()` 为例，讨论 **Input Iterator**/**Bidirectional Iterator**/**Random Access Iterator** 设计的版本：

```
template <class InputIterator, class Distance>
inline void advance(InputIterator& i, Distance n) {
    __advance(i, n, iterator_category(i));    // 根据不同的类型调用不同的重载函数
}
```



```

// 针对 input iterator 和 forward iterator 的版本
template <class InputIterator, class Distance>
inline void __advance(InputIterator& i, Distance n, input_iterator_tag) {
    while (n-->0) ++i; // 只能单向移动
}
// 针对 Bidirectional iterator 的版本
template <class BidirectionalIterator, class Distance>
inline void __advance(BidirectionalIterator& i, Distance n,
                     bidirectional_iterator_tag) {
    if (n >= 0) // 根据方向不同有不同的处理
        while (n-->0) ++i;
    else
        while (n++<0) --i;
}
// 针对 Random access iterator 的版本
template <class RandomAccessIterator, class Distance>
inline void __advance(RandomAccessIterator& i, Distance n,
                     random_access_iterator_tag) {
    i += n; // 随机访问，提高效率
}

```

对这里的 `iterator_category`，SGI STL `<stl_iterator.h>` 中源码是：

```

template <class I>
inline typename iterator_traits<I>::iterator_category
iterator_category(const I&) {
    typedef typename iterator_traits<I>::iterator_category category;
    return category();
}

```

为了满足上述行为，traits 必须在增加一个相应的型别： `iterator_category`

```

template <class I>
struct iterator_traits {
    ...
    typedef typename I::iterator_category iterator_category;
};

```

//针对原生指标而设计的「偏特化版 (partial specialization)」

```

template <class T>
struct iterator_traits<T*> {
    ...
    // 注意，原生指标是一种 Random Access Iterator
    typedef random_access_iterator_tag iterator_category;
};

```

//针对原生的 **pointer-to-const** 而设计的「偏特化版 (partial specialization)」

```
template <class T>
```

```
struct iterator_traits<const T*>
```

```
...
```

```
// 注意，原生的 pointer-to-const 是一种 Random Access Iterator
```

```
    typedef random_access_iterator_tag    iterator_category;
```

```
};
```

任何一个迭代器，其类型永远应该落在“该迭代器所属各种类型中，最强化的那个”，例如 `int*`既是 `random access iterator`,又是 `bidirectional iterator`，同时也是 `input iterator`，那么其类型应该归属为 `random_access_iterator_tag`。

### 3.5 std::iterator 的保证

为了符合规范，任何迭代器都应该提供五个相应型别，以利于 `traits` 萃取，否则可能无法与其它 `STL` 组件顺利搭配，为此 `STL` 提供了一个 `iterators` `class`，如果每个新设计的迭代器都继承自它，就保证符合 `STL` 规范：

总的来说，在 `STL` 中是由容器 (`container`) 来负责设计适当的迭代器 (`iterator`)，由迭代器 (`iterator`) 来负责设计适当的迭代器属性。正因为这一点才使得容器和算法可以完全分离开来，通过迭代器提供的接口来访问容器的内部元素。在这里我们又一次看到了 `traits` 编程技巧的强大功能，它利用嵌套型别的编码技巧与编译器的 `template` 自变量推到功能，弥补了 `C++`语言不是强类型语言的不足之处。

```
template <class Category,
          class T,
          class Distance = ptrdiff_t,
          class Pointer = T*,
          class Reference = T&>
```

```
Struct iterator {
```

```
    typedef Category iterator_category;
```

```
    typedef T          value_type;
```

```
    typedef Distance difference_type;
```

```
    typedef Pointer    pointer;
```

```
    typedef Reference reference;
```

```
};
```

`iterator` `class` 不含任何成员，纯粹只是型别定义，所以继承它并不会招致任何额外负担。由于后三个参数皆有默认值，新的迭代器只须提供前两个参数即可。那么 3.2 节中的 `ListIter` 就应该这么写：

```
template <class Item>
```

```
struct ListIter : public std::iterator<std::forward_iterator_tag, Item>
```

```
{ ... }
```

### 3.6 iterator 源码完整重列

完整的`<stl_iterator.h>`头文件中与本章相关的程序代码。

### 3.7 SGI STL 的私房菜: \_\_type\_traits

traits 编程技法很棒, 适度弥补了 C++ 语言本身的不足。STL 只对迭代器加以规范, 制定出 iterator\_traits 这样的东西。SGI 把这种技法进一步扩大到迭代器以外的世界, 于是有了所谓的 \_\_type\_traits。

Iterator\_traits 负责萃取迭代器的特性, \_\_type\_traits 则负责萃取型别的特性, 包括:

1. 是否存在 non-trivial default constructor
2. 是否存在 non-trivial copy constructor
3. 是否存在 non-trivial assignment operator
4. 是否存在 non-trivial destructor

如果答案是否定的, 则我们对这个型别进行构造, 析构, 拷贝和赋值等动作时, 就可以采取最有效率的措施 (例如, 根本不用调用那些尸位素餐的 constructor, destructor), 而采用内存直接动作如 malloc(), memcpy() 等等, 获得更高效率。这对于大规模而动作频繁的容器, 有着显著的效率提升。

从 iterator\_traits 得来的经验, 我们希望, 程式之中可以这样运用 \_\_type\_traits<T>, T 代表任意型别:

```
__type_traits<T>::has_trivial_default_constructor
__type_traits<T>::has_trivial_copy_constructor
__type_traits<T>::has_trivial_assignment_operator
__type_traits<T>::has_trivial_destructor
__type_traits<T>::is_POD_type // POD: Plain Old Data
```

我们希望上述式子响应我们「真」或「假」(以便我们决定采取什么策略), 但其结果不应该只是个 bool 值, 应该是个有着真/假性质的「对象」, 因为我们希望利用其响应结果来进行自变量推导, 而编译器只有面对 class object 形式的自变量, 才会做自变量推导。为此, 上述式子应该传回这样的东西:

```
struct __true_type { };
struct __false_type { };
```

这两个空白 classes 没有任何成员, 不会带来额外负担, 却又能够标示真假, 满足我们所需。下面是 SGI 的作法:

```
template <class type>
struct __type_traits
{
    typedef __true_type      this_dummy_member_must_be_first;

    typedef __false_type     has_trivial_default_constructor;
    typedef __false_type     has_trivial_copy_constructor;
    typedef __false_type     has_trivial_assignment_operator;
    typedef __false_type     has_trivial_destructor;
    typedef __false_type     is_POD_type;
};
```

```
};
```

**问题：为什么把对象的所有的属性都定义为\_\_false\_type?**

这样是采用最保守的做法，先把所有的对象属性都设置为\_\_false\_type，然后在针对每个基本数据类型设计特化的\_\_type\_traits，就可以达到预期的目的，如可以定义\_\_type\_traits<int>如下：

```
/*以下针对 C++基本型别 char, signed char, unsigned char, short,
unsigned short, int, unsigned int, long, unsigned long, float, double,
long double 提供特化版 A。注意，每一个成员的值都是 __true_type，表示这些
型别都可采用最快速方式（例如 memcpy）来进行拷贝（copy）或赋值（assign）动
作。*/
```

```
//注意，SGI STL <stl_config.h>将以下出现的 __STL_TEMPLATE_NULL
//定义为 template<>，见 1.9.1 节，是所谓的 class template explicit specialization
```

```
__STL_TEMPLATE_NULL struct __type_traits<char> {
    typedef __true_type    has_trivial_default_constructor;
    typedef __true_type    has_trivial_copy_constructor;
    typedef __true_type    has_trivial_assignment_operator;
    typedef __true_type    has_trivial_destructor;
    typedef __true_type    is_POD_type;
};
```

```
__STL_TEMPLATE_NULL struct __type_traits<signed char> {
    typedef __true_type    has_trivial_default_constructor;
    typedef __true_type    has_trivial_copy_constructor;
    typedef __true_type    has_trivial_assignment_operator;
    typedef __true_type    has_trivial_destructor;
    typedef __true_type    is_POD_type;
};
```

```
.....
```

```
.....
```

```
__STL_TEMPLATE_NULL struct __type_traits<long double> {
    typedef __true_type    has_trivial_default_constructor;
    typedef __true_type    has_trivial_copy_constructor;
    typedef __true_type    has_trivial_assignment_operator;
    typedef __true_type    has_trivial_destructor;
    typedef __true_type    is_POD_type;
};
```

```
//注意，以下针对原生指标设计 __type_traits 偏特化版 A。
```

```
//原生指标亦被视为一种基本数据类型。
```

```

template <class T>
struct __type_traits<T*> {
    typedef __true_type    has_trivial_default_constructor;
    typedef __true_type    has_trivial_copy_constructor;
    typedef __true_type    has_trivial_assignment_operator;
    typedef __true_type    has_trivial_destructor;
    typedef __true_type    is_POD_type;
};

```

\_\_type\_traits 在 SGI\_STL 中应用举例：

```

template <class ForwardIterator, class Size, class T>
inline ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n, const T& x) {
    return __uninitialized_fill_n(first, n, x, value_type(first));
}

```

```

template <class ForwardIterator, class Size, class T, class T1>
inline ForwardIterator __uninitialized_fill_n(ForwardIterator first, Size n, const T& x, T1*)
{
    typedef typename __type_traits<T1>::is_POD_type is_POD;
    return __uninitialized_fill_n_aux(first, n, x, is_POD());
}

```

以下就「是否为 POD 型别」采取最适当的措施：

//如果不是 POD 型别，就会派送（dispatch）到这里

```

template <class ForwardIterator, class Size, class T>

```

```

ForwardIterator

```

```

__uninitialized_fill_n_aux(ForwardIterator first, Size n,
                           const T& x, __false_type) {

```

```

    ForwardIterator cur = first;

```

// 为求阅读顺畅简化，以下将原 A 有的异常处理（exception handling）去除。

```

    for ( ; n > 0; --n, ++cur)

```

```

    construct(&*cur, x);

```

```

    return cur;
}

```

//见 2.2.3 节

//如果是 POD 型别，就会派送（dispatch）到这里。下两行是原文件所附注解。

//如果 copy construction 等同于 assignment，而且有 trivial destructor，

//以下就有效。

```

template <class ForwardIterator, class Size, class T>

```

```

inline ForwardIterator

```

```

__uninitialized_fill_n_aux(ForwardIterator first, Size n,
                           const T& x, __true_type) {

```

```

    return fill_n(first, n, x); //交由高阶函式执行，如下所示。

```

```
}
```

```
//以下是定义于 <stl_algobase.h> 中的 fill_n()
template <class OutputIterator, class Size, class T>
OutputIterator fill_n(OutputIterator first, Size n, const T& value) {
    for ( ; n > 0; --n, ++first)
        *first = value;
    return first;
}
```

究竟一个 class 什么时候该有自己的 non-trivial default constructor, non-trivial copy constructor, non-trivial assignment operator, non-trivial destructor 呢？一个简单的判断准则是：如果 class 内含指针成员，并且对它进行内存动态配置，那么这个 class 就需要实作出自己的 non-trivial-xxx。

如果编译器够厉害，我们甚至可以不用自己去定义特化的 \_\_type\_traits，编译器就能够帮我们搞定：)

如何使用呢？

假设现在有个模板函数 fun 需要根据类型 T 是否有 non-trivial constructor 来进行不同的操作，可以这样来实现：

```
template<class T>
void fun()
{
    typedef typename __type_traits<T>::has_trivial_constructor _Trivial_constructor;
    __fun(_Trivial_constructor()); // 根据得到的 _Trivial_constructor 来调用相应的函数
}

// 两个重载的函数
void __fun(__true_type )
{ ..... }

void __fun(__false_type )
{ ..... }
```

第 4 章 序列式容器

4.1 容器的概念与分类

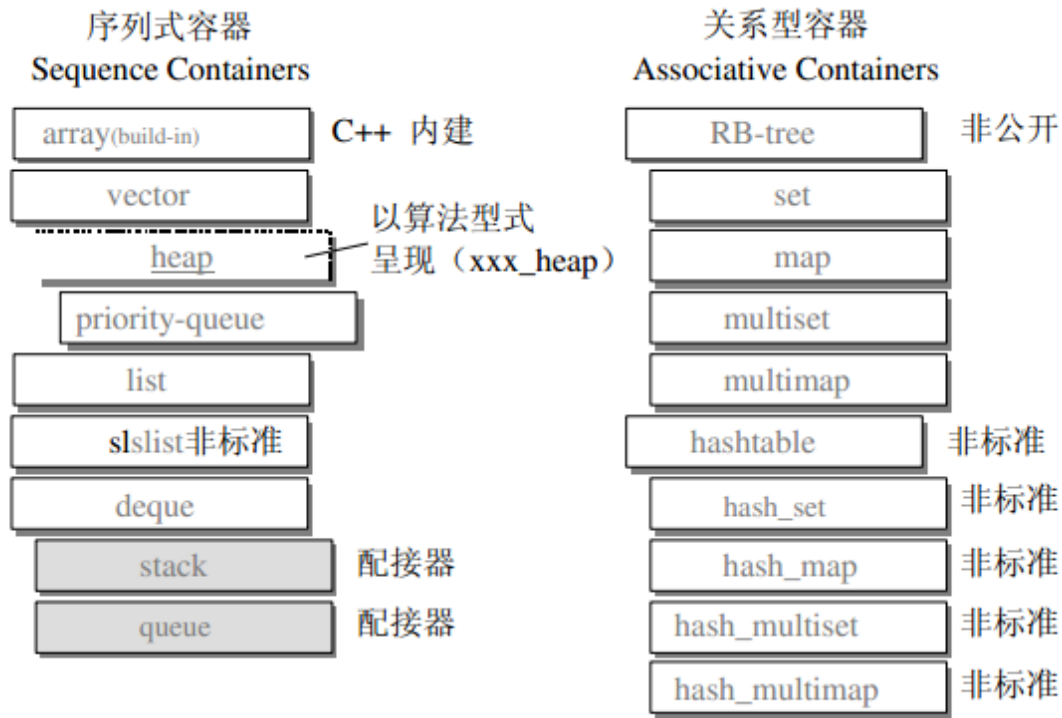


图 4-1 SGI STL 的各种容器。本图以内缩方式来表达基层与衍生层的关系。这里所谓的衍生，并非继承 (inheritance) 关系，而是内含 (containment) 关系。例如 heap 内含一个 vector，priority-queue 内含一个 heap，stack 和 queue 都含一个 deque，set/map/multiset/multimap 都内含一个 RB-tree，hasht\_x都内含一个 hashtable。

4.1.1 序列式容器

序列式容器：其中的元素都可序 (ordered)，但未排序 (sorted)。  
C++语言提供了一个序列式容器：array  
STL 另外再提供 vector, list, deque, stack, queue, priority\_queue 等序列式容器。

4.2 vector

Vector 的数据安排以及操作方式，与 array 非常相似，两者的唯一差别在于空间运用的弹性：array 是静态空间，一旦配置了就不能改变；vector 是动态控件，随着元素的加入，他的内部机制会自行扩充空间以容纳新元素。

Vector 实现关键在于：对空间大小的控制和重新配置时数据搬移效率。

4.2.2 vector 定义摘要

Refer to: <http://blog.csdn.net/hackbuteer1/article/details/7724547>

// alloc 是 SGI STL 的空间配置器，见第二章。

```
1 #include<iostream>
```

```

2  using namespace std;
3  #include<memory.h>
4
5  // alloc 是 SGI STL 的空间配置器
6  template <class T, class Alloc = alloc>
7  class vector
8  {
9  public:
10     // vector 的嵌套类型定义,typedefs 用于提供 iterator_traits<I>支持
11     typedef T value_type;
12     typedef value_type* pointer;
13     typedef value_type* iterator;
14     typedef value_type& reference;
15     typedef size_t size_type;
16     typedef ptrdiff_t difference_type;
17 protected:
18     // 这个提供 STL 标准的 allocator 接口
19     typedef simple_alloc <value_type, Alloc> data_allocator;
20
21     iterator start;           // 表示目前使用空间的头
22     iterator finish;          // 表示目前使用空间的尾
23     iterator end_of_storage;  // 表示实际分配内存空间的尾
24
25     void insert_aux(iterator position, const T& x);
26
27     // 释放分配的内存空间
28     void deallocate()
29     {
30         // 由于使用的是 data_allocator 进行内存空间的分配,
31         // 所以需要同样使用 data_allocator::deallocate()进行释放
32         // 如果直接释放,对于 data_allocator 内部使用内存池的版本
33         // 就会发生错误
34         if (start)
35             data_allocator::deallocate(start, end_of_storage - start);
36     }
37
38     void fill_initialize(size_type n, const T& value)
39     {
40         start = allocate_and_fill(n, value);
41         finish = start + n;           // 设置当前使用内存空间的结束点
42         // 构造阶段,此实作不多分配内存,
43         // 所以要设置内存空间结束点和,已经使用的内存空间结束点相同
44         end_of_storage = finish;
45     }

```



```

46
47 public:
48     // 获取几种迭代器
49     iterator begin() { return start; }
50     iterator end() { return finish; }
51
52     // 返回当前对象个数
53     size_type size() const { return size_type(end() - begin()); }
54     size_type max_size() const { return size_type(-1) / sizeof(T); }
55     // 返回重新分配内存前最多能存储的对象个数
56     size_type capacity() const { return size_type(end_of_storage - begin()); }
57     bool empty() const { return begin() == end(); }
58     reference operator[](size_type n) { return *(begin() + n); }
59
60     // 本实作中默认构造出的 vector 不分配内存空间
61     vector() : start(0), finish(0), end_of_storage(0) {}
62
63
64     vector(size_type n, const T& value) { fill_initialize(n, value); }
65     vector(int n, const T& value) { fill_initialize(n, value); }
66     vector(long n, const T& value) { fill_initialize(n, value); }
67
68     // 需要对象提供默认构造函数
69     explicit vector(size_type n) { fill_initialize(n, T()); }
70
71     vector(const vector<T, Alloc>& x)
72     {
73         start = allocate_and_copy(x.end() - x.begin(), x.begin(), x.end());
74         finish = start + (x.end() - x.begin());
75         end_of_storage = finish;
76     }
77
78     ~vector()
79     {
80         // 析构对象
81         destroy(start, finish);
82         // 释放内存
83         deallocate();
84     }
85
86     vector<T, Alloc>& operator=(const vector<T, Alloc>& x);
87
88     // 提供访问函数
89     reference front() { return *begin(); }

```

```

90     reference back() { return *(end() - 1); }
91
92
93     //////////////////////////////////////
94     // 向容器尾追加一个元素，可能导致内存重新分配
95     //////////////////////////////////////
96     //                                     push_back(const T& x)
97     //                                     |
98     //                                     |----- 容量已满?
99     //                                     |
100    //      -----
101    //      No |                                     | Yes
102    //      |                                     |
103    //      ↓                                     ↓
104    //      construct(finish, x);      insert_aux(end(), x);
105    //      ++finish;                  |
106    //                                     |----- 内存不足，重新分配
107    //                                     |      大小为原来的2倍
108    //      new_finish = data_allocator::allocate(len);      <stl_alloc.h>
109    //      uninitialized_copy(start, position, new_start);
110    //      <stl_uninitialized.h>
111    //      construct(new_finish, x);      <stl_construct.h>
112    //      ++new_finish;
113    //      uninitialized_copy(position, finish, new_finish);
114    //      <stl_uninitialized.h>
115
116    //////////////////////////////////////
117    void push_back(const T& x)
118    {
119        // 内存满足条件则直接追加元素，否则需要重新分配内存空间
120        if (finish != end_of_storage)
121        {
122            construct(finish, x);
123            ++finish;
124        }
125        else
126        {
127            insert_aux(end(), x);
128        }
129    }
130
131    //////////////////////////////////////

```

```

128 // 在指定位置插入元素
129
130 // insert(iterator position, const T& x)
131 // |
132 // |----- 容量是否足够 && 是否是
end()?
133 // |
134 // -----
135 // No | | Yes
136 // | |
137 // ↓ ↓
138 // insert_aux(position, x); construct(finish, x);
139 // | ++finish;
140 // |----- 容量是否够用?
141 // |
142 // -----
143 // Yes | | No
144 // | |
145 // ↓ |
146 // construct(finish, *(finish - 1)); |
147 // ++finish; |
148 // T x_copy = x; |
149 // copy_backward(position, finish - 2, finish - 1); |
150 // *position = x_copy; |
151 // ↓
152 // data_allocator::allocate(len); <stl_alloc.h>
153 // uninitialized_copy(start, position, new_start);
<stl_uninitialized.h>
154 // construct(new_finish, x); <stl_construct.h>
155 // ++new_finish;
156 // uninitialized_copy(position, finish, new_finish);
<stl_uninitialized.h>
157 // destroy(begin(), end()); <stl_construct.h>
158 // deallocate();
159
160 ///////////////////////////////////////////////////////////////////
161 iterator insert(iterator position, const T& x)
162 {
163     size_type n = position - begin();
164     if (finish != end_of_storage && position == end())
165     {
166         construct(finish, x);

```

```
167         ++finish;
168     }
169     else
170         insert_aux(position, x);
171     return begin() + n;
172 }
173
174 iterator insert(iterator position) { return insert(position, T()); }
175
176 void pop_back()
177 {
178     --finish;
179     destroy(finish);
180 }
181
182 iterator erase(iterator position)
183 {
184     if (position + 1 != end())
185         copy(position + 1, finish, position);
186     --finish;
187     destroy(finish);
188     return position;
189 }
190
191
192 iterator erase(iterator first, iterator last)
193 {
194     iterator i = copy(last, finish, first);
195     // 析构掉需要析构的元素
196     destroy(i, finish);
197     finish = finish - (last - first);
198     return first;
199 }
200
201 // 调整 size, 但是并不会重新分配内存空间
202 void resize(size_type new_size, const T& x)
203 {
204     if (new_size < size())
205         erase(begin() + new_size, end());
206     else
207         insert(end(), new_size - size(), x);
208 }
209 void resize(size_type new_size) { resize(new_size, T()); }
210
```

```

211     void clear() { erase(begin(), end()); }
212
213 protected:
214     // 分配空间, 并且复制对象到分配的空间处
215     iterator allocate_and_fill(size_type n, const T& x)
216     {
217         iterator result = data_allocator::allocate(n);
218         uninitialized_fill_n(result, n, x);
219         return result;
220     }
221
222     // 提供插入操作
223
224     //////////////////////////////////////
225     //             insert_aux(iterator position, const T& x)
226     //             |
227     //             |----- 容量是否足够?
228     //             ↓
229     //             -----
230     //             Yes |                               | No
231     //             ↓   |                               |
232     //             从 oposition 开始, 整体向后移动一个位置 |
233     //             construct(finish, *(finish - 1));         |
234     //             ++finish;                                |
235     //             T x_copy = x;                             |
236     //             copy_backward(position, finish - 2, finish - 1); |
237     //             *position = x_copy;                       |
238     //             ↓                                         |
239     //             data_allocator::allocate(len);
240     //             uninitialized_copy(start, position,
new_start);
241     //             construct(new_finish, x);
242     //             ++new_finish;
243     //             uninitialized_copy(position, finish,
new_finish);
244     //             destroy(begin(), end());
245     //             deallocate();
246
247     //////////////////////////////////////
248     template <class T, class Alloc>
249     void insert_aux(iterator position, const T& x)
250     {

```

```

251     if (finish != end_of_storage)    // 还有备用空间
252     {
253         // 在备用空间起始处构造一个元素，并以 vector 最后一个元素值为其初值
254         construct(finish, *(finish - 1));
255         ++finish;
256         T x_copy = x;
257         copy_backward(position, finish - 2, finish - 1);
258         *position = x_copy;
259     }
260     else    // 已无备用空间
261     {
262         const size_type old_size = size();
263         const size_type len = old_size != 0 ? 2 * old_size : 1;
264         // 以上配置元素：如果大小为0，则配置1（个元素大小）
265         // 如果大小不为0，则配置原来大小的两倍
266         // 前半段用来放置原数据，后半段准备用来放置新数据
267
268         iterator new_start = data_allocator::allocate(len); // 实际配置
269         iterator new_finish = new_start;
270         // 将内存重新配置
271         try
272         {
273             // 将原 vector 的安插点以前的内容拷贝到新 vector
274             new_finish = uninitialized_copy(start, position, new_start);
275             // 为新元素设定初值 x
276             construct(new_finish, x);
277             // 调整水位
278             ++new_finish;
279             // 将安插点以后的原内容也拷贝过来
280             new_finish = uninitialized_copy(position, finish, new_finish);
281         }
282         catch(...)
283         {
284             // 回滚操作
285             destroy(new_start, new_finish);
286             data_allocator::deallocate(new_start, len);
287             throw;
288         }
289         // 析构并释放原 vector
290         destroy(begin(), end());
291         deallocate();
292
293         // 调整迭代器，指向新 vector
294         start = new_start;

```

```

295         finish = new_finish;
296         end_of_storage = new_start + len;
297     }
298 }
299
300
301 // 在指定位置插入 n 个元素
302
303 // insert(iterator position, size_type n, const T& x)
304 //
305 // |----- 插入元素个数是否为0?
306 // ↓
307 // -----
308 // No | | Yes
309 // | | |
310 // | | ↓
311 // | | return;
312 // |----- 内存是否足够?
313 // |
314 // -----
315 // Yes | | No
316 // | | |
317 // |----- (finish - position) > n?
318 // | 分别调整指针
319 // ↓ |
320 // ----- |
321 // No | | Yes
322 // | | |
323 // ↓ ↓ |
324 // 插入操作, 调整指针 插入操作, 调整指针 |
325 // ↓
326 // data_allocator::allocate(len);
327 // new_finish = uninitialized_copy(start, position, new_start);
328 // new_finish = uninitialized_fill_n(new_finish, n, x);
329 // new_finish = uninitialized_copy(position, finish, new_finish);
330 // destroy(start, finish);
331 // deallocate();
332
333
334 template <class T, class Alloc>
335 void insert(iterator position, size_type n, const T& x)

```

```

336 {
337     // 如果 n 为0则不进行任何操作
338     if (n != 0)
339     {
340         if (size_type(end_of_storage - finish) >= n)
341         { // 剩下的备用空间大于等于“新增元素的个数”
342             T x_copy = x;
343             // 以下计算插入点之后的现有元素个数
344             const size_type elems_after = finish - position;
345             iterator old_finish = finish;
346             if (elems_after > n)
347             {
348                 // 插入点之后的现有元素个数 大于 新增元素个数
349                 uninitialized_copy(finish - n, finish, finish);
350                 finish += n; // 将 vector 尾端标记后移
351                 copy_backward(position, old_finish - n, old_finish);
352                 fill(position, position + n, x_copy); // 从插入点开始填入新值
353             }
354             else
355             {
356                 // 插入点之后的现有元素个数 小于等于 新增元素个数
357                 uninitialized_fill_n(finish, n - elems_after, x_copy);
358                 finish += n - elems_after;
359                 uninitialized_copy(position, old_finish, finish);
360                 finish += elems_after;
361                 fill(position, old_finish, x_copy);
362             }
363         }
364         else
365         { // 剩下的备用空间小于“新增元素个数”（那就必须配置额外的内存）
366             // 首先决定新长度：就长度的两倍，或旧长度+新增元素个数
367             const size_type old_size = size();
368             const size_type len = old_size + max(old_size, n);
369             // 以下配置新的 vector 空间
370             iterator new_start = data_allocator::allocate(len);
371             iterator new_finish = new_start;
372             __STL_TRY
373             {
374                 // 以下首先将旧的 vector 的插入点之前的元素复制到新空间
375                 new_finish = uninitialized_copy(start, position, new_start);
376                 // 以下再将新增元素（初值皆为 n）填入新空间
377                 new_finish = uninitialized_fill_n(new_finish, n, x);
378                 // 以下再将旧 vector 的插入点之后的元素复制到新空间
379                 new_finish = uninitialized_copy(position, finish,

```



```

new_finish);
380         }
381 #         ifdef  __STL_USE_EXCEPTIONS
382         catch(...)
383         {
384             destroy(new_start, new_finish);
385             data_allocator::deallocate(new_start, len);
386             throw;
387         }
388 #         endif /* __STL_USE_EXCEPTIONS */
389         destroy(start, finish);
390         deallocate();
391         start = new_start;
392         finish = new_finish;
393         end_of_storage = new_start + len;
394     }
395 }
396 }
397 };
398

```

#### 4.2.3 vector 的迭代器

`vector` 维护的是一个连续线性空间，所以不论其元素型别为何，原生指标都可以做为 `vector` 的迭代器而满足所有必要条件，因为 `vector` 迭代器所需要的操作行为如 `operator*`, `operator->`, `operator++`, `operator--`, `operator+`, `operator-`, `operator+=`, `operator-=`，原生指标天生就具备。`vector` 支援随机存取，而原生指标正有着这样的能力。所以，`vector` 提供的是 Random Access Iterators。

```

template <class T, class Alloc = alloc>
class vector {
public:
    typedef T value_type;
    typedef value_type* iterator;
    ...
};

```

根据上述定义，如果用户写出如下代码：

```

Vector<int>::iterator ivite;
Vector<shape>::iterator svite;

```

则 `ivite` 的类型就是 `int*`，`svite` 的类型就是 `shape*`。

#### 4.2.4 vector 的数据结构

`vector` 所采用的数据结构非常简单：线性连续空间。它以两个迭代器 `start` 和 `finish` 分别指向配置得来的连续空间中目前已被使用的范围，并

以迭代器 `end_of_storage` 指向整块连续空间（含备用空间）的尾端：

```
template <class T, class Alloc = alloc>
class vector {
...
protected:
    iterator start;
    iterator finish;
//表示目前使用空间的头
//表示目前使用空间的尾
    iterator end_of_storage; //表示目前可用空间的尾
...
};
```

#### 4.2.5 vector 的构建与内存管理

`vector` 预设使用 `alloc`（第二章）做为空间配置器，并据此另外定义了一个 `data_allocator`，为的是更方便以元素大小为配置单位：

```
template <class T, class Alloc = alloc>
class vector {
protected:
// simple_alloc<> 见 2.2.4 节
    typedef simple_alloc<value_type, Alloc> data_allocator;
...
};
```

于是，`data_allocator::allocate(n)` 表示配置 `n` 个元素空间。

`vector` 提供许多 constructors，其中一个允许我们指定空间大小及初值：

```
// 构造函数，允许指定 vector 大小 n 和初值 value
vector(size_type n, const T& value) {fill_initialize(n, value); }
```

// 充填并予初始化

```
void fill_initialize(size_type n, const T& value) {
    start = allocate_and_fill(n, value);
    finish = start + n;
    end_of_storage = finish;
}
```

// 配置而后充填

```
iterator allocate_and_fill(size_type n, const T& x) {
    iterator result = data_allocator::allocate(n); // 配置 n 个元素空间
    uninitialized_fill_n(result, n, x); // 全域函数，见 2.3 节
    return result;
}
```

uninitialized\_fill\_n()会根据第一参数的型别特性 (type traits, 3.7 节), 决定使用算法 fill\_n()或反复呼叫 construct() 来完成任务 (见 2.3 节描述)。

### Vector::push\_back()

当我们以 push\_back()将新元素安插于 vector 尾端, 该函式首先检查是否还有 备用空间? 如果有就直接在备用空间上建构元素, 并调整迭代器 finish, 使 vector 变大。如果没有备用空间了, 就扩充空间 (重新配置、搬移数据、释放原空间):

当我们以 push\_back()将新元素安插于 vector 尾端, 该函式首先检查是否还有 备用空间? 如果有就直接在备用空间上建构元素, 并调整迭代器 finish, 使 vector 变大。如果没有备用空间了, 就扩充空间 (重新配置、搬移数据、释放原空间):

```
void push_back(const T& x) {
    if (finish != end_of_storage) { //还有备用空间
        construct(finish, x);    //全域函式, 见 2.2.3 节。
        ++finish;
    }
    else
        insert_aux(end(), x); // vector member function, 见以下列表
}

// 提供插入操作
//
//          insert_aux(iterator position, const T& x)
//
//          |
//          | 容量是否足够?
//          ↓
//
// -----
//
//      Yes |                               | No
//          |                               |
//          ↓                               |
// 从 position 开始, 整体向后移动一个位置 |
// construct(finish, *(finish - 1));      |
// ++finish;                             |
// T x_copy = x;                          |
// copy_backward(position, finish - 2, finish - 1); |
// *position = x_copy;                    |
//                                         ↓
//                                         data_allocator::allocate(len);
//                                         uninitialized_copy(start, position, new_start);
//                                         construct(new_finish, x);
//                                         ++new_finish;
//                                         uninitialized_copy(position, finish, new_finish);
```

```

//                                     destroy(begin(), end());
//                                     deallocate();
////////////////////////////////////

template <class T, class Alloc>
void insert_aux(iterator position, const T& x)
{
    if (finish != end_of_storage)    // 还有备用空间
    {
        // 在备用空间起始处构造一个元素，并以 vector 最后一个元素值为其初值
        construct(finish, *(finish - 1));
        ++finish;
        T x_copy = x;
        copy_backward(position, finish - 2, finish - 1);
        *position = x_copy;
    }
    else    // 已无备用空间
    {
        const size_type old_size = size();
        const size_type len = old_size != 0 ? 2 * old_size : 1;
        // 以上配置元素：如果大小为 0，则配置 1（个元素大小）
        // 如果大小不为 0，则配置原来大小的两倍
        // 前半段用来放置原数据，后半段准备用来放置新数据

        iterator new_start = data_allocator::allocate(len);    // 实际配置
        iterator new_finish = new_start;
        // 将内存重新配置
        try
        {
            // 将原 vector 的安插点以前的内容拷贝到新 vector
            new_finish = uninitialized_copy(start, position, new_start);
            // 为新元素设定初值 x
            construct(new_finish, x);
            // 调整水位
            ++new_finish;
            // 将安插点以后的原内容也拷贝过来
            new_finish = uninitialized_copy(position, finish, new_finish);
        }
        catch(...)
        {
            // 回滚操作
            destroy(new_start, new_finish);
            data_allocator::deallocate(new_start, len);
            throw;
        }
    }
}

```

```

    }
    // 析构并释放原 vector
    destroy(begin(), end());
    deallocate();

    // 调整迭代器，指向新 vector
    start = new_start;
    finish = new_finish;
    end_of_storage = new_start + len;
}
}

```

### Vector::insert()

//从 position 开始，安插 n 个元素，元素初值为 x

```
template <class T, class Alloc>
```

```
void vector<T, Alloc>::insert(iterator position, size_type n, const T& x)
```

```

{
    if (n != 0) { // 当 n != 0 才进行以下所有动作
        if (size_type(end_of_storage - finish) >= n) // 备用空间大于等于「新增元素个数」
            T x_copy = x;
            // 以下计算安插点之后的现有元素个数
            const size_type elems_after = finish - position;
            iterator old_finish = finish;
            if (elems_after > n)
                // 「安插点之后的现有元素个数」大于「新增元素个数」，见图示 4-2-1
                uninitialized_copy(finish - n, finish, finish); // 将 finish 前 n 个元素先拷贝到备用空间
                finish += n; // 将 vector 尾端标记后移
                copy_backward(position, old_finish - n, old_finish);
                fill(position, position + n, x_copy); // 从安插点开始填入新值
            }
            else {
                // 「安插点之后的现有元素个数」小于等于「新增元素个数」，见图示 4-2-2
                uninitialized_fill_n(finish, n - elems_after, x_copy);
                finish += n - elems_after;
                uninitialized_copy(position, old_finish, finish);
                finish += elems_after;
                fill(position, old_finish, x_copy);
            }
        }
    }
    else {
        // 备用空间小于「新增元素个数」（那就必须配置额外的内存），见图示 4-2-3
        // 首先决定新长度：旧长度的两倍，或旧长度+新增元素个数。
        const size_type old_size = size();
        const size_type len = old_size + max(old_size, n);
    }
}

```

```

// 以下配置新的 vector 空间
iterator new_start = data_allocator::allocate(len);
iterator new_finish = new_start;
__STL_TRY {
    // 以下首先将旧 vector 的安插点之前的元素复制到新空间。
    new_finish = uninitialized_copy(start, position, new_start);
    // 以下再将新增元素（初值皆为 n）填入新空间。
    new_finish = uninitialized_fill_n(new_finish, n, x);
    // 以下再将旧 vector 的安插点之后的元素复制到新空间。
    new_finish = uninitialized_copy(position, finish, new_finish);
}
#ifdef __STL_USE_EXCEPTIONS
catch(...) {
    // 如有异常发生，实现 "commit or rollback" semantics.
    destroy(new_start, new_finish);
    data_allocator::deallocate(new_start, len);
    throw;
}
#endif /* __STL_USE_EXCEPTIONS */
// 以下清除并释放旧的 vector
destroy(start, finish);
deallocate();
// 以下调整水位标记
start = new_start;
finish = new_finish;
end_of_storage = new_start + len;
}
}
}

```

(1-1) 安插点之后的现有元素个数 3 > 新增元素个数 2

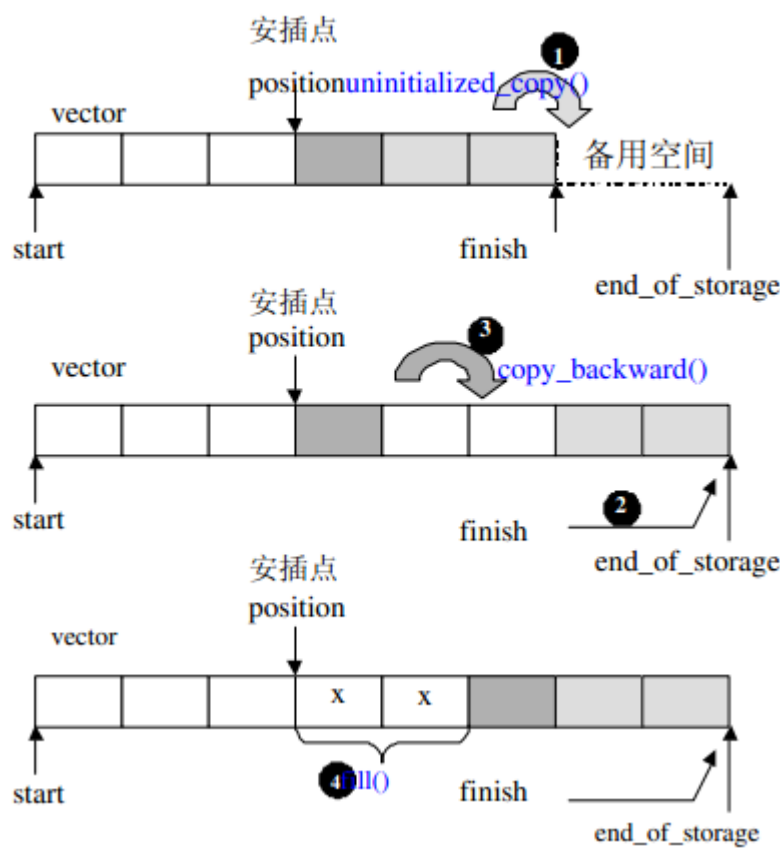


图4-2-1基本思路是要插入 n 个元素，则把数组中最后 n 个元素拷贝到备用空间中，然后再往后调节剩下的未移动的位置，最后插入 n 个元素

(1-2) 安插点之后的现有元素个数 2 新增元素个数 3

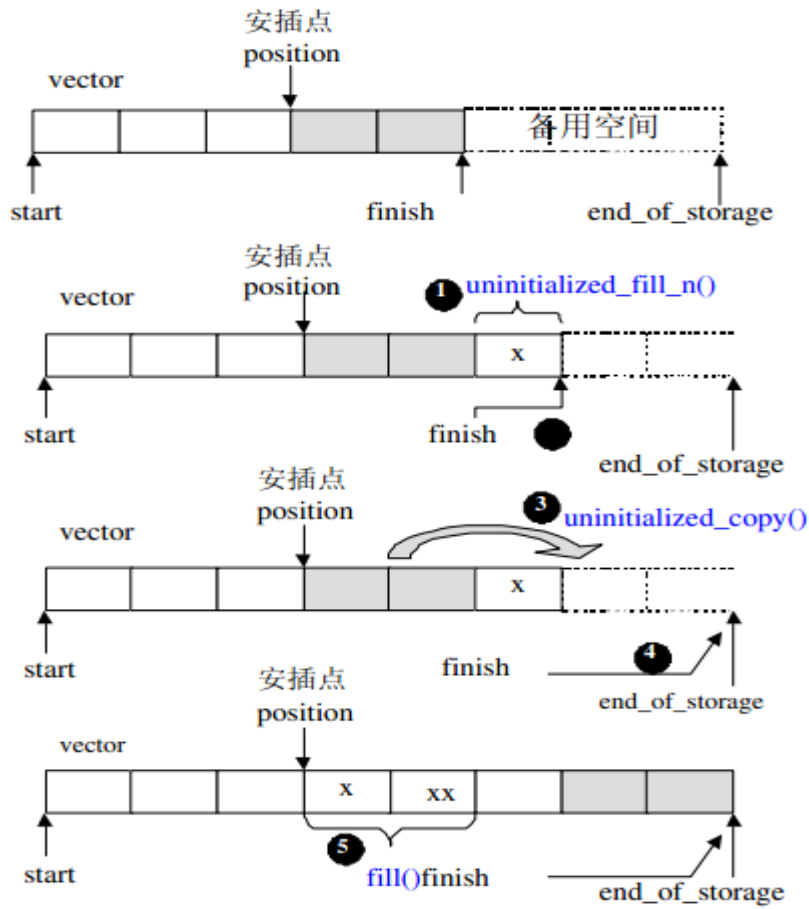


图 4-2-2 基本思路是先将多余的 x 插入到备用空间,使得插入点之后的现有元素个数 $\geq$ 将要插入的元素 (注意已经插入了一些,所以要重新计算),然后插入点之后需要被替换的元素复制到备用空间,最后完成其他 x 的插入。

(2) 备用空间 新增元素个数

例: 下图,  $n=3$

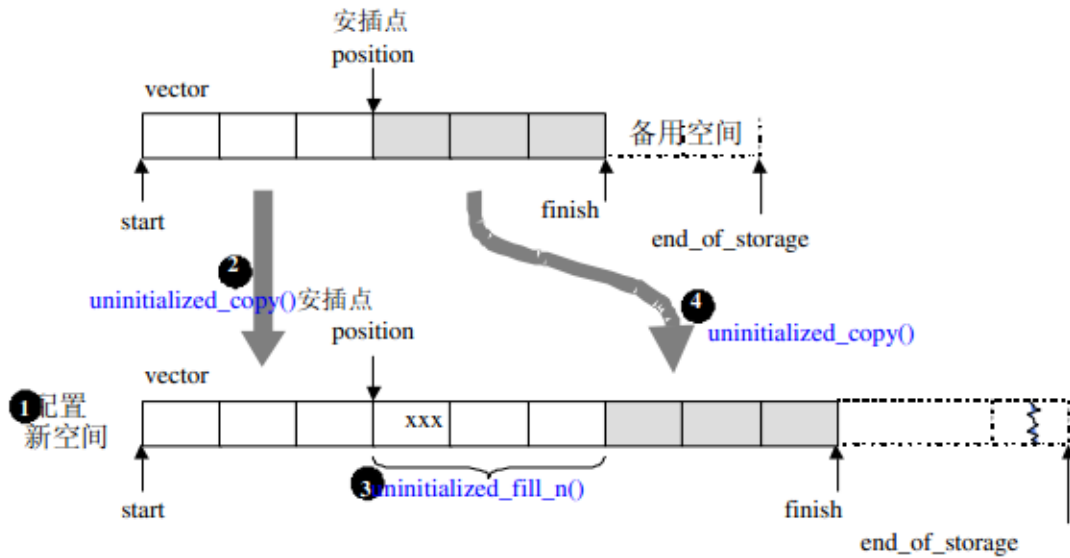




图 4-2-3

注意：vector 的插入操作可能导致内存空间重新分配，以至于原有的迭代器全部失效。

### 4.3 list

相较于 vector 的连续线性空间，list 就显得复杂许多，它的好处是每次插入或删除一个元素，就配置或释放一个元素空间。因此，list 对于空间的运用有绝对的精准，一点也不浪费。而且，对于任何位置的元素插入或元素移除，list 永远是常数时间。

List 和 vector 是最常使用的容器，到底该如何选择这两个容器，必须视元素的多寡、元素的构造复杂度、元素存取行为的特性而定。

list 不仅是一个双向链表，而且还是一个环状双向链表。另外，还有一个重要性质，插入操作和接合（splice）操作都不会造成原有的 list 迭代器失效，这在 vector 是不成立的。因为 vector 的插入操作可能造成记忆体重新配置，导致原有的迭代器全部失效。甚至 list 的元素删除操作（erase），也只有“指向被删除元素”的那个迭代器失效，其他迭代器不受任何影响。

以下是 list 的节点、迭代器数据结构设计以及 list 的源码剖析：

```

////////////////////////////////////
// list 结点, 提供双向访问能力
////////////////////////////////////
template <class T>
struct __list_node
{
    typedef void* void_pointer;
    void_pointer next;
    void_pointer prev;
    T data;
};

```

List 不再像 vector 一样，使用原生指针作为迭代器，因为其节点不保证在储存空间上连续。List 迭代器必须有指向 list 节点，并有能力做正确的递增、递减、取值、成员存取等操作。

List 是双向链表，所以迭代器是 bidirectional iterators。

// 至于为什么不使用默认参数，这个是因为有一些编译器不能提供推导能力，  
// 而作者又不想维护两份代码，故不使用默认参数

```

template<class T, class Ref, class Ptr>
struct __list_iterator
{
    typedef __list_iterator<T, T&, T*>          iterator;    // STL 标准强制要求
    typedef __list_iterator<T, Ref, Ptr>        self;

    typedef  bidirectional_iterator_tag  iterator_category;
    typedef  T value_type;
    typedef  Ptr pointer;
    typedef  Ref reference;

```

```

typedef __list_node<T>* link_type;
typedef size_t size_type;
typedef ptrdiff_t difference_type;

link_type node; //迭代器内部当然要有个普通指针，指向 list 的节点

__list_iterator(link_type x) : node(x) {}
__list_iterator() {}
__list_iterator(const iterator& x) : node(x.node) {}

// 在 STL 算法中需要迭代器提供支持
bool operator==(const self& x) const { return node == x.node; }
bool operator!=(const self& x) const { return node != x.node; }

// 以下对迭代器取值（dereference），取的是节点的数据值
reference operator*() const { return (*node).data; }

// 以下是迭代器的成员存取运算符的标准做法
pointer operator->() const { return &(operator*()); }

// 前缀自加，对迭代器累加 1，就是前进一个节点
self& operator++()
{
    node = (link_type)((*node).next);
    return *this;
}

// 后缀自加，需要先产生自身的一个副本，然后会再对自身操作，最后返回副本
self operator++(int)
{
    self tmp = *this;
    ++*this; //这里调用前缀自加？ This 是指向__list_iterator 的指针，而++操作对象是
            __list_iterator 自身
    return tmp;
}

// 前缀自减
self& operator--()
{
    node = (link_type)((*node).prev);
    return *this;
}

self operator--(int)

```

```

    {
        self tmp = *this;
        --*this;
        return tmp;
    }
};

////////////////////////////////////
// list 不仅是个双向链表, 而且还是一个环状双向链表
////////////////////////////////////

// 默认 allocator 为 alloc, 其具体使用版本请参照<stl_alloc.h>
template <class T, class Alloc = alloc>
class list
{
protected:
    typedef void* void_pointer;
    typedef __list_node<T> list_node;

    // 专属之空间配置器, 每次配置一个节点大小
    typedef simple_alloc<list_node, Alloc> list_node_allocator;

public:
    typedef T    value_type;
    typedef value_type*    pointer;
    typedef value_type&    reference;
    typedef list_node*    link_type;
    typedef size_t        size_type;
    typedef ptrdiff_t      difference_type;

    typedef __list_iterator<T, T&, T*>        iterator;

protected:
    link_type node ;        // 只要一个指针, 便可表示整个环状双向链表

    // 分配一个新结点, 注意这里并不进行构造,
    // 构造交给全局的 construct, 见<stl_stl_uninitialized.h>
    link_type get_node() { return list_node_allocator::allocate(); }

    // 释放指定结点, 不进行析构, 析构交给全局的 destroy
    void put_node(link_type p) { list_node_allocator::deallocate(p); }

    // 产生 (配置并构造) 一个节点, 首先分配内存, 然后进行构造
    // 注: commit or rollback

```

```

link_type create_node(const T& x)
{
    link_type p = get_node();
    construct(&p->data, x);
    return p;
}

```

// 析构结点元素，并释放内存

```

void destroy_node(link_type p)
{
    destroy(&p->data);
    put_node(p);
}

```

protected:

// 用于空链表的建立

```

void empty_initialize()
{
    node = get_node();    // 配置一个节点空间，令 node 指向它
    node->next = node;    // 令 node 头尾都指向自己，不设元素值
    node->prev = node;
}

```

// 创建值为 value 共 n 个结点的链表

// 注: commit or rollback

```

void fill_initialize(size_type n, const T& value)
{
    empty_initialize();
    __STL_TRY
    {
        // 此处插入操作时间复杂度 O(1)
        insert(begin(), n, value);
    }
    __STL_UNWIND(clear(); put_node(node));
}

```

public:

list() { empty\_initialize(); } //产生一个空链表

iterator begin() { return (link\_type)((\*node).next); }

// 链表成环，当指所以头节点也就是 end

iterator end() { return node; }

```

// 头结点指向自身说明链表中无元素
bool empty() const { return node->next == node; }

// 使用全局函数 distance()进行计算, 时间复杂度 O(n)
size_type size() const
{
    size_type result = 0;
    distance(begin(), end(), result);
    return result;
}

size_type max_size() const { return size_type(-1); }
reference front() { return *begin(); }
reference back() { return *(--end()); }

/////////////////////////////////////////////////////////////////
// 在指定位置插入元素
/////////////////////////////////////////////////////////////////
//      insert(iterator position, const T& x)
//              ↓
//      create_node(x)
//      p = get_node();----->list_node_allocator::allocate();
//      construct(&p->data, x);
//              ↓
//      tmp->next = position.node;
//      tmp->prev = position.node->prev;
//      (link_type(position.node->prev))->next = tmp;
//      position.node->prev = tmp;
/////////////////////////////////////////////////////////////////

iterator insert(iterator position, const T& x)
{
    link_type tmp = create_node(x);    // 产生一个节点
    // 调整双向指针, 使 tmp 插入进去
    tmp->next = position.node;
    tmp->prev = position.node->prev;
    (link_type(position.node->prev))->next = tmp;
    position.node->prev = tmp;
    return tmp;
}

// 指定位置插入 n 个值为 x 的元素, 详细解析见实现部分
void insert(iterator pos, size_type n, const T& x);

```

```

void insert(iterator pos, int n, const T& x)
{
    insert(pos, (size_type)n, x);
}
void insert(iterator pos, long n, const T& x)
{
    insert(pos, (size_type)n, x);
}

// 在链表前端插入结点
void push_front(const T& x) { insert(begin(), x); }
// 在链表最后插入结点
void push_back(const T& x) { insert(end(), x); }

// 移除迭代器 position 所指节点
iterator erase(iterator position)
{
    link_type next_node = link_type(position.node->next);
    link_type prev_node = link_type(position.node->prev);
    prev_node->next = next_node;
    next_node->prev = prev_node;
    destroy_node(position.node);
    return iterator(next_node);
}

// 擦除一个区间的结点，详细解析见实现部分
iterator erase(iterator first, iterator last);

void resize(size_type new_size, const T& x);
void resize(size_type new_size) { resize(new_size, T()); }
void clear();

// 删除链表第一个结点
void pop_front() { erase(begin()); }
// 删除链表最后一个结点
void pop_back()
{
    iterator tmp = end();
    erase(--tmp);
}

list(size_type n, const T& value) { fill_initialize(n, value); }
list(int n, const T& value) { fill_initialize(n, value); }
list(long n, const T& value) { fill_initialize(n, value); }

```

```

~list()
{
    // 释放所有结点 // 使用全局函数 distance()进行计算, 时间复杂度 O(n)
size_type size() const
{
    size_type result = 0;
    distance(begin(), end(), result);
    return result;
}
clear();
// 释放头结点
put_node(node);
}

```

```
list<T, Alloc>& operator=(const list<T, Alloc>& x);
```

protected:

```

////////////////////////////////////
// 将[first, last)内的所有元素移动到 position 之前
// 如果 last == position, 则相当于链表不变化, 不进行操作
// 理解这里代码, 可以想象两个链表 list1, list2, 需要从 list1 中删除[first, last)
// 同时需要向 List2 中加入[first, last),注意 last 处是开区间
////////////////////////////////////
void transfer(iterator position, iterator first, iterator last)
{
    if (position != last) // 如果 last == position, 则相当于链表不变化, 不进行操作
    {
        //(*last.node).prev 加入 list2 position 之前
        (*(link_type((*last.node).prev))).next = position.node;
        //修改 list1
        (*(link_type((*first.node).prev))).next = last.node;
        //first.node 加入 list1
        (*(link_type((*position.node).prev))).next = first.node;
        link_type tmp = link_type((*position.node).prev);
        //修改 list2
        (*position.node).prev = (*last.node).prev;
        //修改 list1
        (*last.node).prev = (*first.node).prev;
        //修改 list2
        (*first.node).prev = tmp;
    }
}

```

```

public:
    // 将链表 x 移动到 position 所指位置之前
    void splice(iterator position, list& x)
    {
        if (!x.empty())
            transfer(position, x.begin(), x.end());
    }

    // 将链表中 i 指向的内容移动到 position 之前
    void splice(iterator position, list&, iterator i)
    {
        iterator j = i;
        ++j;
        if (position == i || position == j) return;
        transfer(position, i, j);
    }

    // 将[first, last}元素移动到 position 之前
    void splice(iterator position, list&, iterator first, iterator last)
    {
        if (first != last)
            transfer(position, first, last);
    }

    void remove(const T& value);
    void unique();
    void merge(list& x);
    void reverse();
    void sort();

};

// 移除迭代器 position 所指节点
iterator erase(iterator position) {
    link_type next_node = link_type(position.node->next);
    link_type prev_node = link_type(position.node->prev);
    prev_node->next = next_node;
    next_node->prev = prev_node;
    destroy_node(position.node);
    return iterator(next_node);
}

// 销毁所有结点，将链表置空

```



```

template <class T, class Alloc>
void list<T, Alloc>::clear()
{
    link_type cur = (link_type) node->next;
    while (cur != node)
    {
        link_type tmp = cur;
        cur = (link_type) cur->next;
        destroy_node(tmp);
    }
    // 恢复 node 原始状态
    node->next = node;
    node->prev = node;
}

```

//将数值为 value 之所有元素移除

```

template <class T, class Alloc>
void list<T, Alloc>::remove(const T& value) {
    iterator first = begin();
    iterator last = end();
    while (first != last) { //巡访每一个节点
        iterator next = first;
        ++next;
        if (*first == value)
            erase(first); //找到就移除
        first = next;
    }
}

```

// 链表赋值操作

// 如果当前容器元素少于 x 容器，则析构多余元素，

// 否则将调用 insert 插入 x 中剩余的元素

```

template <class T, class Alloc>
list<T, Alloc>& list<T, Alloc>::operator=(const list<T, Alloc>& x)
{
    if (this != &x)
    {
        iterator first1 = begin();
        iterator last1 = end();
        const_iterator first2 = x.begin();
        const_iterator last2 = x.end();
        while (first1 != last1 && first2 != last2) *first1++ = *first2++;
        if (first2 == last2)
            erase(first1, last1);
    }
}

```

```

        else
            insert(last1, first2, last2);
    }
    return *this;
}

```

// 移除容器内所有的相邻的重复结点，注意只有连续且相同的元素，才会被移除剩一个

// 时间复杂度  $O(n)$

// 用户自定义数据类型需要提供 operator ==()重载

```
template <class T, class Alloc>
```

```
void list<T, Alloc>::unique()
```

```

{
    iterator first = begin();
    iterator last = end();
    if (first == last) return;
    iterator next = first;
    while (++next != last)
    {
        if (*first == *next)
            erase(next);
        else
            first = next;
        next = first;
    }
}

```

// 假设当前容器和 x 都已序，保证两容器合并后仍然有序，将 x 合并到\*this 身上

```
template <class T, class Alloc>
```

```
void list<T, Alloc>::merge(list<T, Alloc>& x) merge 值得一看
```

```

{
    iterator first1 = begin();
    iterator last1 = end();
    iterator first2 = x.begin();
    iterator last2 = x.end();

    // 注意：前提是，两个 lists 都已经递增排序
    while (first1 != last1 && first2 != last2)
        if (*first2 < *first1)
        {
            iterator next = first2;
            transfer(first1, first2, ++next); //将 first2 链入 first1 之前
            first2 = next;
        }
}

```

```

        else
            ++first1;
//当从 while 循环退出时, first1 == last1 || first2 == last2
if (first2 != last2)
    transfer(last1, first2, last2);
}

// reverse()将 *this 的内容逆向重置
template <class T, class Alloc>
void list<T, Alloc>::reverse() {
    // 以下判断, 如果是空白链表, 或仅有一个元素, 就不做任何动作。
    // 使用 size() == 0 || size() == 1 来判断, 虽然也可以, 但是比较慢。
    if (node->next == node || link_type(node->next)->next == node)
        return;
    iterator first = begin();
    ++first;
    while (first != end()) {
        iterator old = first;
        ++first;
        transfer(begin(), old, first);
    }
}

// list 不能使用 STL 算法 sort(), 必须使用自己的 sort() member function,
//因为 STL 算法 sort() 只接受 RandomAccessIterator.
//本函数采用 quick sort.
template <class T, class Alloc>
void list<T, Alloc>::sort() {
    // 以下判断, 如果是空白串行, 或仅有一个元素, 就不做任何动作。
    // 使用 size() == 0 || size() == 1 来判断, 虽然也可以, 但是比较慢。
    if (node->next == node || link_type(node->next)->next == node)
        return;

    // 一些新的 lists, 做为中介数据存放区
    list<T, Alloc> carry;
    list<T, Alloc> counter[64];
    int fill = 0;
    while (!empty()) {
        carry.splice(carry.begin(), *this, begin());
        int i = 0;
        while(i < fill && !counter[i].empty()) {
            counter[i].merge(carry);
            carry.swap(counter[i++]);
        }
    }
}

```

```

        carry.swap(counter[i]);
        if (i == fill) ++fill;
    }

    for (int i = 1; i < fill; ++i)
        counter[i].merge(counter[i-1]);
    swap(counter[fill-1]);
}

```

#### 4.4 deque

Refer to: <http://blog.csdn.net/hackbuteer1/article/details/7729451>

##### 4.4.1 deque 概述

Vector 是单项开口的连续空间，deque 则是一种双向开口的连续线性空间。所谓双向开口，意思是在头为两段分别做元素的插入和删除。

Deque 和 vector 的差异：deque 允许常熟时间内对起端进行元素插入或删除；deque 没有所谓容量的概念，因为它是动态地以分段连续空间组合而成，随时可以增减一段新的空间并连接起来。所以 deque 没有必要提供所谓的空间保留功能。

虽然 deque 也提供 random access iterator，但是他的迭代器不是原生指针，其复杂度比 vector 高很多，因此除非必要，我们应首选使用 vector，对 deque 进行排序，为提高效率，可将 deque 先完整复制到一个 vector 上，对 vector 进行排序，再复制回 deque。

##### 4.4.2 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空间，便分配一段定量连续空间，连接在 deque 的头端或尾端。Deque 的任务便是在这些分段连续空间上，维护其整体连续的假象，并提供随机存取界面。它虽然避开了 vector 中“重新分配、复制、释放”的轮回，但是却引入了复杂的迭代器架构。

deque 采用一块所谓的 map（注意，不是 STL 的 map 容器）做为主控。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。

```

template <class T, class Alloc = alloc, size_t BufSiz = 0>
class deque {
public:
    // Basic types
    typedef T value_type;
    typedef value_type* pointer;
    ...
protected:
    // Internal typedefs
    // 元素的指针的指针（pointer of pointer of T）
    typedef pointer* map_pointer;

protected:
    // Data members
    map_pointer map; // 指向 map，map 是块连续空间，其内的每个元素都是一个指针（称为节点），指向一块缓冲区。
    size_type map_size; // map 内可容纳多少指针。

```

```
...
};
```

实际上从上述代码可知，map 其实是一个 T\*\*。

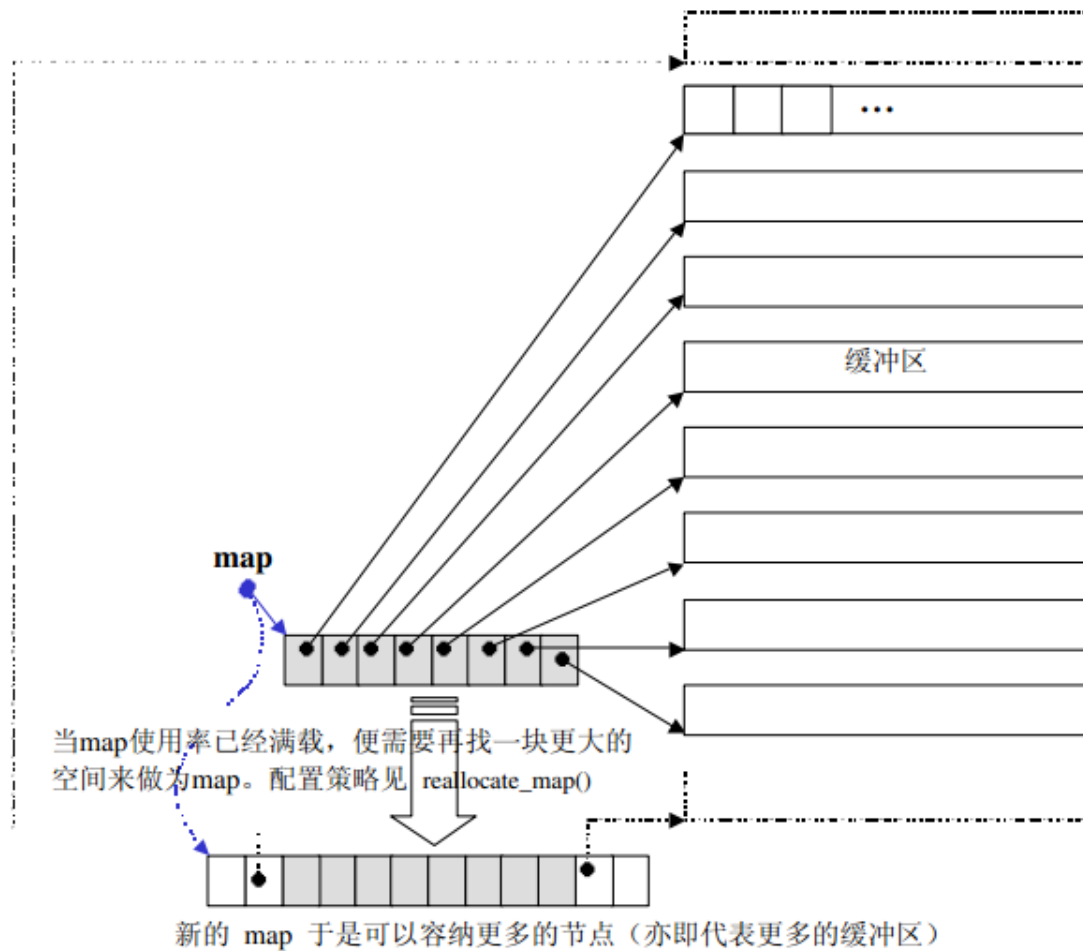


图 4-10 deque 的结构设计中，map 和 node-buffer 的关系

#### 4.4.3 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，落在迭代器的 `operator++` 和 `operator--` 两个运算子身上。

```
template <class T, class Ref, class Ptr, size_t BufSiz>
struct __deque_iterator { // 未继承 std::iterator
    typedef __deque_iterator<T, T&, T*, BufSiz> iterator;
    typedef __deque_iterator<T, const T&, const T*, BufSiz> const_iterator;
    static size_t buffer_size() { return __deque_buf_size(BufSiz, sizeof(T)); }
```

// 未继承 `std::iterator`，所以必须自行撰写五个必要的迭代器相应型别（第 3 章）

```
    Typedef random_access_iterator_tag iterator_category; // (1)
    typedef T value_type; // (2)
```

```

typedef Ptr pointer; // (3)
typedef Ref reference; // (4)
typedef size_t size_type;
typedef ptrdiff_t difference_type; // (5)
typedef T** map_pointer;

typedef __deque_iterator self;

// 保持与容器的联结
T* cur; // 此迭代器所指之缓冲区中的现行 (current) 元素
T* first; // 此迭代器所指之缓冲区的头
T* last; // 此迭代器所指之缓冲区的尾 (含备用空间, 就是尚未完全占用的空间)
map_pointer node; // 指向管控中心中的节点 node
...
};

```

其中用来决定缓冲区大小的函数 `buffer_size()`, 调用 `__deque_buf_size()`, 后者是个全域函数, 定义如下:

```

// 如果 n 不为 0, 传回 n, 表示 buffer size 由使用者自定。
// 如果 n 为 0, 表示 buffer size 使用默认值, 那么
// 如果 sz (元素大小, sizeof(value_type)) 小于 512, 传回 512/sz,
// 如果 sz 不小于 512, 传回 1。
inline size_t __deque_buf_size(size_t n, size_t sz)
{
    return n != 0 ? n : (sz < 512 ? size_t(512 / sz) : size_t(1));
}

```

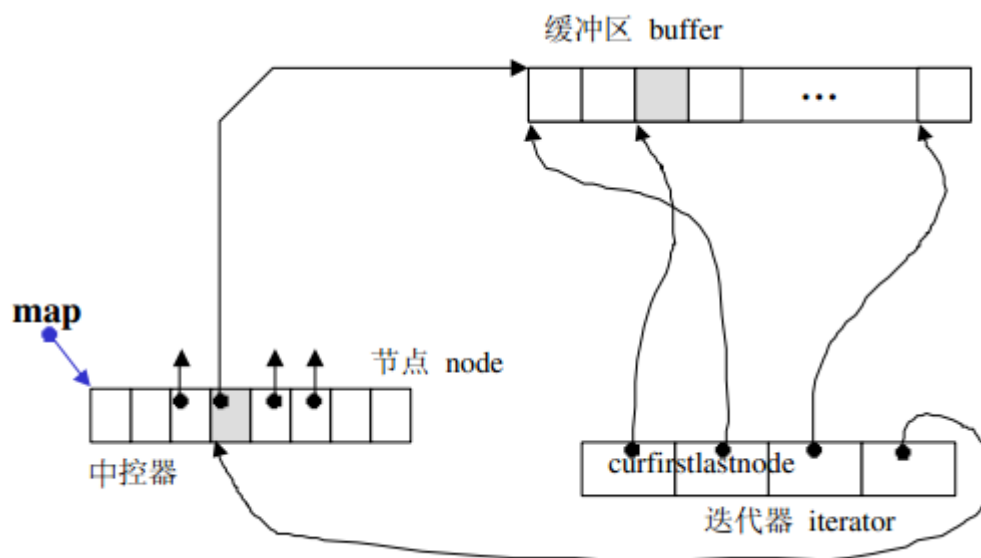


图 4-11 deque 的中控器、缓冲区、迭代器的相互关系

下面是 deque 迭代器的几个关键行为。由于迭代器内对各种指标运算都做了多载化动

作，所以各种指标运算如加、减、前进、后退…都不能直观视之。其中最重点的关键就是：一旦行进时遇到缓冲区边缘，要特别当心，视前进或后退而定，可能需要调用 `set_node()` 跳一个缓冲区：

```
void set_node(map_pointer new_node) {
    node = new_node;
    first = *new_node;
    last = first + difference_type(buffer_size()); //buffer_size 与 node 无关，见前面 buffer_size()
                                                    函数定义
}
```

```
__deque_iterator(T* x, map_pointer y)
    : cur(x), first(*y), last(*y + buffer_size()), node(y) {}
__deque_iterator() : cur(0), first(0), last(0), node(0) {}
__deque_iterator(const iterator& x)
    : cur(x.cur), first(x.first), last(x.last), node(x.node) {}
```

```
reference operator*() const { return *cur; }
```

// 判断两个迭代器间的距离

```
difference_type operator-(const self& x) const //a - b
```

```
{
    //因为 node 节点在 map 中是相邻的，
    return difference_type(buffer_size()) * (node - x.node - 1) +
        (cur - first) + (x.last - x.cur);
}
```

// 参考 More Effective C++, item6: Distinguish between prefix and

// postfix forms of increment and decrement operators.

```
self& operator++() //++操作只对 iterator 内部的 curr 进行修改
```

```
{
    ++cur;    // 切换至下一个元素
    if (cur == last)    // 如果已达到缓冲区的尾端
    {
        set_node(node + 1);    // 就切换至下一节点（亦即缓冲区）
        cur = first;           // 到达新的 node 的第一个元素，这里的机制???
    }
    return *this; //这里 this 的机制???
}
```

// 后缀自增

// 返回当前迭代器的一个副本，并调用前缀自增运算符实现迭代器自身的自增

```
self operator++(int)
```

```
{
    self tmp = *this;
```

```

    ++*this; //调用前缀++操作符
    return tmp;
}

// 前缀自减, 处理方式类似于前缀自增
// 如果当前迭代器指向元素是当前缓冲区的第一个元素
// 则将迭代器状态调整为前一个缓冲区的最后一个元素
self& operator--()
{
    if (cur == first)    // 如果已达到缓冲区的头端
    {
        set_node(node - 1);    // 就切换至前一节点 (亦即缓冲区)
        cur = last;            // 的最后一个元素
    }
    --cur;
    return *this;
}

self operator--(int)
{
    self tmp = *this;
    --*this;
    return tmp;
}

// 以下实现随机存取。迭代器可以直接跳跃 n 个距离
self& operator+=(difference_type n)
{
    difference_type offset = n + (cur - first);
    if (offset >= 0 && offset < difference_type(buffer_size()))
        cur += n;            // 目标位置在同一缓冲区内
    else
    {
        // 目标位置不在同一缓冲区内
        //in node unit
        difference_type node_offset =
            offset > 0 ? offset / difference_type(buffer_size())
            : -difference_type((-offset - 1) / buffer_size()) - 1;
        // 切换至正确的节点 (亦即缓冲区)
        set_node(node + node_offset);
        // 切换至正确的元素
        cur = first + (offset - node_offset * difference_type(buffer_size()));
    }
    return *this;
}

```



```

self operator+(difference_type n) const
{
    self tmp = *this;

    // 这里调用了 operator +=()可以自动调整指针状态
    return tmp += n;
}

// 将 n 变为-n 就可以使用 operator +=()了,
self& operator-=(difference_type n) { return *this += -n; }

self operator-(difference_type n) const
{
    self tmp = *this;
    return tmp -= n;
}

reference operator[](difference_type n) const { return *(*this + n); }

bool operator==(const self& x) const { return cur == x.cur; }
bool operator!=(const self& x) const { return !(*this == x); }
bool operator<(const self& x) const
{
    return (node == x.node) ? (cur < x.cur) : (node < x.node);
}

```

#### 4.4.4 deque 的数据结构

deque 除了维护一个先前说过的指向 map 的指标外，也维护 start, finish 两个迭代器，分别指向第一缓冲区的第一个元素和最后缓冲区的最后一个元素（的下一位置）。此外它当然也必须记住目前的 map 大小。因为一旦 map 所提供的节点不足，就必须重新配置更大的一块 map。

//见 \_\_deque\_buf\_size()。BufSize 默认值为 0 的唯一理由是为了闪避某些  
//编译器在处理常数算式（constant expressions）时的臭虫。  
//预设使用 alloc 为配置器。

```
template <class T, class Alloc = alloc, size_t BufSiz = 0>
```

```
class deque {
public:
    // Basic types
    typedef T value_type;
    typedef value_type* pointer;
    typedef size_t size_type;
```

```
public:
    // Iterators
```

```

    Typedef    __deque_iterator<T, T&, T*,BufSiz>  iterator;

protected:                                     // Internal typedefs
// 元素的指针的指针 (pointer of pointer of T)
    typedef    pointer*    map_pointer;

protected:                                     // Data members
    iterator    start;
    iterator    finish;

    map_pointer    map; //指向 map, map 是块连续空间,
                        // 其每个元素都是个指针, 指向一个节点 (缓冲区)。
    size_type    map_size; // map 内有多少指标。
...
};

```

有了上述结构，以下数个机能便可轻易完成：

```

public:                                     // Basic accessors
    iterator    begin() { return start; }
    iterator    end() { return finish; }

    Reference    operator[](size_type n) {
        return start[difference_type(n)]; //唤起 __deque_iterator<>::operator[]
    }

    Reference    front() { return *start; } // 唤起 __deque_iterator<>::operator*
    Reference    back() {
        iterator tmp = finish;
        --tmp; //唤起 __deque_iterator<>::operator--
        return *tmp; //唤起 __deque_iterator<>::operator*
        // 以上三行何不改为: return *(finish-1);
        // 因为 __deque_iterator<> 没有为 (finish-1) 定义运算符?!
    }

    // 下行最后有两个 ‘;’, 虽奇怪但合乎语法。
    size_type    size() const { return finish - start;; }
// 以上唤起 iterator::operator-
    size_type    max_size() const { return size_type(-1); }
    bool         empty() const { return finish == start; }

```

#### 4.4.5 deque 的构造与内存管理：ctor, push\_back, push\_front

deque 自行定义了两个专属的空间配置器：

```
protected:                                // Internal typedefs
// 专属之空间配置器，每次配置一个元素大小
    Typedef    simple_alloc<value_type, Alloc> data_allocator;
// 专属之空间配置器，每次配置一个指针大小
    Typedef    simple_alloc<pointer, Alloc> map_allocator;
```

并提供有一个 constructor 如下：

```
deque(int n, const value_type& value)
    : start(), finish(), map(0), map_size(0)
{
    fill_initialize(n, value); //使用 value 初始化 n 个元素
}
```

调用 fill\_initialize()负责产生并安排好 deque 的结构，并将元素的初值设定妥当：

```
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::fill_initialize(size_type n, const value_type& value) {
    //计算每一个 map node 对应的缓冲区中能存放多少元素
    Int mapNodeCapacity= BufSize / (size_t)(sizeof(value_type));
    //计算需要多少个 map node
    Int mapNodeNum= n/mapNodeCapacity + 1 ;

    // typedef simple_alloc<pointer,alloc> Map_allocator;
    Map_allocator.allocate(mapNodeNum);

    create_map_and_nodes(n); // 把 deque 的结构都产生并安排好
    map_pointer cur;
    __STL_TRY {
        // 为每个节点的缓冲区设定初值
        for (cur = start.node; cur < finish.node; ++cur)
            uninitialized_fill(*cur, *cur + buffer_size(), value);
        // 最后一个节点的设定稍有不同（因为尾端可能有备用空间，不必设初值）
        uninitialized_fill(finish.first, finish.cur, value);
    }
    catch(...) {
        ...
    }
}
```

其中 create\_map\_and\_nodes()负责产生并安排好 deque 的结构：

```
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::create_map_and_nodes(size_type num_elements)
{
    // 需要节点数=(元素个数/每个缓冲区可容纳的元素个数)+1
    // 如果刚好整除，会多配一个节点。
```

```

size_type num_nodes = num_elements / buffer_size() + 1;

// 一个 map 要管理几个节点。最少 8 个，最多是 “所需节点数加 2”
// （前后各预留一个，扩充时可用）。
map_size = max(initial_map_size(), num_nodes + 2);
map = map_allocator::allocate(map_size);
// 以上配置出一个 “具有 map_size 个节点” 的 map。

// 以下令 nstart 和 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。
map_pointer nstart = map + (map_size - num_nodes) / 2;
map_pointer nfinish = nstart + num_nodes - 1;

map_pointer cur;
__STL_TRY {
    // 为 map 内的每个现用节点配置缓冲区。所有缓冲区加起来就是 deque 的
    // 可用空间（最后一个缓冲区可能留有一些余裕）。
    for (cur = nstart; cur <= nfinish; ++cur)
        *cur = allocate_node(); // 分配节点，大小为缓冲区大小
}
catch(...) {
    // "commit or rollback" 语意：若非全部成功，就一个不留
    ....
}
// 为 deque 的两个迭代器 start 和 end 设定正确内容
Start.set_node(nstart);
Finish.set_node(nfinish);
Start.cur = start.first;
Finish.cur = finish.first + num_elements & buffer_size();
}

```

### **Deque::push\_back()**

```

void push_back(const value_type& t)
{
    // 最后缓冲区尚有两个（含）以上的元素备用空间
    if (finish.cur != finish.last - 1)
    {
        construct(finish.cur, t);    // 直接在备用空间上构造元素
        ++finish.cur;    // 调整最后缓冲区的使用状态
    }
    // 容量已满就要新申请内存了
    else
        push_back_aux(t);
}

```

```

// 只有当 finish.cur == finish.last - 1 时才会被调用
// 也就是说，只有当最后一个缓冲区只剩下一个备用元素空间时才会被调用
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::push_back_aux(const value_type& t)
{
    value_type t_copy = t;
    reserve_map_at_back(); //若符合某种条件，则必须重换一个 map
    *(finish.node + 1) = allocate_node();    // 配置一个新节点（缓冲区）
    __STL_TRY
    {
        construct(finish.cur, t_copy);        //
        finish.set_node(finish.node + 1);    // 改变 finish，令其指向新节点
        finish.cur = finish.first;           // 设定 finish 的状态
    }
    __STL_UNWIND(deallocate_node(*(finish.node + 1)));
}

```

#### **Deque::push\_front()**

```

void push_front(const value_type& t)
{
    if (start.cur != start.first)    // 第一缓冲区尚有备用空间
    {
        construct(start.cur - 1, t); // 直接在备用空间上构造元素
        --start.cur;    // 调整第一缓冲区的使用状态
    }
    else    // 第一缓冲区已无备用空间
        push_front_aux(t);
}

```

// Called only if start.cur == start.first.

```

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::push_front_aux(const value_type& t)
{
    value_type t_copy = t;
    reserve_map_at_front(); //若符合某种条件，则必须重换 map
    *(start.node - 1) = allocate_node();
    __STL_TRY
    {
        start.set_node(start.node - 1);    // 改变 start，令其指向新节点
        start.cur = start.last - 1;        // 设定 start 的状态
        construct(start.cur, t_copy);      // 针对标的元素设值
    }
    catch(...)

```

```

    { //commit or rollback 语义
        start.set_node(start.node + 1);
        start.cur = start.first;
        deallocate_node(*(start.node - 1));
        throw;
    }
}

```

### **Deque::reallocate\_map**

```

void reserve_map_at_back (size_type nodes_to_add = 1) {
    if (nodes_to_add + 1 > map_size - (finish.node - map))
        // 如果 map 尾端的节点备用空间不足
        // 符合以上条件则必须重换一个 map (配置更大的, 拷贝原来的, 释放原来的)
        reallocate_map(nodes_to_add, false);
}

```

```

void reserve_map_at_front (size_type nodes_to_add = 1) {
    if (nodes_to_add > start.node - map)
        // 如果 map 前端的节点备用空间不足
        // 符合以上条件则必须重换一个 map (配置更大的, 拷贝原来的, 释放原来的)
        reallocate_map(nodes_to_add, true);
}

```

```

template <class T, class Alloc, size_t BufSize>

```

```

void deque<T, Alloc, BufSize>::reallocate_map(size_type nodes_to_add, bool add_at_front) {
    size_type old_num_nodes = finish.node - start.node + 1;
    size_type new_num_nodes = old_num_nodes + nodes_to_add;

```

```

    map_pointer    new_nstart;
    if (map_size > 2 * new_num_nodes) {
        new_nstart = map + (map_size - new_num_nodes) / 2
                        + (add_at_front ? nodes_to_add: 0);
        if (new_nstart < start.node)
            copy(start.node, finish.node + 1, new_nstart);
        else
            copy_backward(start.node, finish.node + 1, new_nstart + old_num_nodes);
    }

```

```

    else {
        size_type new_map_size = map_size + max(map_size, nodes_to_add) + 2;
        // 配置一块空间, 准备给新 map 使用。
        map_pointer new_map = map_allocator::allocate(new_map_size);
        new_nstart = new_map + (new_map_size - new_num_nodes) / 2
                        + (add_at_front ? nodes_to_add : 0);
        // 把原 map 内容拷贝过来。

```

```

        copy(start.node, finish.node + 1, new_nstart);
        // 释放原 map
        map_allocator::deallocate(map, map_size);
        // 设定新 map 的起始地址与大小
        map = new_map;
        map_size = new_map_size;
    }

    // 重新设定迭代器 start 和 finish
    start.set_node(new_nstart);
    finish.set_node(new_nstart + old_num_nodes - 1);
}

```

#### 4.4.6 deque 的元素操作:pop\_back, pop\_front, clear, erase, insert

所谓 pop，是将元素拿掉。无论从 deque 的最前端或最尾端取元素，都需考虑在某种条件下，将缓冲区释放掉。

##### **Deque::pop\_back**

```

void pop_back()
{
    if (finish.cur != finish.first)    // 最后缓冲区有一个（或更多）元素
    {
        --finish.cur;    // 调整指针，相当于排除了最后元素
        destroy(finish.cur);    // 将最后元素析构
    }
    else
        // 最后缓冲区没有任何元素
        pop_back_aux();    // 这里将进行缓冲区的释放工作
}

// 只有当 finish.cur == finish.first 时才会被调用
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::pop_back_aux()
{
    deallocate_node(finish.first);    // 释放最后一个缓冲区
    finish.set_node(finish.node - 1);    // 调整 finish 状态，使指向
    finish.cur = finish.last - 1;    // 上一个缓冲区的最后一个元素
    destroy(finish.cur);    // 将该元素析构
}

```

##### **Deque::pop\_front**

```

void pop_front()

```

```

{
    if (start.cur != start.last - 1)    // 第一缓冲区有两个（或更多）元素
    {
        destroy(start.cur);    // 将第一元素析构
        ++start.cur;           // 调整指针，相当于排除了第一元素
    }
    else
        // 第一缓冲区仅有一个元素
        pop_front_aux();    // 这里将进行缓冲区的释放工作
}

// 只有当 start.cur == start.last - 1 时才会被调用
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::pop_front_aux()
{
    destroy(start.cur);    // 将第一个缓冲区的第一个（也是最后一个、唯一一个）元素析构
    deallocate_node(start.first);    // 释放第一缓冲区
    start.set_node(start.node + 1);    // 调整 start 状态，使指向
    start.cur = start.first;    // 下一个缓冲区的第一个元素
}

```

### Deque::clear()

clear(), 用来清除整个 deque。请注意, deque 的最初状态（无任何元素时）保有一个缓冲区, 因此 clear()完成之后回复初始状态, 也一样要保留一个缓冲区。

```

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::clear()
{
    // 以下针对头尾以外的每一个缓冲区
    for (map_pointer node = start.node + 1; node < finish.node; ++node)
    {
        // 将缓冲区内的所有元素析构
        destroy(*node, *node + buffer_size());
        // 释放缓冲区内存
        data_allocator::deallocate(*node, buffer_size());
    }

    if (start.node != finish.node)    // 至少有头尾两个缓冲区
    {
        destroy(start.cur, start.last);    // 将头缓冲区的目前所有元素析构
        destroy(finish.first, finish.cur);    // 将尾缓冲区的目前所有元素析构
        // 以下释放尾缓冲区。注意：头缓冲区保留
        data_allocator::deallocate(finish.first, buffer_size());
    }
}

```



```

else    // 只有一个缓冲区
    destroy(start.cur, finish.cur);    // 将此唯一缓冲区内的所有元素析构
    // 注意：并不释放缓冲区空间，这唯一的缓冲区将保留

    finish = start;    // 调整状态
}

```

### **Deque::erase()**

Erase()清除[first, last)区间的所有元素.

```
template <class T, class Alloc, size_t BufSize>
```

```
deque<T, Alloc, BufSize>::iterator
```

```
deque<T, Alloc, BufSize>::erase(iterator first, iterator last) 该算法值得一看
```

```

{
    if (first == start && last == finish)    // 如果清除区间是整个 deque
    {
        clear();                            // 直接调用 clear()即可
        return finish;
    }
    else
    {
        difference_type    n = last - first;    // 清除区间的长度
        difference_type    elems_before = first - start;    // 清除区间前方的元素个数
        if (elems_before < (size() - n) / 2)    // 如果前方的元素个数比较少
        {
            copy_backward(start, first, last);    // 向后移动前方元素（覆盖清除区间）
            iterator new_start = start + n;        // 标记 deque 的新起点
            destroy(start, new_start);            // 移动完毕，将冗余的元素析构
            // 以下将冗余的缓冲区释放
            for (map_pointer cur = start.node; cur < new_start.node; ++cur)
                data_allocator::deallocate(*cur, buffer_size());
            start = new_start;    // 设定 deque 的新起点
        }
        else    // 如果清除区间后方的元素个数比较少
        {
            copy(last, finish, first);    // 向前移动后方元素（覆盖清除区间）
            iterator new_finish = finish - n;    // 标记 deque 的新尾点
            destroy(new_finish, finish);        // 移动完毕，将冗余的元素析构
            // 以下将冗余的缓冲区释放
            for (map_pointer cur = new_finish.node + 1; cur <= finish.node; ++cur)
                data_allocator::deallocate(*cur, buffer_size());
            finish = new_finish;    // 设定 deque 的新尾点
        }
        return start + elems_before;
    }
}

```

```
}
```

### **Deque::insert()**

```
iterator insert(iterator position, const value_type& x)
{
    // 如果是在 deque 的最前端插入, 那么直接 push_front()即可
    if (position.cur == start.cur)
    {
        push_front(x);
        return start;
    }
    // 如果是在 deque 的末尾插入, 直接调用 push_back()
    else if (position.cur == finish.cur)
    {
        push_back(x);
        iterator tmp = finish;
        --tmp;
        return tmp;
    }
    else
    {
        return insert_aux(position, x);
    }
}
```

```
template <class T, class Alloc, size_t BufSize>
```

```
typename deque<T, Alloc, BufSize>::iterator
```

```
deque<T, Alloc, BufSize>::insert_aux(iterator pos, const value_type& x)
```

```
{
    difference_type index = pos - start;    // 插入点之前的元素个数
    value_type x_copy = x;

    if (index < size() / 2)    // 如果插入点之前的元素个数比较少
    {
        push_front(front());    // 在最前端加入与第一元素同值的元素
        iterator front1 = start;    // 以下标示记号, 然后进行元素移动
        ++front1;
        iterator front2 = front1;
        ++front2;
        pos = start + index; //由于 start 在 push_front 之后改变了, 所以这里的 pos 不再是之
                               前的 pos, 而是在之前的 pos 前面一个位置
        iterator pos1 = pos;
        ++pos1;
        copy(front2, pos1, front1);    // 元素移动
    }
}
```

```

    }
else    // 插入点之后的元素个数比较少
{
    push_back(back());           // 在最尾端加入与最后元素同值的元素
    iterator back1 = finish;    // 以下标示记号，然后进行元素移动
    --back1;
    iterator back2 = back1;
    --back2;
    pos = start + index;
    copy_backward(pos, back2, back1);    // 元素移动
}
*pos = x_copy;    // 在插入点上设定新值
return pos;
}

```

在 4.5 和 4.6 中我们将会看到，SGI STL 以 deque 作为 stack/queue 的底层结构，因为 deque 可以很容易的封住其中某个方向的接口，而且 deque 便于扩容，底层结合了 vector 和 list，比起单一使用 list 或者 vector 来实现 stack/queue 实现更高效。

## 4.5 stack

### 4.5.1 stack 概述

先进后出（FILO）

### 4.5.2 stack 定义完整列表

SGI STL 以 deque 作为默认情况下的 stack 底部结构（因为将 deque 头端开口封闭即可形成先进后出的特性）。

通过修改某种接口形成另一种风貌的，称之为 adapter，因此 STL stack 往往不被归类为 container，而被归类为 container adapter。

```

template <class T, class Sequence = deque<T> >
class stack {
    // 以下的 __STL_NULL_TMPL_ARGS 会开展为 template<>, 见 1.9.1 节
    friend bool operator== __STL_NULL_TMPL_ARGS (const stack&, const stack&);
    friend bool operator< __STL_NULL_TMPL_ARGS (const stack&, const stack&);
public:
    typedef typename Sequence::value_type value_type;
    typedef typename Sequence::size_type size_type;
    typedef typename Sequence::reference reference;
    typedef typename Sequence::const_reference const_reference;
protected:
    Sequence c; //底层容器
public:
    // 以下完全利用 Sequence c 的操作，完成 stack 的操作。
    bool empty() const { return c.empty(); }

```

```

    size_type size() const { return c.size(); }
    Reference top() { return c.back(); }
    const_reference top() const { return c.back(); }
    // deque 是两头可进出，stack 是末端进，末端出（所以后进者先出）。
    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_back(); }
};

template <class T, class Sequence>
bool operator==(const stack<T, Sequence>& x, const stack<T, Sequence>& y)
{
    return x.c == y.c;
}

template <class T, class Sequence>
bool operator<(const stack<T, Sequence>& x, const stack<T, Sequence>& y)
{
    return x.c < y.c;
}

```

#### 4.5.3 stack 没有迭代器

Stack 所有元素都必须符合“先进后出”特性，只有 stack 顶端的元素才有机会被访问，所以 stack 不提供迭代器。

#### 4.5.4 以 list 作为 stack 的底层容器

除了 deque 之外，list 也是双向开口的数据结构。上述 stack 源码中使用的底层容器的函数有 empty, size, back, push\_back, pop\_back，凡此种 list 都具备。因此若以 list 为底部结构并封闭其头端开口，一样能够轻易形成一个 stack。

### 4.6 queue

#### 4.6.1 queue 概述

Queue 先进先出（FIFO）

#### 4.6.2 queue 定义完整列表

SGI STL 默认以 deque 作为底部结构并封闭其底端的出口和前端的入口（注意这里的前端和底端可能与我想象的前端和底端相反，但是无所谓，只要知道 deque 是双端都可以输入输出，我们要构造 queue 就要让某一端入口关闭，另外一端出口关闭）

同样的，STL queue 被归类为 container adapter。

```

template <class T, class Sequence = deque<T> >
class queue {
    // 以下的 __STL_NULL_TMPL_ARGS 会开展为 <>, 见 1.9.1 节
    friend bool operator==( __STL_NULL_TMPL_ARGS (const queue& x, const queue& y);
    friend bool operator< __STL_NULL_TMPL_ARGS (const queue& x, const queue& y);

```

```

public:
    typedef typename Sequence::value_type value_type;
    typedef typename Sequence::size_type size_type;
    typedef typename Sequence::reference reference;
    typedef typename Sequence::const_reference const_reference;
protected:
    Sequence c; //底层容器
public:
    // 以下完全利用 Sequence c 的操作，完成 queue 的操作。
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    Reference front() { return c.front(); }
    const_reference front() const { return c.front(); }
    Reference back() { return c.back(); }
    const_reference back() const { return c.back(); }
    // deque 是两头可进出，queue 是末端进，前端出（所以先进者先出）。
    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_front(); }
};

template <class T, class Sequence>
bool operator==(const queue<T, Sequence>& x, const queue<T, Sequence>& y)
{
    return x.c == y.c;
}
template <class T, class Sequence>
bool operator<(const queue<T, Sequence>& x, const queue<T, Sequence>& y)
{
    return x.c < y.c;
}

```

#### 4.6.3 queue 没有迭代器

#### 4.6.4 以 list 作为 queue 的底层容器

### 4.7 heap

#### 4.7.1 heap 概述

SGI STL 使用 binary heap 作为 priority queue 的底层机制。Binary heap 是一种完全二叉树。对于完全二叉树中元素的存储可以使用 array（通常 array[0]留作它用，不存放元素），array[i]存放父节点，那么 array[2i]处存放左孩子节点，array[2i+1]存放右孩子节点。于是乎，我们可以用 array 来实现 heap 算法，但是 array 的缺点是无法动态改变大小，而 heap 需要这项功能，所以以 vector 代替 array。

```
template <class RandomAccessIterator, class Distance, class T>
inline void __push_heap_aux(RandomAccessIterator first,
```

```

        RandomAccessIterator last, Distance*, T*) {
//Distance((last - first) - 1)获取 binary heap 最末端元素索引
//Distance(0)获取 binary heap 最顶端元素索引，由此可知最顶端元素索引为 0
//T(*(last - 1))获取 binary heap 最末端元素值
__push_heap(first, Distance((last - first) - 1), Distance(0), T(*(last - 1)));
}

//以下这组 push_back()不允许指定「大小比较标准」
template <class RandomAccessIterator, class Distance, class T>
void __push_heap(RandomAccessIterator first, Distance holeIndex, Distance topIndex, T value)
{
    //holeIndex 是上溯过程中代表新插入节点的节点索引，topIndex 是最顶节点索引
    Distance parent = (holeIndex - 1) / 2; //找出父节点
    while (holeIndex > topIndex && *(first + parent) < value) {
        // 当尚未到达顶端，且父节点小于新值（于是不符合 heap 的次序特性）
        // 由于以上使用 operator<，可知 STL heap 是一种 max-heap（大者为父）。
        *(first + holeIndex) = *(first + parent); //令洞值为父值
        holeIndex = parent; //调整洞号（上溯过程中代表新插入节点的节点索引），向上提
        //升至父节点。
        parent = (holeIndex - 1) / 2; //新洞的父节点
    } // 持续至顶端，或满足 heap 的次序特性为止。
    *(first + holeIndex) = value; //令洞值为新值，完成安插动作。
}

```

### Pop\_heap 算法

图 4-7-2-2 是 pop\_heap 算法的图示。既然是 max-heap，pop 动作取走的必然是根节点，为满足完全二叉树的条件，必须将最下一层最右边的叶节点拿掉，并为之寻找一个合适位置。我们先将这个最右叶节点放在根节点的位置，然后执行下放程序：将该节点与其两个孩子节点的值进行比较，并与较大子节点对调位置，如此一直下放，直到这个洞的键值大于左右量子节点或者到达叶节点为止。Pop\_heap 会将最大元素放在 vector 最尾端。

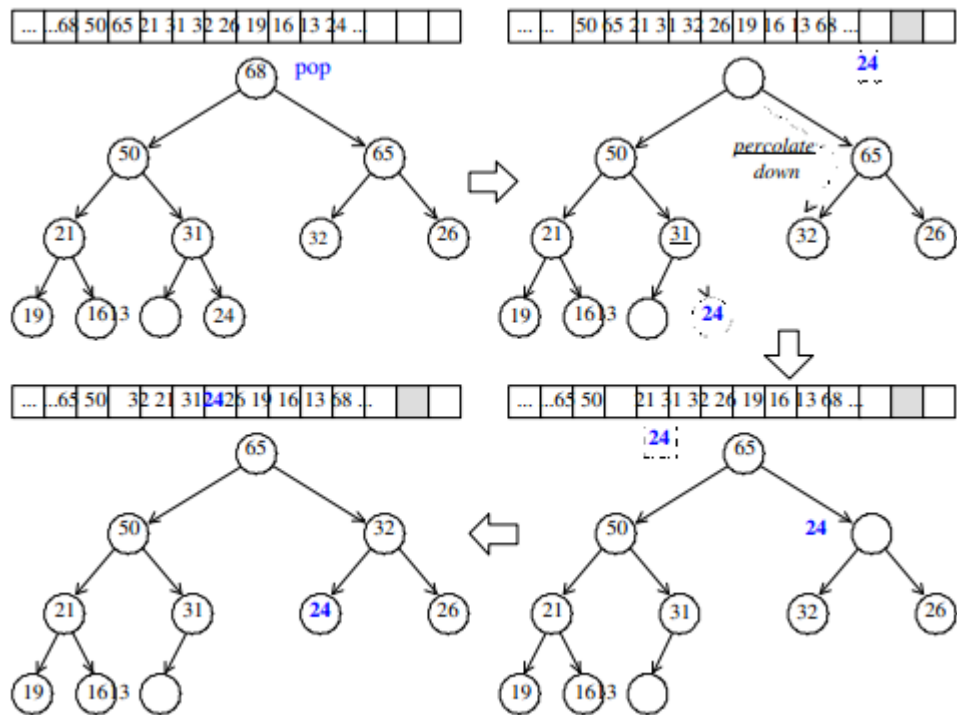


图 4-7-2-2 pop\_heap 算法

```
template <class RandomAccessIterator>
inline void pop_heap(RandomAccessIterator first, RandomAccessIterator last) {
    __pop_heap_aux(first, last, value_type(first));
}

template <class RandomAccessIterator, class T>
inline void __pop_heap_aux(RandomAccessIterator first, RandomAccessIterator last, T*) {
    __pop_heap(first, last-1, last-1, T*(last-1), distance_type(first));
    // 以上, 根据 implicit representation heap 的次序特性, pop 动作的结果
    // 应为底部容器的第一个元素。因此, 首先设定欲调整值为尾值, 然后将首值调至
    // 尾节点 (所以上将迭代器 result 设为 last-1)。然后重整 [first, last-1),
    // 使之重新成一个合格的 heap。
}

//以下这组 __pop_heap() 不允许指定「大小比较标准」
template <class RandomAccessIterator, class T, class Distance>
inline void __pop_heap(RandomAccessIterator first,
                      RandomAccessIterator last,
                      RandomAccessIterator result,
                      T value, Distance*) { //value 被保存, 作为与调整的值 (实为
                                              pop 之前的最后一个元素值)
    *result = *first; // 设定尾值为首值, 尾值即为欲求结果, 可由客端稍后再以底层容器
    __adjust_heap(first, Distance(0), Distance(last - first), value);
    // 以上欲重新调整 heap, 洞号为 0 (亦即树根处), 欲调整值为 value (原尾值)。
```



```
}
```

//以下这个 \_\_adjust\_heap()不允许指定「大小比较标准」

```
template <class RandomAccessIterator, class Distance, class T>
```

```
Void __adjust_heap(RandomAccessIterator first, Distance holeIndex, Distance len, T value)
```

```
{
```

```
    Distance topIndex = holeIndex;
```

```
    Distance secondChild = 2 * holeIndex + 2; //洞节点之右子节点
```

```
    while (secondChild < len) {
```

```
        // 比较洞节点之左右两个子值，然后以 secondChild 代表较大子节点。
```

```
        if (*(first + secondChild) < *(first + (secondChild - 1)))
```

```
            secondChild--; //secondChild 始终代表较大子节点
```

```
        // Percolate down: 令较大子值为洞值，再令洞号下移至较大子节点处。
```

```
        *(first + holeIndex) = *(first + secondChild);
```

```
        holeIndex = secondChild;
```

```
        // 找出新洞节点的右子节点
```

```
        secondChild = 2 * (secondChild + 1);
```

```
    }
```

```
    if (secondChild == len) { //没有右子节点，只有左子节点
```

```
        // Percolate down: 令左子值为洞值，再令洞号下移至左子节点处。
```

```
        *(first + holeIndex) = *(first + (secondChild - 1));
```

```
        holeIndex = secondChild - 1;
```

```
    }
```

```
    // 将欲调整值填入目前的洞号内。注意，此时肯定满足次序特性。
```

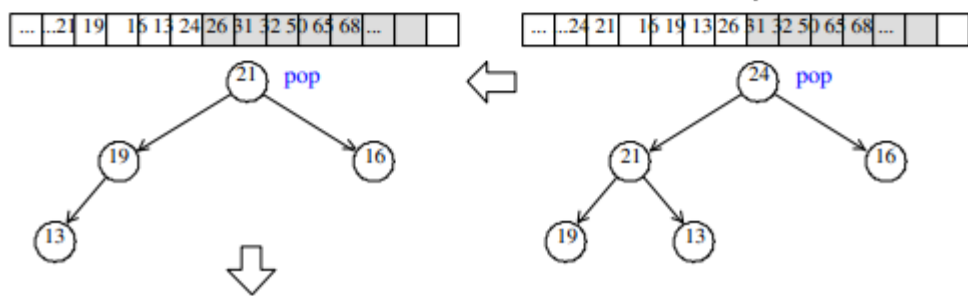
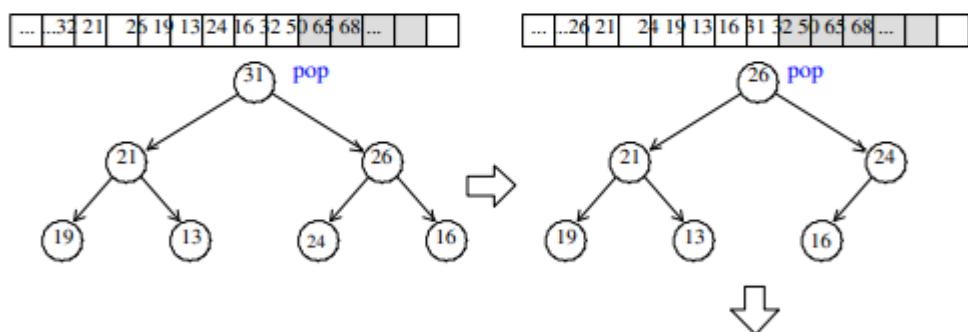
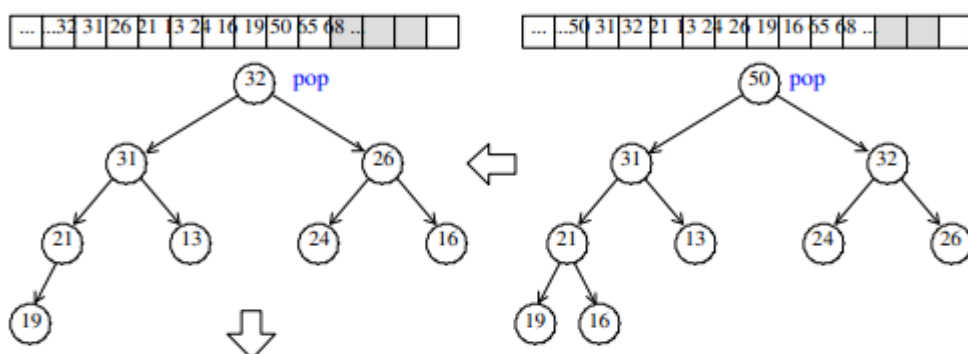
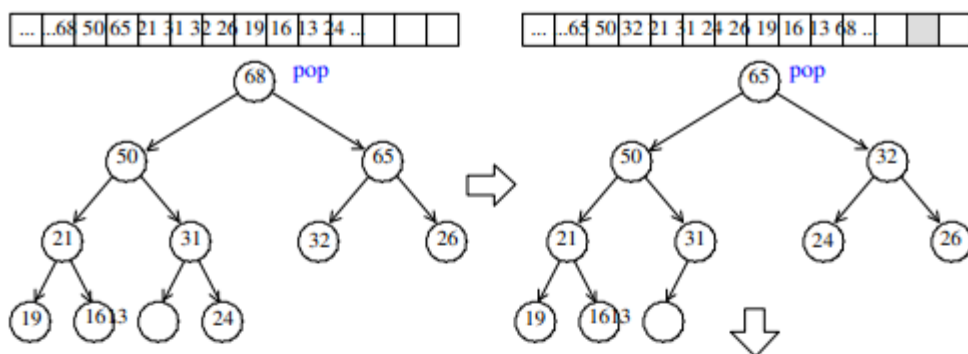
```
    __push_heap(first, holeIndex, topIndex, value);
```

```
}
```

### Sort\_heap 算法

既然每次 pop\_heap 可获得 heap 之中键值最大的元素，如果持续对整个 heap 做 pop\_heap 动作，每次将操作范围从后向前缩减一个元素（因为 pop\_heap 会把键值最大的元素放在底部容器的最尾端），当整个程序执行完毕，我们便有了一个递增序列。

图 4-7-2-3 是 sort\_heap 的图示。



续下页

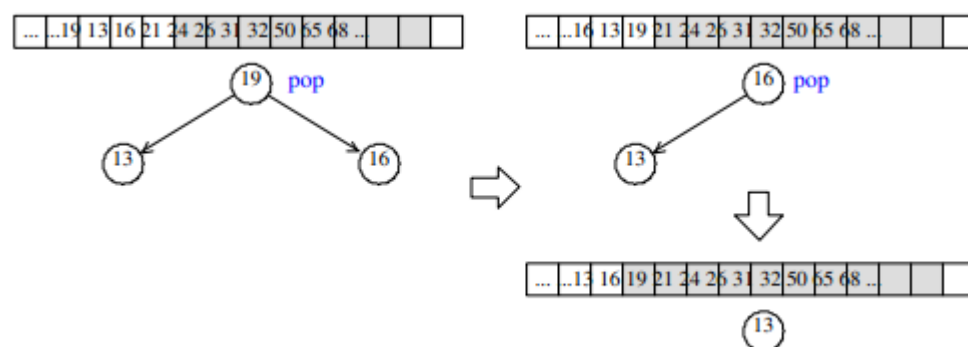


图 4-7-2-3 sort\_heap 算法

```
template <class RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last) {
    // 以下，每执行一次 pop_heap(), 极值（在 STL heap 中为极大值）即被放在尾端。
    // 扣除尾端再执行一次 pop_heap(), 次极值又被放在新尾端。一直下去，最后即得
    // 排序结果。
    while (last - first > 1)
        pop_heap(first, last--); // 每执行 pop_heap() 一次，操作范围即退缩一格。
}
```

### Make\_heap 算法

// 将 [first, last) 排列为一个 heap。

```
template <class RandomAccessIterator>
inline void make_heap(RandomAccessIterator first, RandomAccessIterator last) {
    __make_heap(first, last, value_type(first), distance_type(first));
}
```

// 以下这组 make\_heap() 不允许指定「大小比较标准」。

```
template <class RandomAccessIterator, class T, class Distance>
void __make_heap(RandomAccessIterator first,
                 RandomAccessIterator last, T*,
                 Distance*) {
    if (last - first < 2) return; // 如果长度为 0 或 1，不必重新排列。
    Distance len = last - first;
    // 找出第一个需要重排的子树头部，以 parent 标示出。由于任何叶节点都不需执行
    // perlocate down，所以有以下计算。parent 命名不佳，名为 holeIndex 更好。
    Distance parent = (len - 2) / 2;

    while (true) {
        // 重排以 parent 为首的子树。len 是为了让 __adjust_heap() 判断操作范围
        __adjust_heap(first, parent, len, T*(first + parent));
        if (parent == 0) return; // 走完根节点，就结束。
        parent--; // (即将重排之子树的) 头部向前一个节点
    }
}
```

### 4.7.3 heap 没有迭代器

## 4.8 priority\_queue

### 4.8.1 priority\_queue 概述

Priority queue 在元素的基本操作上和 queue 一致,但是其中的元素并非按照插入到的顺序排列的,而是自动按照元素的权值排列的,缺省情况下,priority queue 是使用一个 max-heap 完成的,所以 priority queue 也是依赖于 vector 的。

由于 priority\_queue 以 heap 完成工作,所以 priority\_queue 也是 container adapter。

### 4.8.2 priority\_queue 完整定义

```
template <class T, class Sequence = vector<T>,
          class Compare = less<typename Sequence::value_type> >
class priority_queue {
public:
    typedef typename Sequence::value_type value_type;
    typedef typename Sequence::size_type size_type;
    typedef typename Sequence::reference reference;
    typedef typename Sequence::const_reference const_reference;
protected:
    Sequence c; //底层容器
    Compare comp; //元素大小比较标准
public:
    priority_queue() : c() {}
    explicit priority_queue(const Compare& x) : c(), comp(x) {}

    //以下用到的 make_heap(), push_heap(), pop_heap()都是泛型算法
    //注意, 任一个构造函数都立刻于底层容器内产生一个 implicit representation heap。
    template <class InputIterator>
    priority_queue(InputIterator first, InputIterator last, const Compare& x)
    : c(first, last), comp(x) { make_heap(c.begin(), c.end(), comp); }

    template <class InputIterator>
    priority_queue(InputIterator first, InputIterator last)
    : c(first, last) { make_heap(c.begin(), c.end(), comp); }

    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    const_reference top() const { return c.front(); }
    void push(const value_type& x) {
        __STL_TRY {
            // push_heap 是泛型算法, 先利用底层容器的 push_back() 将新元素
            // 推入末端, 再重排 heap。见 C++ Primer p.1195。
            // 这里印证了 heap 一节所述, 调用 push_heap 之前, 待插入元素已经 push 到末端
            c.push_back(x);
            push_heap(c.begin(), c.end(), comp); // push_heap 是泛型算法
        }
    }
```

```

    __STL_UNWIND(c.clear());
}

void pop() {
    __STL_TRY {
        // pop_heap 是泛型算法，从 heap 内取出一个元素。它并不是真正将元素
        // 弹出，而是重排 heap，然后再以底层容器的 pop_back() 取得被弹出
        // 的元素。见 C++ Primer p.1195。
        // 类似 push，先 pop_back 将元素放在底层容器的最末端，然后调用 pop_back 取出
        // 元素
        pop_heap(c.begin(), c.end(), comp);
        c.pop_back();
    }
    __STL_UNWIND(c.clear());
}
};

```

#### 4.8.3 priority\_queue 没有迭代器

### 4.9 slist

#### 4.9.1 slist 概述

STL 中 list 是个双向链表，SGI STL 提供了另外一个单向链表 slist (single linked list)。

slist 和 list 的主要区别在于：前者的迭代器是单向的 forward iterator，后者是 bidirectional iterator。

Slist 和 list 共同特色是：他们的插入、删除、接合 (splice) 等动作都不会造成原有迭代器的失效 (当然啦，指向被移除元素的那个迭代器，在移除动作发生之后肯定是会失效的)

4.9.2 --- 4.9.5 不在此分析。

## 第 5 章 关系型容器 (associated containers)

标准的 stl 关联式容器分为 set 和 map 两大类, 以及这两大类的衍生体 multiset 和 multimap, 这些容器的底层机制均以 RB-tree 来实现, RB-tree 也是一个独立容器, 并不开放给外界使用。此外, SGI STL 还提供了一个不在标准之内的关联式容器 hash-table, 以及以之为底层机制而完成的 hash-set, hash-map, hash-multiset, hash-multimap。

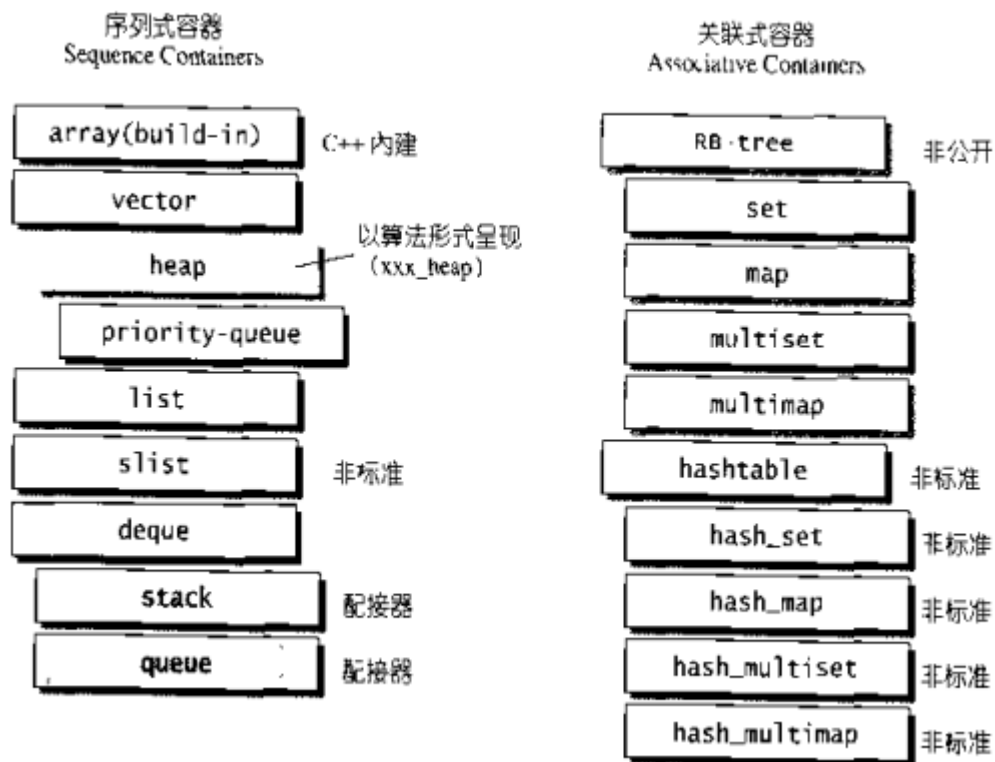


图 5-1 SGI STL 的各种容器, 本图以内缩方式来表达基层与衍生层的关系。这里所谓的衍生, 并非继承关系, 二是内含关系。例如 heap 内含一个 vector, priority\_queue 内含一个 heap, stack 和 queue 都内含一个 deque, set/map/multiset/multimap 都内含一个 RB-tree, hash\_x 都内含一个 hash\_table。

所谓关联式容器, 观念上类似于关联式数据库: 每笔数据都有一个键值 (key) 和一个实际值 (value)。当元素被插入容器时, 内部机制根据键值, 按着一定的规则将元素置于特定的位置。关联式容器没有所谓头尾的概念 (只有最大元素, 最小元素), 所以不会有类似 push\_back(), push\_front() 这样的操作。

### 5.1 树的概览

路径长度: 根节点至任何节点之间有唯一的路径, 路径所经过的边数, 称为路径长度。

节点的深度: 根节点至任一节点的路径长度。根节点的深度永远为 0。

节点的高度: 某节点至其最深子节点的路径长度。根节点的高度, 作为整棵树的高度。

节点的大小: 节点的任何子节点 (包括自己) 及其子节点的子节点的节点总数目。

size of E : 3  
height of E : 1  
depth of E : 2  
length of E : 2

size of C : 4  
height of C : 2  
depth of C : 1  
length of C : 1

size of A : 12  
height of A : 3  
depth of A : 0  
length of A : 0

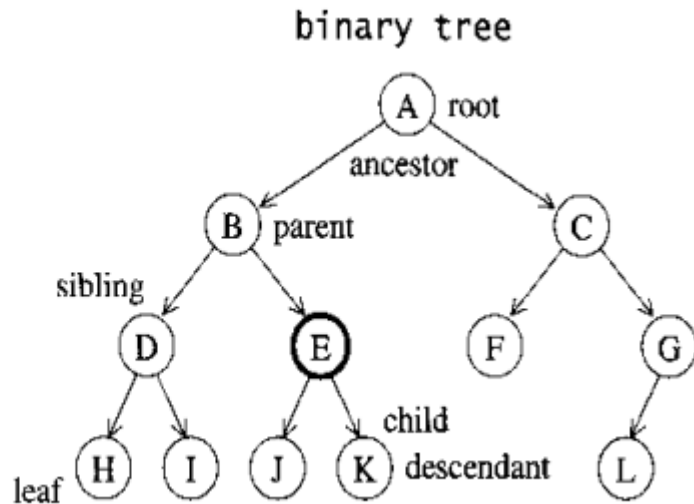


图 5-2 树状结构的相关术语展示

### 5.1.1 二叉搜索树

一般而言，关联式容器的内部结构是二叉搜索树（binary balanced tree），BBT 有很多种类，包括 RB-Tree, AVL-Tree, AA-Tree，其中最被广泛运用的是 RB-tree。

二叉搜索树的规则是：任何结点的键值都大于其左子树中每个结点的键值，而小于其右子树中每一个结点的键值。二叉搜索树支持的操作时间复杂度与树的高度成正比，如果树是平衡的（极端是完全二叉树），其操作效率就高，不平衡（极端是单链），其操作效率就低。在二叉搜索树中查找最大/最小元素，只用沿着根节点一直往右/左走即可。比较麻烦的是插入和删除。

图 5-5 是二叉搜索树的插入操作图解。插入新元素时，从根节点开始，遇到键值较大者就向左，否则向右，一直到尾端，即为插入点。

图 5-6 是二叉搜索树的删除操作图解。欲删除旧节 A，可分两种情况：如果 A 只有一个子节点，就直接将 A 的子节点连至 A 的父节点，并将 A 删除；如果 A 有两个子节点，就以右子树内的最小节点取代 A。

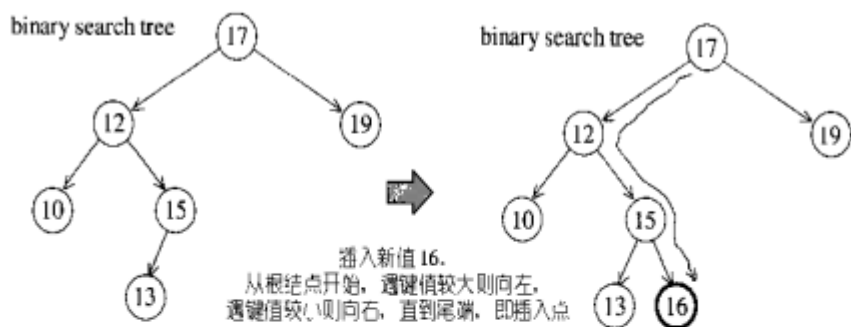


图 5-5 二叉搜索树的节点插入操作

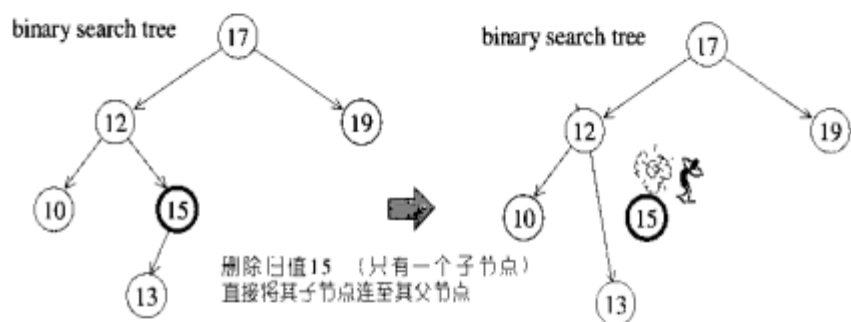


图 5-6a 二叉搜索树的节点删除操作之一（目标节点只有一个子节点）

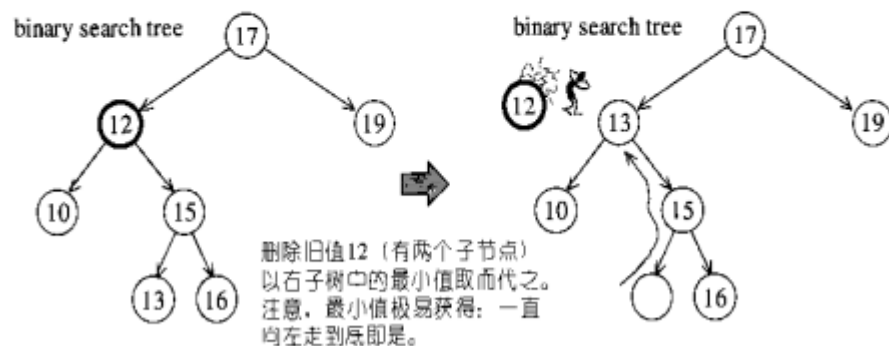


图 5-6b 二叉搜索树的节点删除操作之二（目标节点有两个子节点）

### 5.1.2 平衡二叉搜索树

AVL-tree, RB-tree, AA-tree 均是平衡二叉树（所谓平衡，就是没有任何一个节点过深）。

### 5.1.3 AVL tree (adelson-veskil-landis tree)

AVL tree 要求任何节点的左右子树高度相差最多 1. 这样就能保证整棵树的深度为  $O(\log N)$ 。

图 5-8 左侧所示的是一颗 AVL tree，插入节点 11 之后，灰色节点违反 AVL tree 的平衡条件，由于只有“插入点至根节点”路径上的各节点可能改变平衡状态，因此，只要调整其中最深的那个节点，便可使整棵树重新获得平衡。



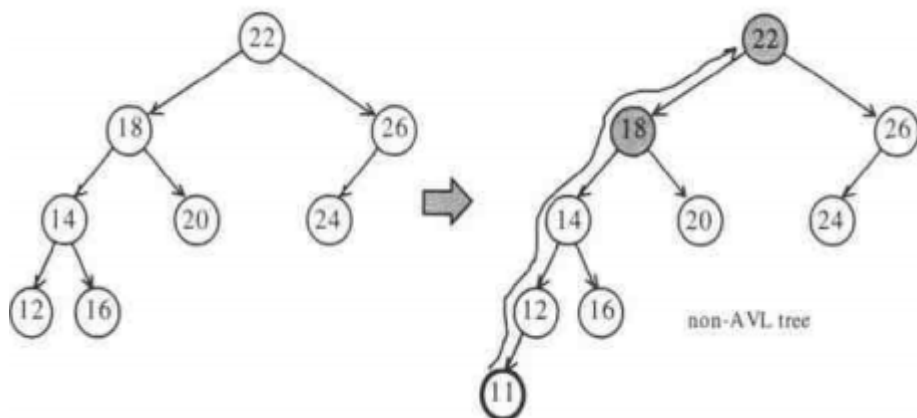


图 5-8 图左是 AVL tree。插入节点 11 后，图右灰色节点违反 AVL tree 条件

前面说过，只要调整“插入节点至根节点”路径上，平衡状态被破坏之各节点中最深的那一个，便可使整棵树重新获得平衡，假设该最深节点为  $x$ ，由于节点最多拥有两个子节点，而所谓“平衡破坏”，则意味着  $x$  的左右两棵子树的高度相差 2，因此有四种情况：

1. 插入节点位于  $x$  的左子结点的左子树 --- 左左；
2. 插入节点位于  $x$  的左子结点的右子树 --- 左右；
3. 插入节点位于  $x$  的右子结点的左子树 --- 右左；
4. 插入节点位于  $x$  的右子结点的右子树 --- 右右；

情况 1,4 彼此对称，称为外侧插入，可以采用单旋转调整解决；情况 2,3 彼此对称，称为内侧插入，可以采用双旋转调整解决。

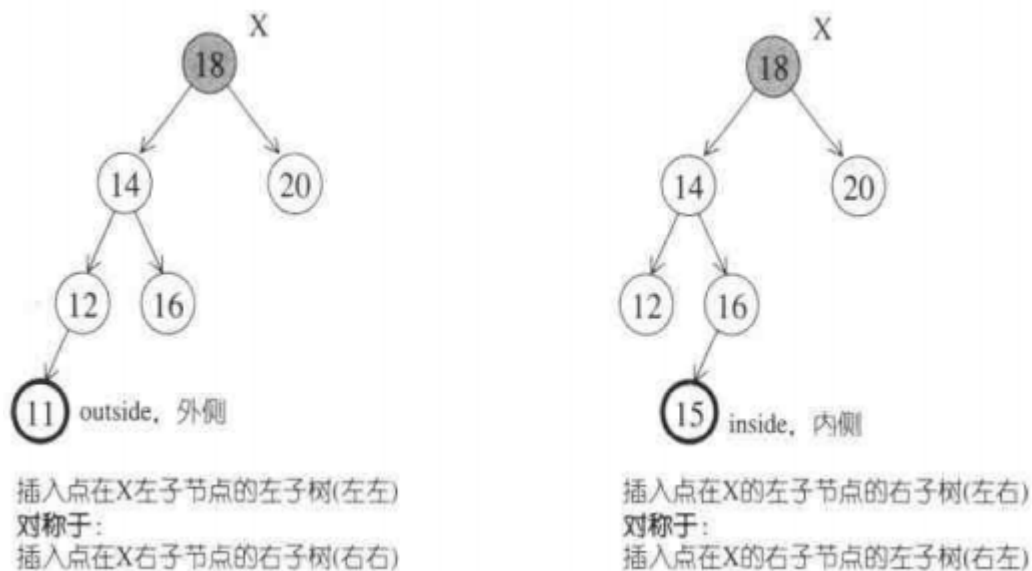


图 5-9 AVL-tree 的四种“平衡破坏”情况

#### 5.1.4 单旋转

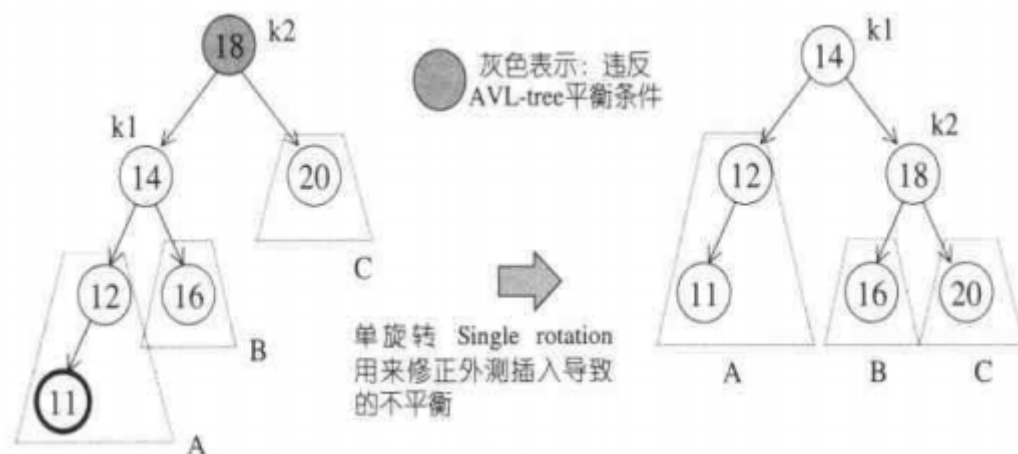


图 5-10 延续图 5-8 的状况，以“单旋转”修正外侧插入导致的不平衡

### 5.1.5 双旋转

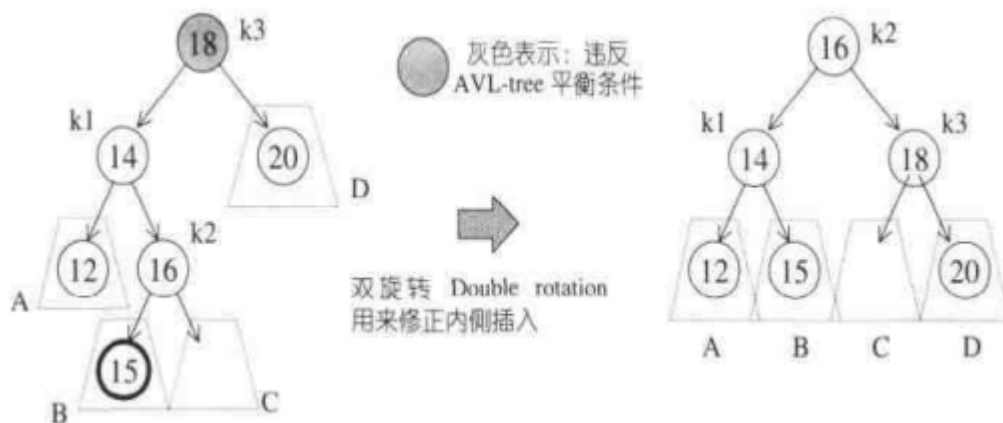


图 5-11 延续图 5-8 的状况，以双旋转修正因内侧插入导致的不平衡。

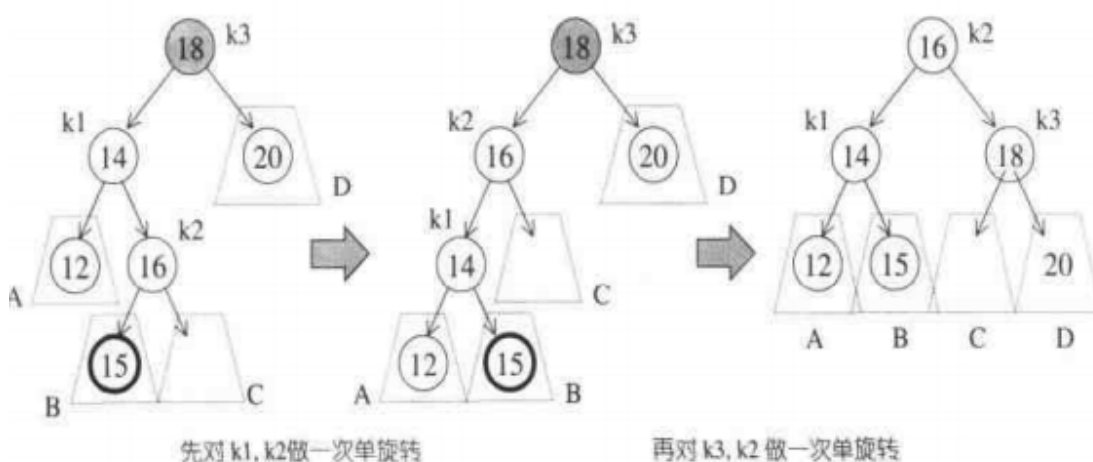


图 5-12 双旋转（如图 5-11）可由两次单旋转合并而成

单旋转和双旋转实现代码：

对照着上述的旋转方法，给出基本的旋转操作（暂时不考虑其他节点为空的情况）

Void singleRotation( Node\* n,bool left) //n 节点表示当前节点违反了平衡条件, left 表示新插入节点在 n 的左子树还是右子树

```
{
    Node *newHead;
    If(left==TRUE) //新插入的节点在 n 的左子树, 左左(因为我们确信当执行 singleRotation
        时, 只可能是左左/右右)
    {
        newHead = n->left;          //在这里指 k2
        if(newHead->right!=null)
            n->left=newHead->right;
        newHead->right=n;
        n=newHead;
    }
    Else                                //右右, 进行左上提拉节点
    {
        newHead = n->right;
        if(newHead->left!=null)
            n->right= newHead->left;
        newHead->left = n;
        n= newHead;
    }
}
```

Void doubleRotation(Node\* n, bool left)//n 节点表示当前节点违反了平衡条件, left 表示先插入节点在 n 的左子树还是右子树

```
{
    If(left == TRUE) //新插入的节点在 n 的左子树, 左右
    {
        singleRotation(n->left,false); //第一次旋转, 对左子树进行一次左旋, 如下图所示
        singleRotation(n,true);        //第二次旋转, 然后对该节点进行一次右旋
    }
    Else
    {
        singleRotation(n->right,true);
        singleRotation(n,false);
    }
}
```

## 5.2 RB-tree

Refer to: <http://blog.csdn.net/hackbuteer1/article/details/7740956>

RB-tree 不仅是一个二叉搜索树, 而且必须满足:

- 1.每个节点不是红色就是黑色(图中神色底纹代表黑色, 浅色底纹代表红色);
- 2.跟节点为黑色;
- 3.如果某个节点为红色, 则其子节点必须为黑色

4.任一节点至 NULL（树尾端）的任何路径，所含之黑节点数必须相同。（算法导论中陈述为：对于每个节点，从该节点到其子孙节点的所有路径上包含相同数目的黑色节点）

根据规则 4，新增节点必须为红；根据规则 3，新增节点之父节点必须为黑（因为如果父节点为红，则子节点必为黑）；如果新节点根据二叉搜索树的规则到达插入点后，未能符合上述条件，就必须调整颜色并旋转树形。见图 5-13 说明。

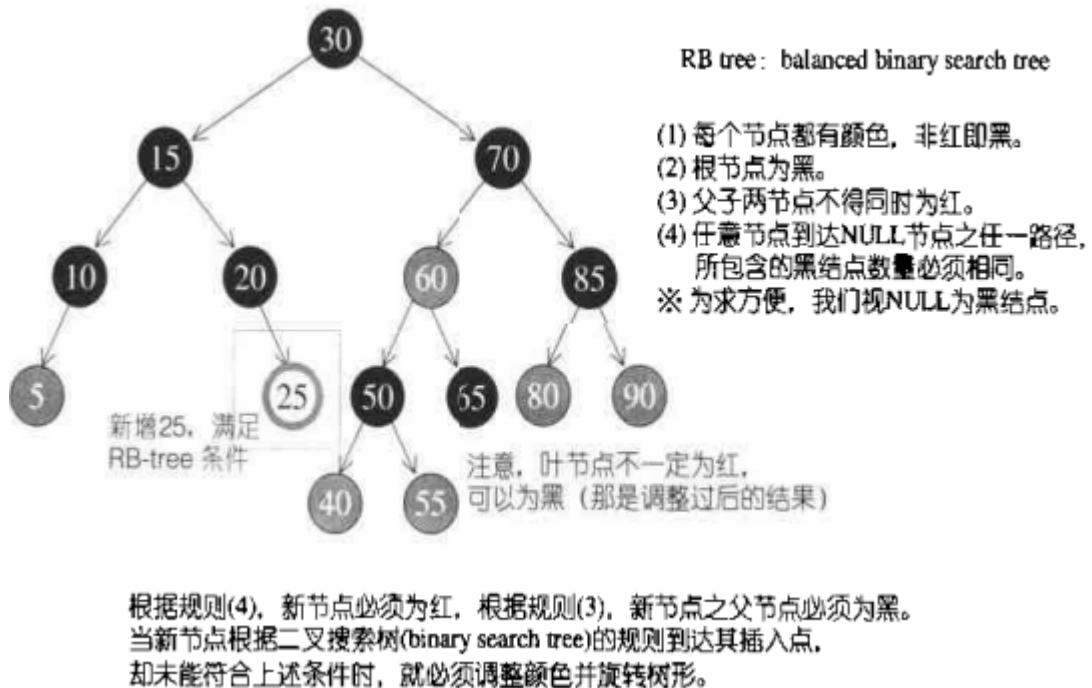


图 5-13 RB-tree 的条件与实例

### 5.2.1 插入节点

关于插入节点我们打算从两个地方分别学习：

一是博客 <http://blog.csdn.net/hackbuteer1/article/details/7740956>

二是 STL 源码分析一书，但是鉴于该书中讲解不太明晰，所以下面只学习博客中的。

下面是学习博客 <http://blog.csdn.net/hackbuteer1/article/details/7740956> 中讲解。

在讨论红黑树的插入操作之前必须要明白，任何一个即将插入的新结点的初始颜色都为红色。这一点很容易理解，因为插入黑点会增加某条路径上黑结点的数目，从而导致整棵树黑高度的不平衡。但如果新结点的父结点为红色时（如下图所示），将会违反红黑树的性质：一条路径上不能出现相邻的两个红色结点。这时就需要通过一系列操作来使红黑树保持平衡。

为了清楚地表示插入操作以下在结点中使用“新”字表示一个新插入的结点；使用“父”字表示新插入点的父结点；使用“叔”字表示“父”结点的兄弟结点；使用“祖”字表示“父”结点的父结点。插入操作分为以下几种情况：

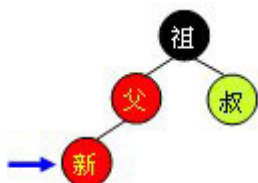
#### 1、黑父

如下图所示，如果新节点的父结点为黑色结点，那么插入一个红点将不会影响红黑树的平衡，此时插入操作完成。红黑树比 AVL 树优秀的地方之一在于黑父的情况比较常见，从而使红黑树需要旋转的几率相对 AVL 树来说会少一些。



## 2、红父

如果新节点的父结点为红色，这时就需要进行一系列操作以保证整棵树红黑性质。如下图所示，由于父结点为红色，此时可以判定，祖父结点必定为黑色。因此可以将问题放到以祖父结点为根节点的子树中来解决：只要不改变子树的黑高度，就不会对树的其他部分产生影响。这时需要根据叔父结点的颜色来决定做什么样的操作。青色结点表示颜色未知。由于有可能需要根结点到新点的路径上进行多次旋转操作，而每次进行不平衡判断的起始点（我们可将其视为新点）都不一样。所以我们在此使用一个蓝色箭头指向这个起始点，并称之为判定点。



### 2.1 红叔

当叔父结点为红色时，如下图所示，无需进行旋转操作，只要将父和叔结点变为黑色，将祖父结点变为红色即可（**关于这里为什么要将祖父节点变为红色呢？**本来如果将父节点和叔父节点变为黑色，这样可以满足祖父节点到它的子孙节点的所有路径上包含相同数目的黑色节点，但是可能导致祖父节点的祖先节点到其子孙节点的所有路径上黑色节点数目不同（也就是黑高度不同），因为很明显经由祖父节点的路径上黑色节点数目增加了1）。但由于祖父结点的父结点有可能为红色，从而违反红黑树性质。此时必须将祖父结点作为新的判定点继续向上（**迭代**）进行平衡操作。

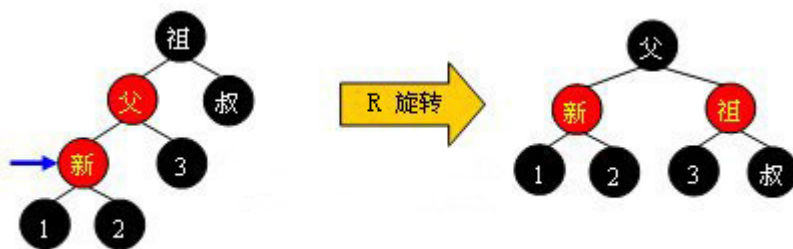


需要注意的是，无论“父节点”在“叔节点”的左边还是右边，无论“新节点”是“父节点”的左孩子还是右孩子，它们的操作都是完全一样的（其实这种情况包括4种，只需调整颜色，不需要旋转树形）。

### 2.2 黑叔

当叔父结点为黑色时（这里叔父节点为黑，可能是 NULL，NULL 节点被认为是黑色的），需要进行旋转（在旋转过程中，将最终的子树根节点都设为黑色，然后考虑黑高度不要改变，就很好理解下面的过程了），以下图示了所有的旋转可能：

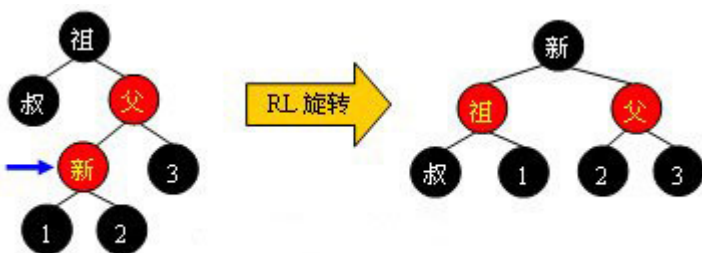
Case 1:



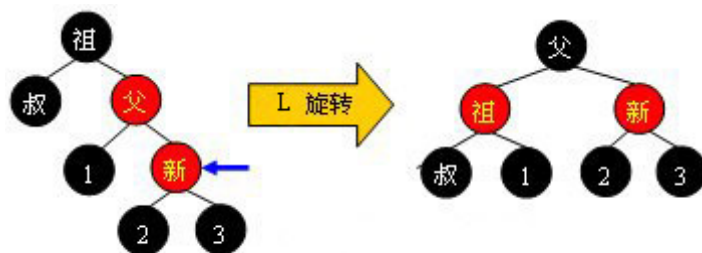
Case 2:



Case 3:



Case 4:



可以观察到，当旋转完成后，新的旋转根全部为黑色，此时不需要再向上回溯进行平衡操作，插入操作完成。需要注意，上面四张图的“叔”、“1”、“2”、“3”结点有可能为黑哨兵结点。

**红黑树的插入操作代码如下：**

```

// 元素插入操作 insert_unique()
// 插入新值：节点键值不允许重复，若重复则插入无效
// 注意，返回值是个 pair，第一个元素是个红黑树迭代器，指向新增节点
// 第二个元素表示插入成功与否
template<class Key , class Value , class KeyOfValue , class Compare , class Alloc>
pair<typename rb_tree<Key , Value , KeyOfValue , Compare , Alloc>::iterator , bool>
rb_tree<Key , Value , KeyOfValue , Compare , Alloc>::insert_unique(const Value &v)
{
    rb_tree_node* y = header;    // 根节点 root 的父节点
    rb_tree_node* x = root();    // 从根节点开始
    bool comp = true;
    while(x != 0)
    {
        y = x;
        comp = key_compare(KeyOfValue()(v) , key(x));    // v 键值小于目前节点之键值?
        x = comp ? left(x) : right(x);    // 遇“大”则往左，遇“小于或等于”则往右
    }
    // 离开 while 循环之后，y 所指即插入点之父节点（此时的它必为叶节点）
    iterator j = iterator(y);    // 令迭代器 j 指向插入点之父节点 y
    if(comp)    // 如果离开 while 循环时 comp 为真（表示遇“大”，将插入于左侧）
    {
        if(j == begin())    // 如果插入点之父节点为最左节点
            return pair<iterator , bool>(_insert(x , y , z) , true);
        else    // 否则（插入点之父节点不为最左节点）
            --j;    // 调整 j，回头准备测试
    }
    if(key_compare(key(j.node) , KeyOfValue()(v) ))
        // 新键值不与既有节点之键值重复，于是以下执行安插操作
        return pair<iterator , bool>(_insert(x , y , z) , true);
    // 以上，x 为新值插入点，y 为插入点之父节点，v 为新值

    // 进行至此，表示新值一定与树中键值重复，那么就不应该插入新值
    return pair<iterator , bool>(j , false);
}

// 真正地插入执行程序 _insert()
template<class Key , class Value , class KeyOfValue , class Compare , class Alloc>
typename<Key , Value , KeyOfValue , Compare , Alloc>::_insert(base_ptr x_ , base_ptr y_ , const
Value &v)
{
    // 参数 x_ 为新值插入点，参数 y_ 为插入点之父节点，参数 v 为新值
    link_type x = (link_type) x_;
    link_type y = (link_type) y_;
    link_type z;

```

```

// key_compare 是键值大小比较准则。应该会是个 function object
if(y == header || x != 0 || key_compare(KeyOfValue()(v), key(y)))
{
    z = create_node(v);    // 产生一个新节点
    left(y) = z;           // 这使得当 y 即为 header 时，leftmost() = z
    if(y == header)
    {
        root() = z;
        rightmost() = z;
    }
    else if(y == leftmost())    // 如果 y 为最左节点
        leftmost() = z;        // 维护 leftmost(), 使它永远指向最左节点
}
else
{
    z = create_node(v);    // 产生一个新节点
    right(y) = z;          // 令新节点成为插入点之父节点 y 的右子节点
    if(y == rightmost())
        rightmost() = z;    // 维护 rightmost(), 使它永远指向最右节点
}
parent(z) = y;            // 设定新节点之父节点
left(z) = 0;              // 设定新节点的左子节点
right(z) = 0;             // 设定新节点的右子节点
// 新节点的颜色将在 _rb_tree_rebalance() 设定 (并调整)
_rb_tree_rebalance(z, header->parent);    // 参数一为新增节点, 参数二为根节点 root
++node_count;            // 节点数累加
return iterator(z);       // 返回一个迭代器, 指向新增节点
}

```

// 全局函数

// 重新令树形平衡 (改变颜色及旋转树形)

// 参数一为新增节点, 参数二为根节点 root

**inline void \_rb\_tree\_rebalance(\_rb\_tree\_node\_base\* x, \_rb\_tree\_node\_base\*& root)**

```

{
    x->color = _rb_tree_red;    // 新节点必为红
    while(x != root && x->parent->color == _rb_tree_red)    // 父节点为红
    {
        if(x->parent == x->parent->parent->left)    // 父节点为祖父节点之左子节点
        {
            _rb_tree_node_base* y = x->parent->parent->right;    // 令 y 为伯父节点
            if(y && y->color == _rb_tree_red)    // 伯父节点存在, 且为红
            {

```



```

        x->parent->color = _rb_tree_black;           // 更改父节点为黑色
        y->color = _rb_tree_black;                   // 更改伯父节点为黑色
        x->parent->parent->color = _rb_tree_red;      // 更改祖父节点为红色
        x = x->parent->parent;                        // 准备继续往上层检查
    }
else // 无伯父节点，或伯父节点为黑色
{
    if(x == x->parent->right) // 如果新节点为父节点之右子节点
    {
        x = x->parent;
        _rb_tree_rotate_left(x, root); // 第一个参数为左旋点
    }
    x->parent->color = _rb_tree_black; // 改变颜色
    x->parent->parent->color = _rb_tree_red;
    _rb_tree_rotate_right(x->parent->parent, root); // 第一个参数为右旋点
}
}
else // 父节点为祖父节点之右子节点
{
    _rb_tree_node_base* y = x->parent->parent->left; // 令 y 为伯父节点
    if(y && y->color == _rb_tree_red) // 有伯父节点，且为红
    {
        x->parent->color = _rb_tree_black;           // 更改父节点为黑色
        y->color = _rb_tree_black;                   // 更改伯父节点为黑色
        x->parent->parent->color = _rb_tree_red;      // 更改祖父节点为红色
        x = x->parent->parent;                        // 准备继续往上层检查
    }
else // 无伯父节点，或伯父节点为黑色
{
    if(x == x->parent->left) // 如果新节点为父节点之左子节点
    {
        x = x->parent;
        _rb_tree_rotate_right(x, root); // 第一个参数为右旋点
    }
    x->parent->color = _rb_tree_black; // 改变颜色
    x->parent->parent->color = _rb_tree_red;
    _rb_tree_rotate_left(x->parent->parent, root); // 第一个参数为左旋点
}
}
}
}
}
}
root->color = _rb_tree_black; // 根节点永远为黑色
}

```

// 左旋函数

```
inline void _rb_tree_rotate_left(_rb_tree_node_base* x , _rb_tree_node_base*& root)
{
    // x 为旋转点
    _rb_tree_node_base* y = x->right;          // 令 y 为旋转点的右子节点
    x->right = y->left;
    if(y->left != 0)
        y->left->parent = x;                    // 别忘了回马枪设定父节点
    y->parent = x->parent;

    // 令 y 完全顶替 x 的地位（必须将 x 对其父节点的关系完全接收过来）
    if(x == root)    // x 为根节点
        root = y;
    else if(x == x->parent->left)    // x 为其父节点的左子节点
        x->parent->left = y;
    else    // x 为其父节点的右子节点
        x->parent->right = y;
    y->left = x;
    x->parent = y;
}
```

// 右旋函数

```
inline void _rb_tree_rotate_right(_rb_tree_node_base* x , _rb_tree_node_base*& root)
{
    // x 为旋转点
    _rb_tree_node_base* y = x->left;          // 令 y 为旋转点的左子节点
    x->left = y->right;
    if(y->right != 0)
        y->right->parent = x;                // 别忘了回马枪设定父节点
    y->parent = x->parent;

    // 令 y 完全顶替 x 的地位（必须将 x 对其父节点的关系完全接收过来）
    if(x == root)
        root = y;
    else if(x == x->parent->right)    // x 为其父节点的右子节点
        x->parent->right = y;
    else    // x 为其父节点的左子节点
        x->parent->left = y;
    y->right = x;
    x->parent = y;
}
```

### 5.2.2 删除节点

算法导论书上给出的红黑树的性质如下，跟 STL 源码剖析书上面的4条性质大同小异。

- 1、每个结点或是红色的，或是黑色的
- 2、根节点是黑色的
- 3、每个叶结点（NIL）是黑色的
- 4、如果一个节点是红色的，则它的两个儿子都是黑色的。
- 5、对于每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑色结点。

从红黑树上删除一个节点，可以先用普通二叉搜索树的方法，将节点从红黑树上删除掉，然后再将被破坏的红黑性质进行恢复。

我们回忆一下普通二叉树的节点删除方法：**Z** 指向需要删除的节点，**Y** 指向**实质结构上被删除的结点**，如果 **Z** 节点只有一个子节点或没有子节点，那么 **Y** 就是指向 **Z** 指向的节点。如果 **Z** 节点有两个子节点，那么 **Y** 指向 **Z** 节点的后继节点（其实前趋也是一样的），而 **Z** 的后继节点绝对不可能有左子树。因此，仅从结构来看，二叉树上实质被删除的节点最多只可能有一个子树。

现在我们来分析红黑性质的恢复过程：

如果 **Y** 指向的节点是个红色节点，那么直接删除掉 **Y** 以后，红黑性质不会被破坏。操作结束。

如果 **Y** 指向的节点是个黑色节点，那么就有几条红黑性质可能受到破坏了。首先是包含 **Y** 节点的所有路径，黑高度都减少了一（第 5 条被破坏）。其次，如果 **Y** 有红色子节点，**Y** 又有红色的父节点，那么 **Y** 被删除后，就出现了两个相邻的红色节点（第 4 条被破坏）。最后，如果 **Y** 指向的是根节点，而 **Y** 的子节点又是红色的，那么 **Y** 被删除后，根节点就变成红色的了（第 2 条被破坏）。

其中，第 5 条被破坏是让我们比较难受的。因为这影响到了全局。这样动作就太大太复杂了。而且在这个条件下，进行其它红黑性质的恢复也很困难。所以我们首先解决这个问题：如果不改变含 **Y** 路径的黑高度，那么树的其它部分的黑高度就必须做出相应的变化来适应它。所以，我们想办法恢复原来含 **Y** 节点的路径的黑高度。做法就是：**无条件的把 Y 节点的颜色，推到它的子节点 X 上去。**（**X** 可能是 NIL 节点）。这样，**X 就可能具有双重黑色（双重黑色是指原来是黑色，现在还是黑色），或同时具有红黑两色（原来是红色，现在变为黑色），**也就是第 1 条性质被破坏了。

但第 1 条性质是比较容易恢复的：

一、如果 **X** 是同时**具有红黑两色**，那么好办，直接把 **X** 涂成黑色，就行了。而且这样把所有问题都解决了。因为将 **X** 变为黑色，2、4 两条如果有问题的话也会得到恢复，算法结束。

二、如果 **X** 是双黑色，那么我们希望把这种情况向上推一直推到根节点（调整树结构和颜色，**X** 的指向新的双黑色节点，**X** 不断向上移动），让根节点具双黑色，这时，直接把 **X** 的一层黑色去掉就行了（因为根节点被包含在所有的路径上，所以这样做所有路径同时黑高减少一，不会破坏红黑特征）。

下面就具体地分析如何恢复1、2、4三个可能被破坏的红黑特性：我们知

道，如果  $X$  指向的节点是有红黑两色，或是  $X$  是根节点时，只需要简单的对  $X$  进行一些改变就行了（不需要对  $x$  以外的节点进行操作）。**要对除  $X$  节点外的其它节点进行操作时，必定是这样的情况：** $X$  节点是双层黑色，且  $X$  有父节点  $P$ 。由此可知， $X$  必然有兄弟节点  $W$ ，而且这个  $W$  节点必定有两个子节点。（因为这是原树满足红黑条件要求而自然具备的。 $X$  为双黑色，那么  $P$  的另一个子节点以下一定要有至少两层的节点，否则黑色高度不可能和  $X$  路径一致）。所以我们就分析这些节点之间如何变形，把问题限制在比较小的范围内解决。另一个前提是： $X$  在一开始，肯定是树底的叶节点或是  $NIL$  节点，所以在递归向上的过程中，每一步都保证下一步进行时，至少  $X$  的子树是满足红黑特性的。因此子树的情况就可以认为是已经正确的了，这样，分析就只限制在  $X$  节点， $X$  的父节点  $P$  和  $X$  的兄弟节点  $W$ ，以及  $W$  的两个子节点这些个节点中。

下面仅仅考虑  $X$  原本是黑色的情况即可。

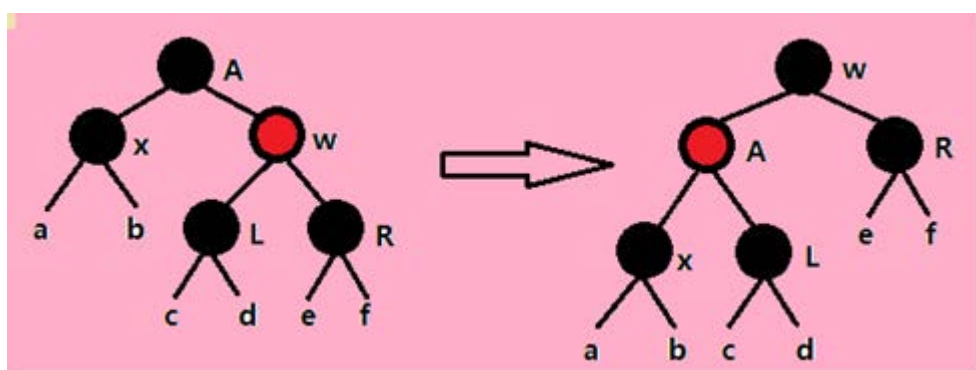
在这种情况下， $X$  此时应该具有双重黑色，算法的过程就是将这多出的一重黑色向上移动，直到遇到红节点或者根节点。

接着往下分析，会遇到4种情况，实际上是8种，因为其中4种是相互对称的，这可以通过判断  $X$  是其父节点的右孩子还是左孩子来区分。下面我们以  $X$  是其父节点的左孩子的情况来分析这4种情况，实际上接下来的调整过程，就是要想方设法将经过  $X$  的所有路径上的黑色节点个数增加1。

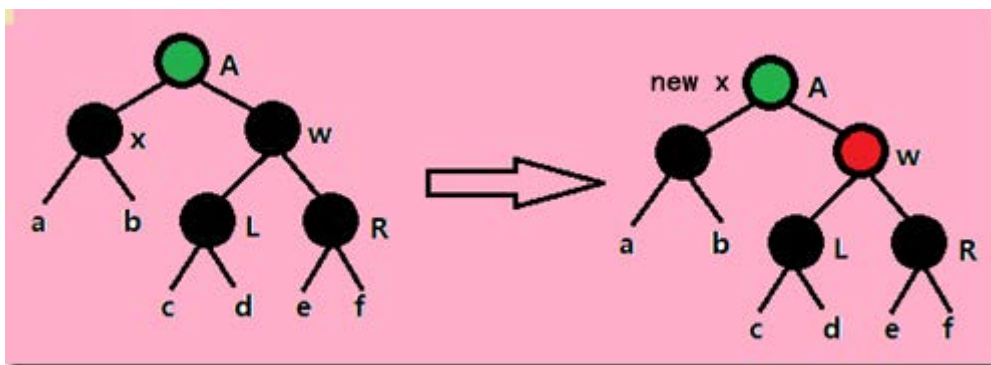
具体分为以下四种情况：（下面针对  $x$  是左儿子的情况讨论，右儿子对称）

**Case1:  $X$  的兄弟  $W$  是红色（想办法将其变为黑色）**

由于  $W$  是红色的，因此其儿子节点和父节点必为黑色，只要将  $W$  和其父节点的颜色对换，在对父节点进行一次左旋转，便将  $W$  的左子节点放到了  $X$  的兄弟节点上， $X$  的兄弟节点变成了黑色，且红黑性质不变。但还不算完，只是暂时将情况1转变成了下面的情况2或3或4。



**Case2:  $X$  的兄弟节点  $W$  是黑色的，而且  $W$  的两个子节点都是黑色的。此时可以将  $X$  的一重黑色和  $W$  的黑色同时去掉，而转加给他们的父节点上，这是  $X$  就指向它的父节点了，因此此时父节点具有双重颜色了。这一重黑色节点上移。**

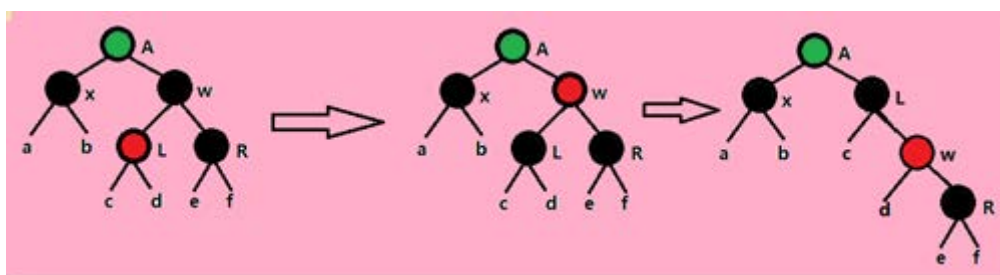


如果父节点原来是红色的，现在又加一层黑色，那么 X 现在指向的这个节点就是红黑两色的，直接把 X（也就是父节点）着为黑色。问题就已经完整解决了。

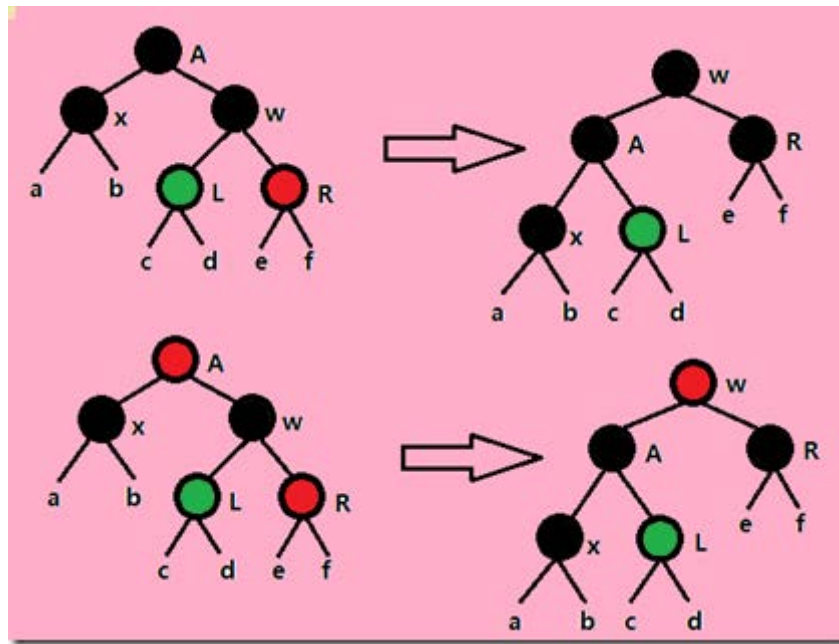
如果父节点现在是双层黑色，那就以父节点为新的 X 进行向上的下一轮的递归。

**Case3:** X 的兄弟节点 W 是黑色的，而且 W 的左子节点是红色的，右子节点是黑色的。此时通过交换 W 和其左子节点的颜色并进行一次向右旋转就可转换成下面的第四种情况。注意，原来 L 是红色的，所以 L 的子节点一定是黑色的，所以旋转中 L 节点的一个子树挂到之后着为红色的 W 节点上不会破坏红黑性质。

变形后黑色高度不变。



**Case4:** X 的兄弟节点 W 是黑色的，而且 W 的右子节点是红色的。这种情况下，做一次左旋，W 就处于根的位置，将 W 保持为原来的根的位置的颜色，同时将 W 的两个新的儿子节点的颜色变为黑色，去掉 X 的一重黑色。这样整个问题也就得到了解决。递归结束。（在代码上，为了标识递归结束，我们把 X 指向根节点）



因此，只要按上面四种情况一直递归处理下去，X 最终总会指向根结点或一个红色结点，这时我们就可以结束递归并把问题解决了。

以上就是红黑树的节点删除全过程。

#### 总结：

如果我们通过上面的情况画出所有的分支图，我们可以得出如下结论

**插入操作：解决的是 红-红 问题**

**删除操作：解决的是 黑-黑 问题**

即你可以从分支图中看出，需要往上遍历的情况为红红(插入)，或者为黑黑黑(删除)的情况，如果你认真分析并总结所有的情况后，并坚持下来，红黑树也就没有想象中的那么恐怖了，并且很美妙；

详细的红黑树删除节点的代码如下：

```
#include<iostream>
using namespace std;

// 定义节点颜色
enum COLOR
{
    BLACK = 0,
    RED
};

// 红黑树节点
typedef struct RB_Tree_Node
{
    int key;
    struct RB_Tree_Node *left;
    struct RB_Tree_Node *right;
```

```

    struct RB_Tree_Node *parent;
    unsigned char RB_COLOR;
}RB_Node;

// 红黑树，包含一个指向根节点的指针
typedef struct RBTree
{
    RB_Node* root;
}*RB_Tree;

// 红黑树的NIL节点
static RB_Tree_Node NIL = {0, 0, 0, 0, BLACK};

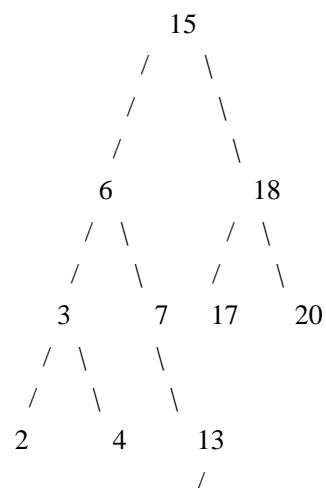
#define PNIL (&NIL)    // NIL 节点地址

void Init_RBTree(RB_Tree pTree) // 初始化一棵红黑树
{
    pTree->root = PNIL;
}

// 查找最小键值节点
RB_Node* RBTREE_MIN(RB_Node* pRoot)
{
    while (PNIL != pRoot->left)
    {
        pRoot = pRoot->left;
    }
    return pRoot;
}

/*

```



```

*/
// 查找指定节点的后继节点
RB_Node* RBTREE_SUCCESSOR(RB_Node* pRoot)
{
    if (PNIL != pRoot->right)    // 查找图中 6 的后继节点时就调用 RBTREE_MIN 函数
    {
        return RBTREE_MIN(pRoot->right);
    }
    // 节点没有右子树的时候，进入下面的 while 循环（如查找图中 13 的后继节点时，它的后继节点是 15）
    RB_Node* pParent = pRoot->parent;
    while((PNIL != pParent) && (pRoot == pParent->right))
    {
        pRoot = pParent;
        pParent = pParent->parent;
    }
    return pParent;
}

// 红黑树的节点删除
RB_Node* Delete(RB_Tree pTree, RB_Node* pDel)
{
    RB_Node* rel_delete_point;
    if(pDel->left == PNIL || pDel->right == PNIL)
        rel_delete_point = pDel;
    else
        rel_delete_point = RBTREE_SUCCESSOR(pDel);    // 查找后继节点

    RB_Node* delete_point_child;
    if(rel_delete_point->right != PNIL)
    {
        delete_point_child = rel_delete_point->right;
    }
    else if(rel_delete_point->left != PNIL)
    {
        delete_point_child = rel_delete_point->left;
    }
    else
    {
        delete_point_child = PNIL;
    }
    delete_point_child->parent = rel_delete_point->parent;
}

```



```

if(rel_delete_point->parent == PNIL)    // 删除的节点是根节点
{
    pTree->root = delete_point_child;
}
else if(rel_delete_point == rel_delete_point->parent->right)
{
    rel_delete_point->parent->right = delete_point_child;
}
else
{
    rel_delete_point->parent->left = delete_point_child;
}
if(pDel != rel_delete_point)
{
    pDel->key = rel_delete_point->key;
}
if(rel_delete_point->RB_COLOR == BLACK)
{
    DeleteFixUp(pTree , delete_point_child);
}
return rel_delete_point;
}

```

/\*

算法导论上的描述如下:

**RB-DELETE-FIXUP(T, x)**

```

1 while x ≠ root[T] and color[x] = BLACK
2     do if x = left[p[x]]
3         then w ← right[p[x]]
4             if color[w] = RED
5                 then color[w] ← BLACK                      Case 1
6                     color[p[x]] ← RED                      Case 1
7                     LEFT-ROTATE(T, p[x])                  Case 1
8                     w ← right[p[x]]                      Case 1
9             if color[left[w]] = BLACK and color[right[w]] = BLACK
10                then color[w] ← RED                      Case 2
11                x p[x]                                     Case 2
12            else if color[right[w]] = BLACK
13                then color[left[w]] ← BLACK              Case 3
14                    color[w] ← RED                      Case 3
15                    RIGHT-ROTATE(T, w)                  Case 3
16                    w ← right[p[x]]                    Case 3
17                color[w] ← color[p[x]]                  Case 4
18                color[p[x]] ← BLACK                     Case 4

```

```

19             color[right[w]] ← BLACK                                Case 4
20             LEFT-ROTATE(T, p[x])                                    Case 4
21             x ← root[T]                                            Case 4
22             else (same as then clause with "right" and "left" exchanged)
23 color[x] ← BLACK
*/
//接下来的工作，很简单，即把上述伪代码改写成 c++代码即可
void DeleteFixUp(RB_Tree pTree, RB_Node* node)
{
    while(node != pTree->root && node->RB_COLOR == BLACK)
    {
        if(node == node->parent->left)
        {
            RB_Node* brother = node->parent->right;
            if(brother->RB_COLOR==RED)    //情况 1: x 的兄弟 w 是红色的。
            {
                brother->RB_COLOR = BLACK;
                node->parent->RB_COLOR = RED;
                RotateLeft(node->parent);
            }
            else    //情况 2: x 的兄弟 w 是黑色的,
            {
                if(brother->left->RB_COLOR == BLACK && brother->right->RB_COLOR
== BLACK) //w 的两个孩子都是黑色的
                {
                    brother->RB_COLOR = RED;
                    node = node->parent;
                }
                else
                {
                    if(brother->right->RB_COLOR == BLACK)    //情况 3: x 的兄弟 w 是
黑色的, w 的右孩子是黑色 (w 的左孩子是红色)。
                    {
                        brother->RB_COLOR = RED;
                        brother->left->RB_COLOR = BLACK;
                        RotateRight(brother);
                        brother = node->parent->right;    //情况 3 转换为情况 4
                    }
                    //情况 4: x 的兄弟 w 是黑色的, 且 w 的右孩子时红色的
                    brother->RB_COLOR = node->parent->RB_COLOR;
                    node->parent->RB_COLOR = BLACK;
                    brother->right->RB_COLOR = BLACK;
                    RotateLeft(node->parent);
                    node = pTree->root;
                }
            }
        }
    }
}

```



```

Const __rb_tree_color_type  __rb_tree_red= false;
Const __rb_tree_color_type  __rb_tree_black=true;

Struct __rb_tree_node_base//实际上__rb_tree_node_base 就相当于 list_head
{
    Typedef __rb_tree_color_type color_type; //类型定义
    Typedef __rb_tree_node_base *base_ptr; //指针类型
    Color_type color;
    Base_ptr  parent; //节点保存父亲（因为 rb-tree 的许多操作必须知道父节点）
    Base_ptr  left;//左孩子
    Base_ptr  right;//右孩子

    Static base_ptr minimum ( base_ptr x)
    {
        While ( x->left != 0)
            X=x->left;
        Return x;
    }
    Static base_ptr maximum(base_ptr x){
        While( x->right != 0)
            X = x->right;
        Return x;
    }
};

Template<class value>
Struct __rb_tree_node: public __rb_tree_node_base
{
    Typedef __rb_tree_node<Value>* link_type;
    Value value_field; // 节点值
    //其他基本的节点信息，如颜色，左右子树，父亲节点等都已经基类中定义
};

```

#### 5.2.4 RB-tree 的迭代器

为了更大的弹性，SGI 将 Rb-tree 迭代器设计为两层，图 5-16 展示了双层节点结构和双层迭代器结构之间的关系：\_\_rb\_tree\_node 继承自 \_\_rb\_tree\_node\_base, \_\_rb\_tree\_iterator 继承自 \_\_rb\_tree\_base\_iterator.

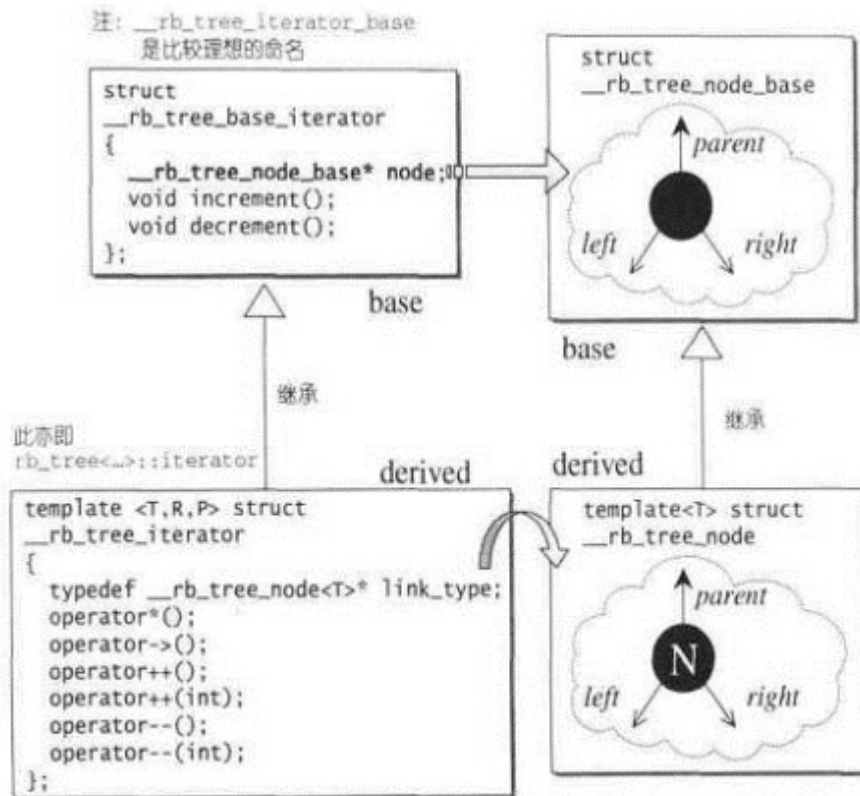


图 5-16 RB-tree 的节点和迭代器之间的关系。

这种双层架构和 4.9 节的 `slist` 极相似，请参考图 4-25。

图 5-16 RB-tree 的节点和迭代器之间的关系

Rb-tree 迭代器属于双向迭代器，但是不具备随机定位能力，rb-tree 迭代器的 `operator++()` 调用了基层迭代器的 `increment()`，rb-tree 迭代器的 `operator--()` 调用了基层迭代器的 `decrement()`。前进或者后退的行为完全依据二叉搜索树的节点排列法则。

Struct `__rb_tree_base_iterator`

```

{
    Typedef __rb_tree_node_base::base_ptr    base_ptr;
    Typedef bidirectional_iterator_tag    iterator_category;
    Typedef ptrdiff_t difference_type;
    Base_ptr    node; //实际节点指针，和容器产生一个连结关系

```

Void `increment()` //在二叉搜索树中找到正确位置

```

{
    If ( node->right != 0 ) //如果有右子节点，查找右子树的最左节点
    {
        Node = node->right;
        While ( node->left !=0)
            Node = node -> left;
    }
}

```

```

Else    //如果没有右子节点
{
    Base_ptr y = node -> parent;    //找出父亲节点
    //如果当前节点在父节点的右子树，那就要上溯，因为父节点的父节点的子树
    种存在大于当前节点的节点，直到不为右子节点
    While( y->right == node)    //一直上溯，直到不为右节点为止
    {
        Node = y;
        Y = y->parent;
    }

    If( node->right != y) //若此时的右子结点不等于此时的父节点，这是为了应付一
    种特殊情况：我们欲寻找根节点的下一节点，而恰巧根节点无右子结点，当然这里
    特殊做法必须配合 rb-tree 根节点与特殊之 header 之间的特殊关系 【状况 4】
        Node = y;
    }
}

Void decrement()
{
    //如果是红节点，且父节点的父节点等于自己，这种情况发生于 node 为 header 时（亦
    即 node 为 end()时），header 之右子结点指向整棵树的 max 节点 【状况 1】
    If( node->color == __rb_tree_red && node->parent->parent == node)
        Node = node-> right;
    Else if ( node->left !=0) //如果有左子节点
    {
        Base_ptr y = node->left;
        While( y->right != 0)
            y = y->right ;
        node = y ;
    }
    Else //既非根节点，也无左节点
    {
        Base_ptr y = node->parent;
        While( node== y->left) //当现行节点身为左子节点时
        {
            Node = y ;
            Y = y->parent ;
        }
        Node =y ;    //直到它是父节点的右子节点
    }
}

```

RB-tree 的真正迭代器

```

Template<class Value, class Ref, class Ptr>
Struct __rb_tree_iterator : public __rb_tree_base_iterator
{
    Typedef Value value_type;
    Typedef Ref reference;
    Typedef Ptr pointer;

    Typedef __rb_tree_iterator<Value, Value&, Value*> iterator;
    Typedef __rb_tree_iterator<Value, const Value&, const Value*> const_iterator;
    Typedef __rb_tree_iterator<Value, Ref, Ptr> self;
    Typedef __rb_tree_node<Value>* link_type; //底层指针类型,指向 RB-tree 的节点
    __rb_tree_node

    __rb_tree_iterator(){}
    __rb_tree_iterator( link_type x) { node=x;}
    __rb_tree_iterator( const iterator& it) { node = it.node;}

    Reference operator*() const {return link_type(node) -> value_field;}
    Pointer operator->() const{return &(operator*());}

    Self& operator++() //再做进一步的封装，调用的是基类的通用函数 increment。
    {
        Increment();
        Return *this;
    }

    Self operator++(int)
    {
        Self tmp = *this;
        Increment();
        Return tmp;
    }

    Self& operator--()
    {
        Decrement();
        Return *this;
    }
    Self operator--(int)
    {
        Self tmp = *this;
        Decrement();
        Return tmp;
    }
}

```

```
};
```

在 `__rb_tree_iterator_base` 的 `increment()` 和 `decrement()` 两函数中，比较令人费解的是 `increment` 中的状况 4 和 `decrement` 中的状况（见源代码注释部分），他们分别发生于图 5-17 所展示的状态下。

当迭代器指向根节点而后者无右子节点时，若对迭代器进行 ++ 操作，会进入 `__rb_tree_base_iterator::increment()` 的状况 (2),(4)。

当迭代器为 `end()` 时，若对它进行操作，会进入 `__rb_tree_base_iterator::decrement()` 的状况 (1)。

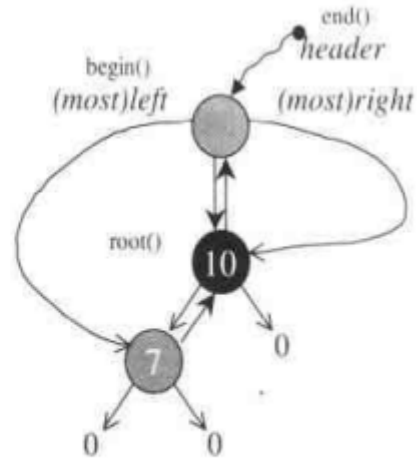


图 5-17 `increment()` 和 `decrement()` 两函数中较令人费解的状况 4 和状况 1，其中的 `header` 是实现上的特殊技巧，见稍后说明。

### 5.2.5 rb-tree 的数据结构

```
Template<class Key, class Value, class KeyOfValue, class Compare, class Alloc=alloc>
```

```
Class rb_tree
```

```
{
```

```
Protected:
```

```
    Typedef void* void_pointer;
```

```
    Typedef __rb_tree_node_base* base_ptr;
```

```
    Typedef __rb_tree_node<value> rb_tree_node;
```

```
    Typedef simple_alloc<rb_tree_node,Alloc>    rb_tree_node_allocator;
```

```
    Typedef __rb_tree_color_type color_type;
```

```
Public:
```

```
    Typedef Key key_type;
```

```
    Typedef Value value_type;
```

```
    Typedef Value_type* pointer;
```

```
    Typedef const value_type* const_pointer;
```

```
    Typedef value_type& reference;
```

```
    Typedef const value_type& const_reference;
```

```
    Typedef rb_tree_node* link_type;
```

```
    Typedef size_t size_type;
```

```
    Typedef ptrdiff_t difference_type;
```

```
Protected:
```



```
Link_type get_node() { return rb_tree_node_allocator::allocate();} //分配一个节点
Void put_node(link_type p){rb_tree_node_allocator::deallocate(p);} //释放一个节点
```

```
Link_type create_node(const value_type& x) //创建一个值为 x 的节点
```

```
{
//底层的创建节点，并没有涉及到树！
    Link_type tmp = get_node(); //配置空间
    __STL_TRY
    {
        Construct(&tmp->value_field,x);
        //构造内容，将 x 赋值在 value_field
    }
    __STL_UNWIND( put_node(tmp)); //否则销毁节点！
    Return tmp;
}
```

```
//底层的复制，并没有涉及到树！
```

```
Link_type clone_node(link_type x) //只能复制【值和色】，不能赋值其左右指针
```

```
{
    Link_type tmp = create_node ( x->value_field );
    Tmp->color = x->color;
    Tmp->left = 0;
    Tmp->right = 0;
    Return tmp;
}
```

```
Void destroy_node( link_type p) //底层的销毁，并没有涉及到树！
```

```
{
    Destroy(&p->value_field);
    Put_node(p);
}
```

Protected:

```
Size_type node_count;
```

```
Link_type header;
```

```
Compare key_compare; //节点的键值大小比较函数
```

```
//以下三个函数用来方便获取 header 的成员
```

```
Link_type& root() const {return (link_type&) header->parent;}
```

```
Link_type& leftmost() const { return (link_type&) header->left;}
```

```
Link_type& rightmost() const { return ( link_type&) header->right;}
```

```
//以下六个函数用来方便取得节点 x 的成员
```

```
Static link_type& left(link_type x)
```

```

{ return (link_type&) (x->left); }
Static link_type& right(link_type x)
{ return (link_type&)(x->right);}
Static link_type& parent(link_type x)
{ return (link_type&) (x->parent);}
Static reference value(link_type x)
{ return x->value_field;}
Static const Key& key(link_type x)
{ return KeyOfValue()(value(x));}
Static color_type& color(link_type x)
{ return (color_type&)(x->color);}

```

//以下六个函数用来方便去的节点 x 的成员

```

Static link_type & left(base_ptr x)
{ return (link_type&)(x->left);}
Static link_type & right(base_ptr x)
{ return (link_type&)(x->right);}
Static link_type & parent(base_ptr x)
{ return (link_type&)(x->parent);}
Static reference value(base_ptr x)
{return ((link_type)x)->value_filed;}
Static const Key& key(base_ptr x)
{return KeyOfValue()(value(link_type(x)));}
Static color_type& color(base_ptr x)
{return (color_type&)(link_type(x)->color);}

```

//求取极大值和极小值

```

Static link_type minimum(link_type x)
{
    Return (link_type) __rb_tree_node_base::minimum(x);
}
Static link_type maxnum( link_type x)
{
    Return (link_type) __rb_tree_node_base::maximum(x);
}

```

Public:

```

Typedef __rb_tree_iterator<value_type, reference, pointer> iterator;

```

Private:

```

Iterator __insert(base_ptr x, base_ptr y, const value_type& v);
Link_type __copy ( link_type x , link_type p);
Void __erase( link_type x);
Void init()

```

```

{
    Header=get_node();//产生一个节点，令 header 指向它
    Color(header) = __rb_tree_red;//令 header 为红色，用来区分 header 和 root
    Root()=0;
    Leftmost()=header;
    Rightmost()=header;
}

```

Public:

```

Rb_tree ( const Compare& comp= Compare() )
    : node_count(0),key_compare(comp) { init(); }
~rb_tree() { clear(); put_node(header);}

```

Public:

```

Compare key_comp() const {return key_compare;}
Iterator begin() { return leftmost();}
Iterator end() {return header;}
Bool empty() const { return node_count==0;}
Size_type size() const {return node_count;}
Size_type max_size() const {return size_type(-1);}

```

Public:

```

//将 x 插入到 rb-tree 中（保持节点值独一无二）
Pair<iterator,bool> insert_unique(const value_type& x);
//将 x 插入到 rb-tree 中（允许节点值重复）
Iterator insert_equal( const value_type& x);
};

```

### 5.2.6 rb-tree 的构造与内存管理

Rb-tree 所定义的专属空间分配器 rb\_tree\_node\_allocator，每次可分配一个节点：

Template<class key, class value, class KeyOfValue, class compare, class Alloc alloc>

Class rb\_tree {

Protected:

Typedef \_\_rb\_tree\_node<value> rb\_tree\_node;

Typedef simple\_alloc<rb\_tree\_node, alloc> rb\_tree\_node\_allocator;

.....

}

Rb\_tree 的构造方式有两种：一是以现有的 rb\_tree 复制一个新的 rb\_tree，另一种是产生一颗空树。

```

rb_tree(const Compare& comp = Compare())
    :node_count(0), key_compare(comp) {init();}

```

其中的 init()是实现技巧的关键：

Private:

```

Void init()

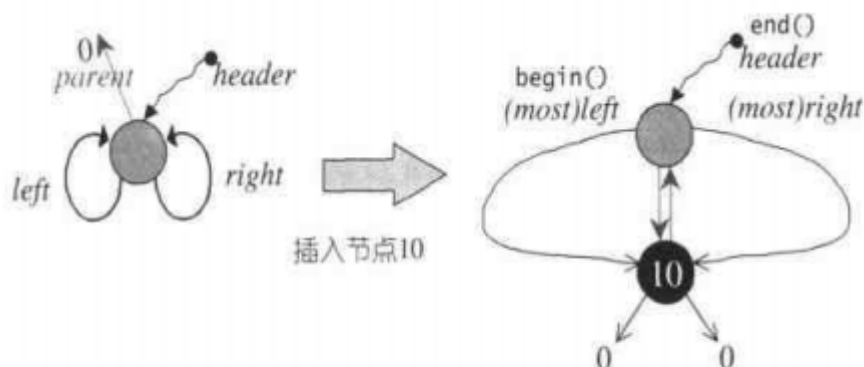
```

```

{
    Header=get_node();//产生一个节点，令 header 指向它
    Color(header) = __rb_tree_red;//令 header 为红色，用来区分 header 和 root
    Root();
    Leftmost=header;//令 header 的左子结点为自己
    Rightmost=header;//令 header 的右子结点为自己
}

```

我们知道树状结构的操作，最需注意的就是边界情况的发生，当走到根节点时要有特殊的处理。为简化处理，SGI STL 特别为根节点再设计一个父节点 header，其初始状态如图 5-18 所示。



注意，header和root互为对方的父节点，这是一种实现技巧

图 5-18 图左是 rb-tree 的初始状态，图右为加入第一个节点后的状态

当插入新值时，不但要依照 rb-tree 规则来调整，还要维护 header 的正确性，使其父节点指向根节点，左子结点指向最小节点，右子结点指向最大节点。

### 5.2.7 rb-tree 的元素操作

Rb-tree 提供两种插入操作：insert\_unique()和 insert\_equal(),前者表示被插入节点的键值在整棵树中必须独一无二，否则插入不会进行，后者表示被插入节点的键值在整棵树中可以重复。

#### 元素插入操作

//插入新值：节点键值允许重复

```

Template<class Key, class Value, class KeyOfValue, class Compare, class Alloc>
Typename rb_tree< Key, Value, KeyOfValue, Compare, Alloc>::iterator
Rb_tree<Key , Value, KeyOfValue, Compare,Alloc>::insert_equal( const Value& v)
{
    Link_type y= header;//y 用于记录 x 的 parent
    Link_type x= root();
    While(x!=0)                //寻找适当的插入点
    {
        Y = x;
        x= key_compare ( KeyOfValue()(v,key(x))? Left(x):right(x);
        //利用二叉搜索树的规则，找到合适的插入点
    }
}

```

```

    }
    Return __insert(x,y,v); // x 是新值插入点， y 是该空点的父亲， v 是值
}

```

```

Template<class Key, class Value, class KeyOfValue, class Compare, class Alloc>
Pair<typename rb_tree<Key,Value,KeyOfValue,Compare,Alloc>::iterator, bool>
Rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::Insert_unique(const Value& v)
{
    Link_type y = header;
    Link_type x = root();
    Bool comp= true;
    While(x!=0)
    {
        Y=x;
        Comp = key_compare (KeyOfValue()(v), key(x));
        X = comp ? left(x) : right(x);
    }
    Iterator j = iterator(y);    // 迭代器指向插入点的父节点 y
    If(comp)    // 如果离开 while 时 comp 为真，即遇到比 v 大的，所以要插入左侧
        If(j==begin()) //如果插入节点的父节点是最左节点
            Return pair<iterator,bool>(__insert(x,y,v),true);
        Else
            --j; //否则往前（左）偏一位（查看前面的是否会出现相等的情况）
    //KeyOfValue()用于根据给定的 value 获取 key
    If ( key_compare ( key(j.node), KeyOfValue()(v) )) //比较如果小于 v，那么就插入右侧
        Return pair<iterator,bool>(__insert(x,y,v),true);
    Return pair< iterator, bool> (j,false); //相等！
}

```

**真正在 RB 树结构上完成插入的函数 \_\_insert()**

```

Template<class Key, class Value , class KeyOfValue, class Compare, class Alloc>
Typename rb_tree<Key,Value,KeyOfValue,Compare,Alloc>::iterator
Rb_tree<Key,Value,KeyOfValue,Compare,Alloc>::__insert(base_ptr x_, base_ptr y_, const
Value& v)//x_是新值插入点， y_是插入点的父节点， v 是新值
{
    Link_type x = (link_type) x_;
    Link_type y = (link_type) y_;
    Link_type z;
    If (y == header || key_compare( KeyOfValue()(v) ,key(y) )) // y 等于 header 时即树为空
    {
        Z = create_node(v);
        Left(y) = z;    //y 的左子树指向 z
        If(y==header)    //考虑得十分周到，还有一开始树为空的情况，极端情况
        {

```

```

        Root() = z; //同时，也就是说树一开始为空，初始化时 root, right, left 都指向 z
        Rightmost() = z;
    }
    Else if ( y == leftmost() ) //如果父节点 y 是最左，那 z 更左
        Leftmost() = z;
}
Else //y 的值小于 v
{
    Z = create_node(v);
    Right(y) = z;
    If ( y == rightmost() ) //考虑得十分周到，还有 rightmost 这个边界情况
        Rightmost() = z;
}
Parent(z) = y;
Left(z) = 0;
Right(z) = 0;
__rb_tree_rebalance ( z ,header->parent); //重新编排红黑树！
++node_count;
Return iterator(z);
}

```

### 调整 rb-tree（旋转及改变颜色）

//插入节点后，可能不满足红黑树的特性，所以需要进行 rebalance

Inline void

```

__rb_tree_rebalance( __rb_tree_node_base* x, __rb_tree_node_base* &root)
{
    x->color= __rb_tree_red; //新插入的节点必然为红色，之后可以再调整颜色
    while( x!= root && x->parent->color == __rb_tree_red) //父节点为红色，发生了冲突
    {
        If( x->parent == x->parent->parent->left)//父节点是祖先节点的左节点
        {
            __rb_tree_node_base* y= x->parent->parent->right;//取得伯父节点
            if( y && y->color==__rb_tree_red)
                //不需要旋转，直接调整颜色，但是要调整到祖父的颜色，所以要再检查祖父
                的颜色是否达到要求，
            {
                x->parent->color = __rb_tree_black;
                y->color = __rb_tree_black;
                x->parent->parent->color= __rb_tree_red;
                x = x->parent->parent; //让祖父节点再被考察一次
            }
        }
        Else // 没有伯父节点或者伯父节点为黑
        {
            If ( x == x->parent->right)//如果新节点为父节点的右子节点，则以父节点

```

```

{
    X = x->parent; //获取父节点
    __rb_tree_rotate_left(x, root); //进行左旋
}
x->parent->color = __rb_tree_black;
x->parent->parent->color = __rb_tree_red;
__rb_tree_rotate_right(x->parent->parent, root); //以祖父节点为轴右旋
}
}
Else //父节点是右子节点
{
    __rb_tree_node_base* y = x->parent->parent->left; //取得伯父节点 y
    If( y && y->color == rb_tree_red)
    {
        //简单修改颜色就可以
        x->parent->color = __rb_tree_black;
        y->color = __rb_tree_black;
        y->color = __rb_tree_black;
        x->parent->parent->color = __rb_tree_red;
        x = x->parent->parent;
    }
    Else
    {
        If( x == x->parent->left) //如果新节点是父节点的左孩子，则以父节点为轴
            右旋
        {
            X = x->parent;
            __rb_tree_rotate_right(x, root);
        }
        x->parent->color = __rb_tree_black;
        x->parent->parent->color = __rb_tree_red;
        __rb_tree_rotate_left(x->parent->parent, root);
    }
}
} //while 结束
Root->color = __rb_tree_black;
}

```

```
// 左旋函数
```

```
inline void _rb_tree_rotate_left(_rb_tree_node_base* x, _rb_tree_node_base*& root)
{
    // x 为旋转点
```

```

    _rb_tree_node_base* y = x->right;          // 令 y 为旋转点的右子节点
    x->right = y->left;
    if(y->left != 0)
        y->left->parent = x;                    // 别忘了回马枪设定父节点
    y->parent = x->parent;

    // 令 y 完全顶替 x 的地位（必须将 x 对其父节点的关系完全接收过来）
    if(x == root)    // x 为根节点
        root = y;
    else if(x == x->parent->left)                // x 为其父节点的左子节点
        x->parent->left = y;
    else                                            // x 为其父节点的右子节点
        x->parent->right = y;
    y->left = x;
    x->parent = y;
}

// 右旋函数
inline void _rb_tree_rotate_right(_rb_tree_node_base* x , _rb_tree_node_base*& root)
{
    // x 为旋转点
    _rb_tree_node_base* y = x->left;          // 令 y 为旋转点的左子节点
    x->left = y->right;
    if(y->right != 0)
        y->right->parent = x;                  // 别忘了回马枪设定父节点
    y->parent = x->parent;

    // 令 y 完全顶替 x 的地位（必须将 x 对其父节点的关系完全接收过来）
    if(x == root)
        root = y;
    else if(x == x->parent->right)                // x 为其父节点的右子节点
        x->parent->right = y;
    else                                            // x 为其父节点的左子节点
        x->parent->left = y;
    y->right = x;
    x->parent = y;
}

```

## 元素查找

```

Template<class Key, class Value, class KeyOfValue, class Compare, class Alloc>
typename rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::iterator
Rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::find(const Key& k)

```



```

{
    Link_type y = header; //y 用于保存最后一个不小于 k 的节点
    Link_type x= root();
    While(x!=0)
    {
        If (!key_compare(key(x),k) ) // 将 root 的 key 和 k 进行比较
            Y = x, x=left(x); //如果大于等于，则向左走
        Else
            X=right(x);
    }
    Iterator j = iterator(y); //找到等于或者最靠近的大于
    Return ( j==end() || key_compare( k, key(j.node)) ? end(): j ;
}

```

### 5.3 set

集合 set 的特征：所有元素都会根据元素的值自动被排序，而且不像 map 一样拥有 value 和 key，set 的元素 key 键值就是实值 value。Set 不允许两个元素有相同的键值。

我们不可以随便的通过 set 的迭代器来改变 set 的元素值，因为这样会破坏了 set 的组织，所以我们的 set 迭代器是一个 constant iterator。

Set 与 list 有些相同的性质：当用户对它进行元素插入和删除操作时，操作之前的所有迭代器，在操作之后都依然有效。当然被删除的那个元素的迭代器例外。

Set 提供了一组 set/multiset 相关算法：交集 set\_intersection、并集 set\_union、差集 set\_difference、对称差集 set\_symmetric\_difference 等。

标准的 STL 中的 set 底层使用的是 RB-tree。

Set 源码摘录：略。。。

### 5.4 map

Map 中所有元素都是以 pair 的形式出现，<key,value>,然后所有元素都会根据 key 来排序，当然，key 不可以有相同的，而 value 可以有相同的。

Template<class T1, class T2> //classT1 for key, ClassT2 for value

Struct pair

```

{
    Typedef T1 first_type;
    Typedef T2 second_type;
    T1 first;
    T2 second;
    Pair() : first(T1()), second( T2()) {}
    Pair( const T1& a , const T2& b) : first(a), second(b){ }
};

```

和 set 一样的在于不可以修改 map 的 key，因为会改变到整个 map 的排列，而不一样的是，map 可以改变元素的 value，所以这样一来 map 的迭代器既不是一个 constance iterator

(key 和 value 都不可以修改)，也不是一个 mutable iteration (key 和 value 都可以修改)

Map 拥有和 list 相同的某些性质：当用户对它进行元素插入和删除操作时，操作之前的所有迭代器，在操作之后都依然有效。当然被删除的那个元素的迭代器例外。

标准 STL map 以 RB-tree 为底层机制。

图 5-20 展示 map 架构。

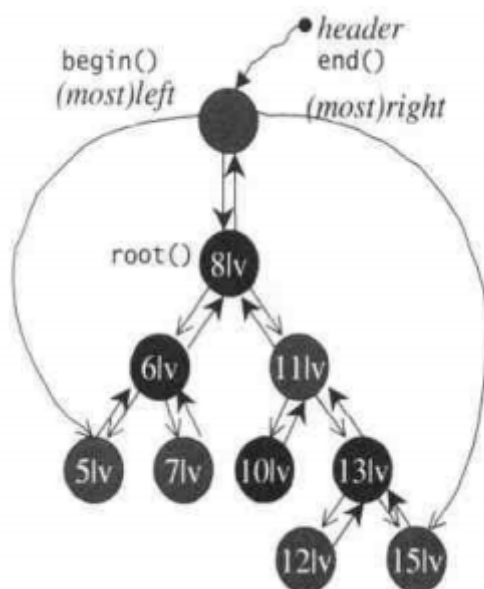


图 5-20 SGI STL map 以红黑树为底层机制，每个节点的内容是一个 pair

Pair 的第一个元素被视为 key，第二个元素被视为 value

Map 源码摘录：略。。。

## 5.5 multiset

Multiset 的特性及用法和 set 一致，唯一的差别在于它允许键值重复，所以插入操作调用的是底层机制 rb-tree 的 `insert_equal` 而不是 `inset_unique`。

## 5.6 multimap

Multimap 的特性及用法和 map 一致，唯一的差别在于它允许键值重复，所以插入操作调用的是底层机制 rb-tree 的 `insert_equal` 而不是 `inset_unique`。

## 5.7 hashtable

二叉搜索树再插入，删除，查找具有对数的平均时间，但是它是基于这样的假设，插入的数据具有一定的随机性，不然的话，如果是有序元素的插入，会使得二叉搜索树严重不平衡。而 hashtable 这种数据结构，在插入，删除，查找也有常数平均时间，而不依赖于插入元素的随机性，是以统计为基础的。

解决 hash 冲突的方法包括：线性探测、二次探测、开放寻址等。

线性探测：当 hash function 计算出某个元素的插入位置，而该位置上的空间已不再可用

时，最简单的方法就是循环往下寻找（如果到达尾端，就回到头部继续寻找），直到找到一个可用空间为止。当进行元素查找时，如果 hash function 计算出来的位置上的元素值与我们的搜寻目标不符，就循环往下寻找，直到找到吻合者，或遇到空格元素。对于元素的删除，则使用**惰性删除 (lazy deletion)**，也就是只标记删除几号，实际删除操作等到表格重新整理时在进行--- 这是因为 hash table 中的每一个元素不仅描述它自己，也关系到其它元素的排列。图 5-22 所示是线性探测的一个实例。

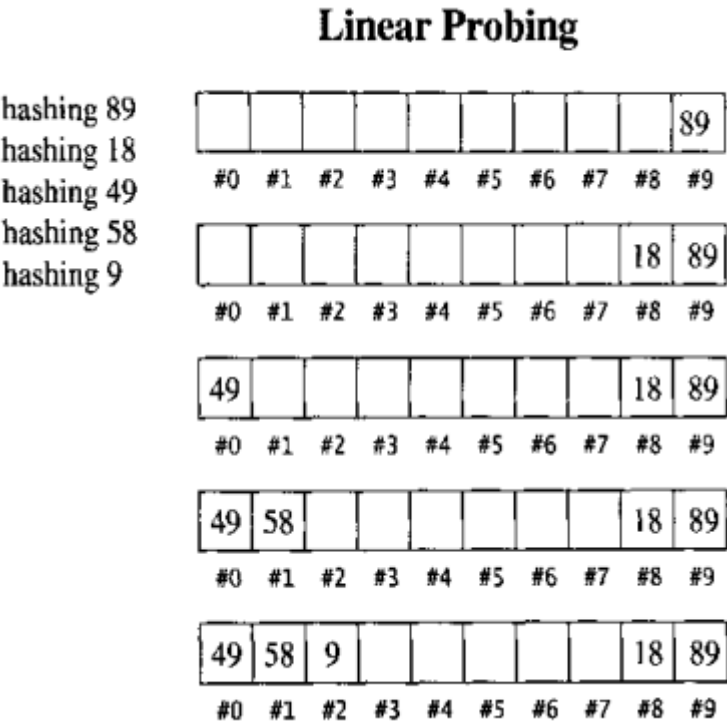


图 5-22 线性探测，依序插入 5 个元素，array 的变化

二次探测：如果 hash function 计算出新元素的位置为  $h$ ，而该位置实际已被使用，那么我们就依序尝试  $h+1^2, h+2^2, h+3^2, \dots, h+i^2$ ，而不是像线性探测那样依序尝试  $h+1, h+2, h+3, h+4, \dots, h+i$ 。图 5-23 是二次探测的一个实例。

## Quadratic Probing

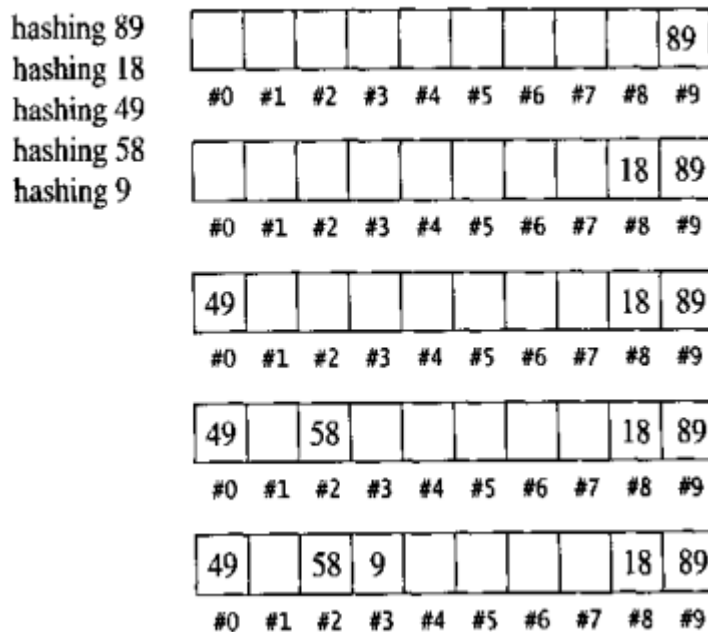


图 5-23 二次探测 依序插入 5 个元素，array 的变化

开放寻址/开链：在每一个表格元素中维护一个 list: hash function 为我们分配某一个 list，然后我们在那个 list 上执行元素插入、查找、删除等。

SGI STL 的 hash table 就是采用开链方法。

### 5.7.2 hashtable buckets/nodes

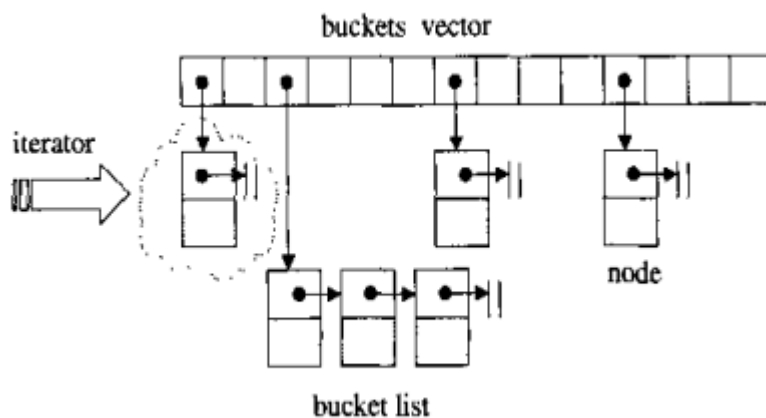


图 5-24 以开链方法完成的 hash table

Bucket 维护的 linked list，采用的是自行维护的 hash table node，buckets 集合则以 vector 完成，以便有动态扩充能力。

```

Template<class Value>
Struct __hashtable_node
{
    __hashtable_node* next ;

```

```

    Value val ;
};

```

### 5.7.3 hashtable 的迭代器

```

Template<class Value, class Key, class HashFun, class ExtractKey, class EqualKey, class Alloc>
Struct __hashtable_iterator
{
    Typedef hashtable<value,key,hashfun,extractKey,equalKey,alloc> hashtable;
    Typedef __hashtable_iterator<value,key,hashFun,extractKey,equalKey,alloc> iterator;
    Typedef __hashtable_const_iterator<value, key,hashFun,extractkEY, EqualKey,Alloc>
    const_iterator;
    Typedef __hashtable_node<value> node;

    Typedef forward_iterator_tag iterator_category;
    Typedef Value value_type;
    Typedef ptrdiff_t difference_type;
    Typedef size_t size_type;
    Typedef Value& reference;
    Typedef Value* pointer;

    //真正的 hashtable 的迭代器使用的东西
    Node* curr; //迭代器当前所指节点
    Hashtable* ht; //跳跃到其他桶时必须知道原来的 hashtable 的情况

    __hashtable_iterator(node* n, hashtable* tab): cur(n), ht(tab){}
    __hashtable_iterator(){}
    Reference operator*()const{return cur->val;}
    Pointer operator->()const{return &(operator*());}

    Iterator& operator++(); //指向同一个桶的下一个元素，如果没有则指向下一个桶的元素
    {
        Const node* old= cur;
        Cur = cur->next; //将迭代器的 node 指针指向下一个即可
        If ( !cur ) //如果没有下一个，那就要跳桶
        {
            //根据元素值，定位出下一个 bucket
            Size_type bucket = ht->bkt_num(old->val);
            While (!cur && ++bucket < ht->bucket.size())//直到找到有 node 的 bucket
                Cur= ht->buckets[bucket];
        }
        Return *this;
    }

    Iterator operator++(int); //后置自增

```

```

{
    Iterator tmp= *this;
    ++*this;//调用 operator++()
    Return tmp;
}

Bool operator==(const itetator& it) const{return cur== it.cur;}
Bool operator!=(const itetator& it) const{return cur!= it.cur;}
};

```

hashtable 迭代器前进操作是首先尝试从目前所指节点出发，前进一个节点，如果目前节点正好是 list 的尾端，就跳至下一个 bucket 上。

注意，这里涉及的 hashtable 的迭代器没有后退操作(operator--()),hashtable 也没有定义所谓的逆向迭代器(reverse iterator).

#### 5.7.4 hashtable 的数据结构

```
Template<class value, class Key, class HashFun, class ExtractKey, class EqualKey, class Alloc>
```

```
Class hashtable
```

```

{
Public:
    Typedef HashFun hasher;
    Typedef EqualKey key_equal;
    Typedef size_t size_type;

Private:
    Hasher hash;
    Key_equal equals;
    ExtractKey get_key;
    Typedef __hashtable_node<value> node;
    Typedef simple_alloc<node,Alloc> node_allocator;
    Vector<node*, Alloc> buckets; //以 vector 存放 buckets
    Size_type num_elements;

Public:
    Size_type bucket_count() const{return buckets.size();}
    .....
};

```

Hashtable 的模板参数包括：

Value: 节点的实值型别

Key: 节点的键值型别

HashFcn: hash function

ExtractKey: 从节点中取出键值

EqualKey: 判断键值是否相等

Alloc: 空间分配器

SGI STL 以质数涉及表格大小，并且先将 28 个质数（逐渐呈现大约两倍关系）计算好，以备随时使用，同时提供一个函数，用来查询这 28 个质数中，最接近某数并大于某数的质数：

```
Static const int __stl_num_primes=28;
Static const unsigned long __stl_prime_list[ __stl_num_primes] =
{
    53, 97,193,  389,  769,
    ....
}

Inline unsigned long __stl_next_prime(unsigned long n)
{
    Const unsigned long* first = __stl_prime_list;
    Const unsigned long* last = __stl_prime_list+ __stl_num_primes;
    Const unsigned long* pos= lower_bound(first,last,n);
    Return pos== last? *(last-1):*pos;
}
//最多有多少个 buckets
Size_type max_bucket_count() const
{return __stl_prime_list[__stl_num_primes - 1];}
```

#### 5.7.5 hashtable 的构造与内存管理

这部分内容比较熟悉，所以在此略过。。。

#### 5.8 hash\_set

STL set 多以 RB-tree 为底层机制，SGI 在 STL 标准之外提供了 hash\_set，以 hashtable 为底层机制。

Set 和 hash\_set 都能够快速搜寻元素，但是 RB-tree 有自动排序功能而 hashtable 没有，所以 set 的元素有自动排序功能而 hash\_set 没有。

#### 5.9 hash\_map

SGI 在 STL 标准之外提供了 hash\_map，以 hashtable 为底层机制。

map 和 hash\_map 都能够根据键值快速搜寻元素，但是 RB-tree 有自动排序功能而 hashtable 没有，所以 map 的元素有自动排序功能而 hash\_map 没有。

#### 5.10 hash\_multiset

Hash\_multiset 与 multiset 唯一的差别在于底层机制是 hashtable，因此 hash\_multiset 元素不会被自动排序。

Hash\_multiset 和 hash\_set 实现上的唯一差别是：前者元素插入操作采用 hashtable 的 insert\_equal()，而后者采用 inset\_unique()。

### 5.11 hash\_multimap

Hash\_multimap 与 multimap 唯一的差别在于底层机制是 hashtable，因此 hash\_multimap 元素不会被自动排序。

Hash\_multimap 和 hash\_map 实现上的唯一差别是：前者元素插入操作采用 hashtable 的 insert\_equal()，而后者采用 inset\_unique()。

### 5.12 STL 容器特征总结（含迭代器失效）

#### Vector

- 1、内部数据结构：连续存储，例如数组。
- 2、随机访问每个元素，所需要的时间为常量。
- 3、在末尾增加或删除元素所需时间与元素数目无关，在中间或开头增加或删除元素所需时间随元素数目呈线性变化。
- 4、可动态增加或减少元素，内存管理自动完成，但程序员可以使用 reserve() 成员函数来管理内存。
- 5、迭代器失效

插入：vector 的迭代器在内存重新分配时将失效（它所指向的元素在该操作的前后不再相同）。当把超过 capacity()-size() 个元素插入 vector 中时，内存会重新分配，所有的迭代器都将失效；

删除：当进行删除操作（erase, pop\_back）后，指向删除点及其后元素的迭代器全部失效。建议：使用 vector 时，用 reserve() 成员函数预先分配需要的内存空间，它既可以保护迭代器使之不会失效，又可以提高运行效率。

#### deque

- 1、内部数据结构：连续存储或者分段连续存储，具体依赖于实现，例如数组。
- 2、随机访问每个元素，所需要的时间为常量。
- 3、在开头和末尾增加元素所需时间与元素数目无关，在中间增加或删除元素所需时间随元素数目呈线性变化。
- 4、可动态增加或减少元素，内存管理自动完成，不提供用于内存管理的成员函数。
- 5、迭代器失效

插入：增加任何元素都将使 deque 的迭代器失效。

删除：在 deque 的中间删除元素将使迭代器失效。在 deque 的头或尾删除元素时，只有指向该元素的迭代器失效。

#### list

- 1、内部数据结构：双向环状链表。
- 2、不能随机访问一个元素。
- 3、可双向遍历。
- 4、在开头、末尾和中间任何地方增加或删除元素所需时间都为常量。
- 5、可动态增加或减少元素，内存管理自动完成。
- 6、迭代器失效

插入：增加任何元素都不会使迭代器失效。

删除：删除元素时，除了指向当前被删除元素的迭代器外，其它迭代器都不会失效。



### slist

- 1、内部数据结构：单向链表。
- 2、不可双向遍历，只能从前到后地遍历。
- 3、其它的特性同 list 相似。

建议：尽量不要使用 slist 的 insert, erase, previous 等操作。因为这些操作需要向前遍历，但是 slist 不能直接向前遍历，所以它会从头开始向后搜索，所需时间与位于当前元素之前的元素个数成正比。slist 专门提供了 insert\_after, erase\_after 等函数进行优化。但若经常需要向前遍历，建议选用 list。

### Stack

- 1、适配器，它可以将任意类型的序列容器转换为一个堆栈，一般使用 deque 作为支持的序列容器。
- 2、元素只能后进先出（LIFO）。
- 3、不能遍历整个 stack。
- 4、适配器，它可以将任意类型的序列容器转换为一个队列，一般使用 deque 作为支持的序列容器。
- 5、元素只能先进先出（FIFO）。
- 6、不能遍历整个 queue。
- 7、适配器，它可以将任意类型的序列容器转换为一个优先级队列，一般使用 vector 作为底层存储方式。
- 8、只能访问第一个元素，不能遍历整个 priority\_queue。
- 9、第一个元素始终是优先级最高的一个元素。

### queue

#### priority\_queue

建议：当需要 stack, queue, 或 priority\_queue 这种数据结构时，直接使用对应的容器类，不要使用 deque 去做它们类似的工作。

### Set

- 1、键和值相等。
- 2、键唯一。
- 3、元素默认按升序排列。
- 4、迭代器失效

删除：如果迭代器所指向的元素被删除，则该迭代器失效。

### map

- 1、键唯一。
- 2、元素默认按键的升序排列。
- 3、迭代器失效

删除：如果迭代器所指向的元素被删除，则该迭代器失效。

### hash 与 set、multiset、map、multimap 的结合

它里面的元素不一定是按键值排序的，而是按照所用的 hash 函数分派的，它能提供更快的搜索速度（当前和 hash 函数有关）。

建议：当元素的有序比搜索更重要时，应选用 set, multiset, map 或 multimap。否则，选用 hash\_set, hash\_multiset, hash\_map 或 hash\_multimap

建议：往容器中插入元素时，若元素在容器中的顺序无关紧要，请尽量加在最后面。若经常需要在序列容器的开头或中间增加或删除元素时，应选用 list。

建议：对关联容器而言，尽量不要使用 C 风格的字符串（即字符指针 char\*）作为键值。如

果非用不可，应显式地定义字符串比较运算符，即 `operator<`，`operator==`，`operator<=` 等。建议：当容器作为参数被传递时，请采用引用传递方式。否则将调用容器的拷贝构造函数，其开销是难以想象的。

STL 组建与平台无关，与应用无关，与数据类型无关。

## 第 6 章 算法

在 STL 源码分析中分节讲述了诸多算法，我在此仅仅罗列出其中几个我所关心的算法：

### 6.4.3 Copy

Copy 操作通常运用 `assignment operator` 或 `copy constructor`（copy 算法采用的是前者），但是某些元素类型拥有的是 `trivial assignment operator`，因此如果能够使用内存直接复制（如

c 语言的 memmove 或 memcpy)，便能提高效率，因此 SGI STL 的 copy 算法用尽各种办法，包括函数重载，型别特性（type traits），偏特化等编程技巧。

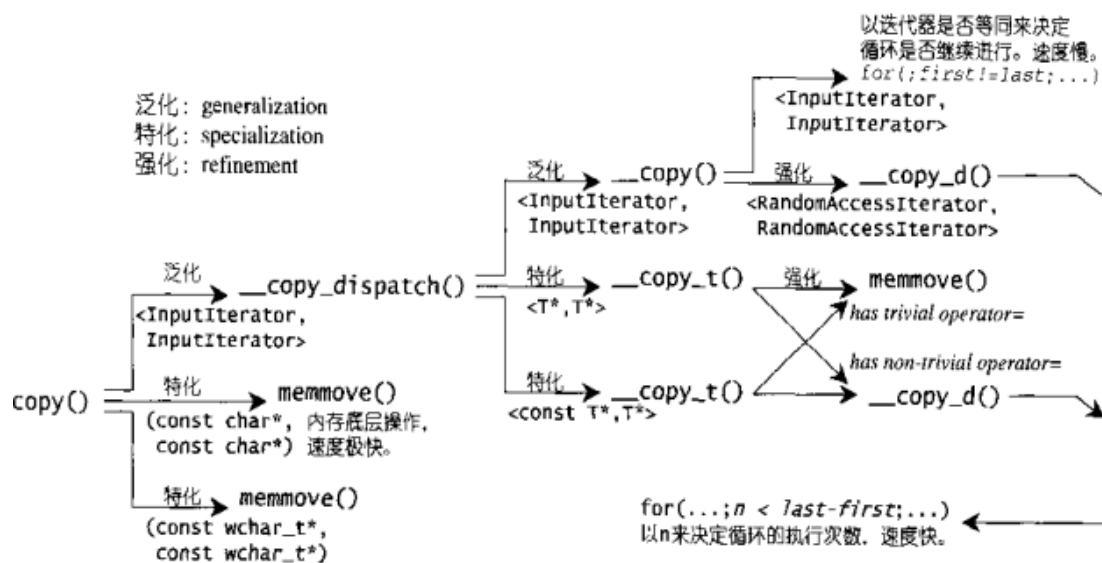


图 6-2 copy 算法完整脉络

Copy 算法将输入区间[first, last)内的元素复制到输出区间[result, result + (last - first))内。如果输入区间和输出区间完全没有重叠，copy 操作不会出现任何问题，否则需要特别注意：图 6-3 第二种情况可能产生错误。Copy 算法是一一进行元素赋值操作，如果输出区间的起点位于输入区间内，copy 算法可能会在输入区间的某些元素尚未被赋值之前，就覆盖其值，导致错误结果。我们这里用“可能”这个字眼，是因为如果 copy 算法根据其所接受的迭代器的特性决定使用 memmove()来执行，就不会发生上述错误，因为 memmove()会先将整个输入区间的内容复制下来，没有被覆盖的危险。

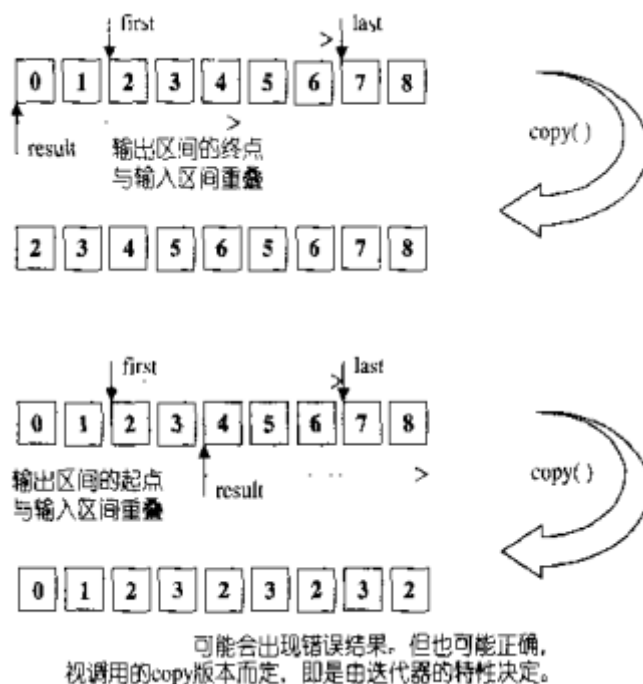


图 6-3 copy 算法，需要特别注意区间重叠的情况

注意，`copy` 更改的是`[result, result + (last - first))`中的迭代器所指对象，而非迭代器本身，他会为输出区间内的元素赋予新值，而不是产生新的元素。他不能改变输出区间的迭代器的个数。换句话说，`copy` 不能直接用来将元素插入空容器中。

#### 6.4.4 `copy_backward`

Template <class BidirectionalIterator1, class BidirectionalIterator2>

Inline BidirectionalIterator2 `copy_backward`(BidirectionalIterator1 first,  
BidirectionalIterator1 last,  
BidirectionalIterator2 result)

`Copy_backward()`将`[first, last)`区间内的每一个元素以逆行的方向赋值到以 `result - 1` 为起点，方向亦为逆行的区间上。`Copy_backward` 所接受的迭代器必须是 `BidirectionalIterators`。`Copy_backward` 也要注意像 `copy` 一样的区间重叠问题，如图 6-4.

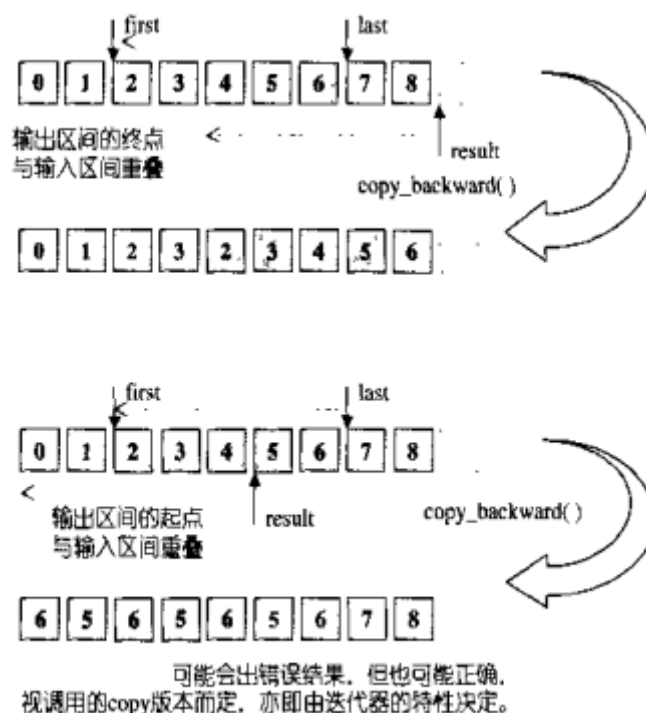


图 6-4 `copy_backward` 算法操作示意图，需要特别注意区间重叠情况

## 第 7 章 函数对象（仿函数）

仿函数（“行为类似函数”的对象）也称为函数对象，是一种具有函数特质的对象。调用者可以像函数一样的使用这些对象，例如在很多 STL 算法中，都可以看到，我们可以将一个方法作为模板内的参数传入到算法实现中，例如 `sort` 的时候我们可以根据我们传入的自定义的 `compare` 函数来进行比较排序。解决办法是使用函数指针，或者是将这个“操作”设计为一个所谓的仿函数，再用这个仿函数生成一个对象，并用这个对象作为算法的一个参数。

那为什么 STL 不使用函数指针而使用仿函数呢，因为函数指针不能满足 STL 对抽象性的要求，无法和 STL 的其他组件搭配以产生更加灵活的效果。

怎样实现这样一个仿函数呢？（可以直接使用对象名来使用函数）？答：我们必须自定义或者重载函数调用的运算符 `operator ()`，先产生类对象的一个匿名对象，再调用相应的函数。如：

```
#include <functional>
#include <iostream>
Using namespace std;

Int main(){
    Greater<int> ig;
    Cout << boolalpha << ig(4, 6); //
    Cout << greater<int>()(6, 4); //
}
```

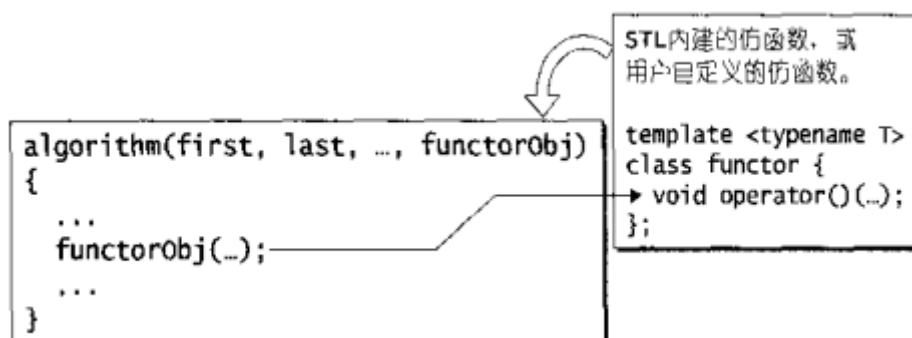


图 7-1 STL 仿函数与 STL 算法之间的关系

## 7.2 可配接 (adaptable) 的关键

STL 仿函数应该有能力被函数适配器修饰, 就像积木一样串接, 然而, 为了拥有配接能力, 每个仿函数都必须定义自己的 **associative types** (主要用来表示函数参数类型和返回值类型), 就想迭代器如果要融入整个 STL 大家庭, 也必须按照规定定义自己的 5 个相应的类型一样, 这些 **assocaiative type** 是为了让配接器可以取得仿函数的某些信息, 当然, 这些 **associative type** 都只是一些 **typedef**, 所有必要操作在编译器就全部完成了, 对程序的执行效率没有任何影响, 不会带来额外的负担。

STL 仿函数分类, 若以操作数个数划分, 可分为一元和二元仿函数, 若以功能划分, 可分为算术运算、关系运算、逻辑运算三大类。

仿函数的型别主要用来表现函数参数型别和返回值型别。为方便起见, `<stl_function.h>` 定义了两个 **classes**, 分别代表一元仿函数和二元仿函数, 其中没有任何 **data members** 或 **member functions**, 唯有一些型别定义, 任何仿函数, 只要依个人需求选择继承其中一个 **class**, 便自动拥有了那些相应型别, 也就自动拥有了适配能力。

### 7.2.1 一元仿函数 (unary\_function)

用来呈现一元函数的参数类型和返回值类型

```
Template<class Arg, class Result>
```

```
Struct unary_function
```

```
{
    Typedef Arg argument_type;
    Typedef Result result_type;
}
```

//自定义的一元仿函数可以继承上类来获得类型定义

```
Template<class T>
```

```
Struct negate:public unary_function<T,T>
```

```
{
    T operator()(const T& x)const {return  - x;}
};
```

### 7.2.2 binary\_function

用来呈现二元函数的第一个参数类型, 第二个参数类型, 和返回值类型

```
Template<class Arg1, class Arg2, class Result>
```

```
Struct binary_function
```

```
{
    Typedef Arg1 first_argument_type;
    Typedef Arg2 second_argument_type;
    Typedef Result result_type;
};
```

```
Template<class T>
```

```
Struct plus: public binary_function<T,T,T>
```

```

{
    T operator()(const T& x, const T& y) const { return x+y ;}
};

//以下适配器用于将某个二元仿函数转化为一元仿函数
Template<class Operation>
Class binder1st
{
Protected:
    Operation op;
    Typename Operation::first_argument_type value;
Public:
    Typename Operation::result_type
    operator() ( const typename Operation::second_argument_type& x) const{
        .....
    }
};

```

### 7.3 算术类仿函数

STL 内建的“算术类仿函数”，支持 加法，减法，乘法，除法，模数，否定。

```

// 以下6个为算术类 (Arithmetic) 仿函数
template <class T>
struct plus : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const { return x + y; }
};

template <class T>
struct minus : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const { return x - y; }
};

template <class T>
struct multiplies : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const { return x * y; }
};

template <class T>
struct divides : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const { return x / y; }
};

template <class T>
struct modulus : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const { return x % y; }
};

template <class T>
struct negate : public unary_function<T, T> {
    T operator()(const T& x) const { return -x; }
};

```

这些仿函数所产生的对象，用法和一般函数完全相同，当然，我们也可以产生一个无名的临时对象来履行函数功能。下面实例中，显示了两种用法：

```
// file: 7functor-arithmetic.cpp
#include <iostream>
#include <functional>
using namespace std;

int main()
{
    // 以下产生一些仿函数实体（对象）
    plus<int> plusobj;
    minus<int> minusobj;
    multiplies<int> multipliesobj;
    divides<int> dividesobj;
    modulus<int> modulusobj;
    negate<int> negateobj;

    // 以下运用上述对象，履行函数功能
    cout << plusobj(3,5) << endl;           // 8
    cout << minusobj(3,5) << endl;          // -2
    cout << multipliesobj(3,5) << endl;      // 15
    cout << dividesobj(3,5) << endl;         // 0
    cout << modulusobj(3,5) << endl;         // 3
    cout << negateobj(3) << endl;            // -3

    // 以下直接以仿函数的临时对象履行函数功能
    // 语法分析：functor<T>() 是一个临时对象，后面再接一对小括号
    // 意指调用 function call operator
    cout << plus<int>{}(3,5) << endl;        // 8
    cout << minus<int>{}(3,5) << endl;       // -2
    cout << multiplies<int>{}(3,5) << endl;  // 15
    cout << divides<int>{}(3,5) << endl;     // 0
    cout << modulus<int>{}(3,5) << endl;     // 3
    cout << negate<int>{}(3) << endl;        // -3
}
```

但是仿函数一般都不在这么单纯的情况下使用，仿函数的主要用途是为了搭配 STL 算法。下面例子，表示以 1 为基本元素，对 vector iv 中的每一个元素进行乘法运算：

```
Accumulate(iv.begin(), iv.end(), 1, multiplies<int>());
```

7.4 关系运算类仿函数（略）

7.5 逻辑运算类仿函数（略）



## 第 8 章 适配器 adapters

在 design patterns 一书中对 adaptor 模式定义如下: 将一个 class 的接口转换为另一个 class 的接口, 使原本因接口不兼容而不能合作的 classes, 可以一起运作。

### 8.1 配接器分类

#### 8.1.1 container adapters

容器 queue 和 stack 就是一种 container adapter, 他们修饰 deque 的接口形成一种新的容器接口。

#### 8.1.2 iterator adapters

Insert iterator, reverse iterator, istream iterator 都属于 iterator adaptor.

Insert iterator 将一般的 iterator 的赋值操作 (assign) 修饰为 插入 (insert) 操作, 例如从尾端插入操作的 back\_insert\_iterator, 从头端插入操作的 front\_insert\_iterator, 从任意位置插入操作的 insert\_iterator. 由于这三个 iterator adaptors 的使用接口不直观, STL 提供三个相应函数: back\_inserter(), front\_inserter(), inserter(), 如图 8-1:

辅助函数 (helper function)	实际产生的对象
Back_inserter (Container& x);	back_insert_iterator<Container>(x);
front_inserter (Container& x);	front_insert_iterator<Container>(x);
Inserter(Container& x, Iterator i);	insert_iterator<Container> (x, Container::iterator(i));

图 8-1 三个辅助函数, 使三种 iterator adapters 更易使用

Reverse iterator 将一般的 iterator 的前进方向逆转, 使得原本应该前进的++操作变成了后退操作, 使原本后退的--操作符编程了前进操作符。Zhezhongiterator 主要用在“从尾端开始进行”的算法上。

Iostream iterator 将 iterator 绑定到某个 istream 上, 绑定到 istream 对象上的称为 istream\_iterator, 便拥有输入功能; 绑定到 ostream 对象上的称为 ostream\_iterator, 便拥有输出功能。下面实例, 描述了上述三种 iterator adapters 之运用:

```
//file: biterator-adapter.cpp
#include <iterator> //for iterator adapters
#include <deque>
#include <algorithm>
#include <iostream>
Using namespace std;

Int main(){
```

```

//将 outside 绑定到 cout，每次对 outfile 指派一个元素，就后接一个 “ ”
Ostream_iterator<int> outite(cout, “ ”);
Int ia[] = {0, 1, 2, 3, 4, 5};
Deque<int> id(ia, ia+6);

//将所有的元素都拷贝到 outite（也就是拷贝到 cout）
Copy(id.begin(), id.end(), outite);//输出 0 1 2 3 4 5
Cout<<endl;

//将 ia[]的部分元素拷贝到 id 内，使用 front_insert_iterator
//注意，front_insert_iterator 会将 assign 操作改为 push_front 操作
Copy(ia+1, ia+2, front_inserter(id));
Copy(id.begin(), id.end(), outite);//1 0 1 2 3 4 5
Cout<<endl;

//将 ia[]的部分元素拷贝到 id 内，使用 back_insert_iterator。
Copy(ia+3, ia+4, back_inserter(id));
Copy(id.begin(), id.end(), outite);//1 0 1 2 3 4 5 3

//搜寻元素 5 的位置
Deque<int>::iterator ite = find(id.begin(), id.end(), 5);
//将 ia[]的部分元素拷贝到 id 内，使用 insert_iterator
Copy(ia+0, ia+3, inserter(id, ite));
Copy(id.begin(), id.end(), outite);//1 0 1 2 3 4 0 1 2 5 3
Cout<<endl;

//将所有元素逆向拷贝到 outite
//rbegin()和 rend()与 reverse_iterator 有关
Copy(id.rbegin(), id.rend(), outite);//3 5 2 1 0 4 3 2 1 0 1
Cout<<endl;

//一下将 inite 绑定到 cin，将元素拷贝到 inite，直到 eos 出现
Istream_iterator<int> inite(cin), eos;//eos:end of stream
Copy(inite, eos, inserter(id, id.begin()));
Copy(id.begin(), id.end(), outite);//
}

```

### 8.1.3 functor adapters

Function adapters 非常灵活，可以配接、配接、再配接，这些配接包括：bind，negate，compose，以及对一般函数或成员函数的修饰（使其成为一个仿函数）。

C++ standard 规定这些配接器的接口可由<functional>获得，SGI STL 将之定义于<stl\_function.h>。

Function adapters 的价值在于，通过他们之间的绑定/组合/修饰能量，几乎可以无限制的创造出各种可能的表达式。如：

1. 我们希望找出某个序列中不小于 12 的所有元素的个数，可以如下：  
`not1(bind2nd(less<int>(), 12))`, 该式将 `less<int>()` 的第二个参数绑定为 12，再加上否定操作，便形成了“不小于 12”的语义。
2. 假设如果我们希望对容器中的每一个元素 `v` 进行  $(v+2) * 3$  的操作，我们可以使用 `compose1(f(x), g(y))`，并令  $f(x) = x * 3$ ,  $g(y) = y + 2$ 。

## 8.2 container adapters

Template<class T, class Sequence=deque<T> >

Class stack

{

Protected:

**Sequence c; //底层容器**

...

};

Queue 也是同样的道理

Template< class T, class Sequence = deque<T> >

Class queue

{

Protected:

**Sequence c; //底层容器**

...

};

## 8.3 iterator adapters

### 8.3.1 insert iterators

下面是三种 insert iterator 的完整实现。主要思想：每一个 insert iterators 内部都维护一个容器（必须由用户指定）；容器当然有自己的迭代器，于是当用户对 insert iterators 做赋值（assign）操作时，就在 insert iterators 中被转为对该容器的迭代器做插入操作，也就是说在 insert iterators 的 operator= 操作符中调用底层容器的 `push_front()` 或 `push_back()` 或 `insert()` 函数。至于其它的迭代器惯常行为如 `operator++`, `operator++(int)`, `operator*` 都被关闭功能，更没有提供 `operator--` 或 `operator--(int)` 或 `operator->` 等。因此对 insert iterators 来讲，前进，后退，取值，成员取用等操作没有意义。

Template<class Container>

Class back\_insert\_iterator

{

Protected:

**Container\* container; //底层容器**

Public:

**Typedef output\_iterator\_tag iterator\_category; //注意类型**

Typedef void value\_type;

Typedef void difference\_type;

Typedef void pointer;

Typedef void reference;

```

Explicit back_insert_iterator(Container& x) : container(&x){}
back_insert_iterator<Container>& operator=( const typename Container::value_tyep& value)
{
    Container->push_back( value );
    Return *this;
}
//下面的操作不起作用（关闭功能）
Back_insert_iterator<Container>& operator*() {return *this;}
Back_insert_iterator<Container>& operator++(){return *this;}
Back_insert_iterator<Container>& operator++(int){return *this;}

//这是一个辅助函数，帮助我们方便使用 back_insert_iterator
Template<class container>
Inline back_insert_iterator<container> back_inserter(container &x){
    Return back_insert_iterator<container>(x);
}
};

```

```

Template<class Container>
Class front_insert_iterator
{
Protected:
    Container* container; //底层容器
Public:
    Typedef output_iterator_tag iterator_categoty;//注意类型
    Typedef void value_type;
    Typedef void difference_type;
    Typedef void pointer;
    Typedef void reference;
    Explicit front_insert_iterator(Container& x) : container(&x){}
    front_insert_iterator<Container>& operator=( const typename Container::value_tyep& value)
    {
        Container->push_front( value );
        Return *this;
    }
    //下面的操作不起作用（关闭功能）
    front_insert_iterator<Container>& operator*() {return *this;}
    front_insert_iterator<Container>& operator++(){return *this;}
    front_insert_iterator<Container>& operator++(int){return *this;}

    //这是一个辅助函数，帮助我们方便使用 back_insert_iterator
    Template<class container>
    Inline front_insert_iterator<container> front_inserter(container &x){
        Return front_insert_iterator<container>(x);
    }
};

```

```

    }
};

Template<class Container>
Class insert_iterator
{
Protected:
    Container* container; //底层容器
Public:
    Typedef output_iterator_tag iterator_category; //注意类型
    Typedef void value_type;
    Typedef void difference_type;
    Typedef void pointer;
    Typedef void reference;
    insert_iterator(Container& x, typename Container::iterator i) : container(&x), iter(i) {}
    insert_iterator<Container>& operator=( const typename Container::value_type& value)
    {
        Container->insert(iter, value );
        ++iter; //是 insert iterator 永远随其目标贴身移动
        Return *this;
    }
    //下面的操作不起作用（关闭功能）
    Back_insert_iterator<Container>& operator*() {return *this;}
    Back_insert_iterator<Container>& operator++() {return *this;}
    Back_insert_iterator<Container>& operator++(int) {return *this;}

    //这是一个辅助函数，帮助我们方便使用 back_insert_iterator
    Template<class container>
    Inline insert_iterator<container> inserter(container &x, iterator i){
        Typedef typename container::iterator iter;
        Return insert_iterator<container>(x, iter(i));
    }
};

```

### 8.3.2 reverse iterators

所谓 reverse iterator 就是将迭代器的移动行为倒转。例如：

Copy( id.rbegin(), id.rend(), ite); //使用的是 reverse iterator, 那么会将所有元素逆向拷贝到 ite 所指的位置上去。

先看看 vector 中 rbegin 和 rend 的定义：

```

Template< class T, class Alloc=alloc>
Class Vector
{

```

```

Public:
    Typedef T value_type;
    Typedef value_type* iterator;
    Typedef reverse_iterator<iterator> reverse_iterator;
    Reverse_iterator rbegin(){return reverse_iterator( end() );}
    //借助 reverse_iterator 构造函数，传入普通迭代器，来获取一个 reverse 迭代器
    Reverse_iterator rend() {return reverse_iterator( begin() );}
    ...
};

```

再看看 deque 中 rbegin 和 rend 的定义：

```

Template<class T, class Alloc = alloc, size_t BufSiz = 0>
Class deque {
Public:
    Typedef __deque_iterator<T, T&, Y*, BufSiz> iterator;
    Typedef reverse_iterator<iterator> reverse_iterator;
    Iterator begin() {return start;}
    Iterator end() {return finish;}
    Reverse_iterator rbegin(){return reverse_iterator( end() );}
    //借助 reverse_iterator 构造函数，传入普通迭代器，来获取一个 reverse 迭代器
    Reverse_iterator rend() {return reverse_iterator( begin() );}
}

```

其他任何双向序列容器，只要提供了 begin 和 end，那么它的 rbegin 和 rend 就会类似于上述情况。而单向序列容器不可以使用 reserve iterator，有些容器 stack 等没有提供 begin/end，当然也不能使用 rbegin/rend。

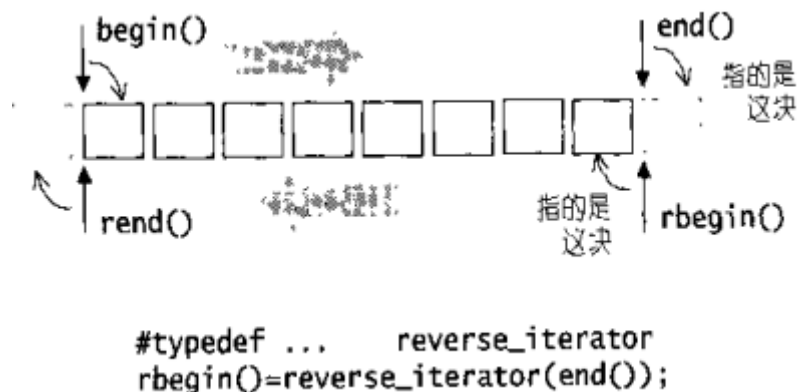


图 8-3 正向迭代器和逆向迭代器关系

下面是 reverse\_iterator 的源码

```

Template< class Iterator>
Class reverse_iterator
{

```

Protected:

Iterator current;

Public:

//逆向迭代器的 5 种型别都和正向迭代器相同

Typedef typename iterator\_traits<Iterator>::iterator\_category iterator\_category;

Typedef typename iterator\_traits<Iterator>::value\_type value\_type;

Typedef typename iterator\_traits<Iterator>::difference\_type difference\_type;

Typedef typename iterator\_traits<Iterator>::pointer pointer;

Typedef typename iterator\_traits<Iterator>::reference reference;

Typedef Iterator iterator\_type;//正向迭代器

Typedef reverse\_iterator<Iterator> self;//逆向迭代器

Public:

Reverse\_iterator(){}

//这个 ctor 将某个 reverse\_iterator 与某个迭代器 x 结合起来

Explicit reverse\_iterator( iterator\_type x ) : current(x){ }

Reverse\_iterator( const self& x):current( x.current){ }

Iterator\_type base() const{return current;}//取出对应的正向迭代器

Reference operator\*() const

{

Iterator tmp= current;

return \*--tmp;

}

Pointer operator->() const {return &(operator\*());}

Self& operator++()

{

--current;

Return \*this;

}

Self& operator++(int)

{

Self tmp=\*this;

--current;

Return tmp;

}

Self operator+(difference\_type n) const

{

Return self( current-n);

}

Self& operator+=(difference\_type n)

{

Current += n;

Return \*this;

}

};

### 8.3.3 stream iterators

**Stream iterator:** 将迭代器绑定到一个 stream 对象上，例如绑定到 istream 对象的称为 istream\_iterator，绑定到 ostream 对象的成为 ostream\_iterator。

所谓绑定到一个 istream 对象，其实就是在 istream 的 iterator 内部维护一个 istream member，客户端对这个迭代器所做的 operator ++ 等操作，会被导引调用迭代器内部所含的那个 istream member 的输入操作 (>>)。这个迭代器是一个 input iterator，不具备 operator--。

所谓绑定到一个 ostream 对象，其实就是在 ostream 的 iterator 内部维护一个 ostream member，客户端对这个迭代器所做的 operator ++ 等操作，会被导引调用迭代器内部所含的那个 ostream member 的输出操作 (<<)。这个迭代器是一个 output iterator。

Istream iterator 源码:

```
Template< class T, class Distance=ptrdiff_t >
```

```
Class istream_iterator
```

```
{
    Friend bool operator== __STL_NULL_TMPL_ARGS( const istream_iterator<T,Distance>&x,
        const istream_iterator<T,Distance>&y);
```

Protected:

```
    Istream* stream;
```

```
    T value;
```

```
    Bool end_marker;
```

```
    Void read()
```

```
{
    End_marker=(*stream) ? true: false;
    If( end_marker )
        *stream>>value; //关键在这里
    //输入后，stream 的状态可能改变，所以下面再判断一次决定 end_marker
    //当读到 eof 或者类型不符合的资料时，stream 处于 false
    End_maker = (*stream) ? true: false;
}
```

Public:

```
    Typedef      input_iterator_tag      iterator_category;
    Typedef      T                          value_type;
    Typedef      Distance                  difference_type;
    Typedef      const T&                  reference;
    Typedef      const T*                  pointer;
```

```
Istream_iterator():stream(&cin),end_maker(false){ }
```

```
Istream_iterator( istream& s) : stream(&s){read();}
```

```
Reference operator*() const{return value;}
```

```
Pointer operator->() const {return &(operator*());}
```



```

Istream_iterator<T,Distance>& operator++()
//istream iterator 的++操作其实就是底层 istream 的 read 操作
{
    Read();
    Return *this;
}
Istream_iteraor<T, Distance> operator++(int) {
    Istream_iterator<T, Dinstance> tmp = *this;
    Read();
    Return tmp;
}
};

```

Ostream iterator 源码:

Template <class T>

Class ostream\_iterator {

Protected:

Ostream \*stream;

Const char \* string;//每次输出后的间隔符号

Public:

Typedef output\_iterator\_tag iterator\_category;

Typedef void value\_type;

Typedef void diffence\_type;

Typedef void pointer;

Typedef void reference;

Ostream\_iterator(ostream &s):stream(&s), string(0){ }

Ostream\_iterator(ostream& s, const char\* s ):stream(&s), string(s) { }

//对迭代器做赋值操作，就代表要输出数据

Ostream\_iteraor<T>& operator= (const T& value) {

\*stream << value;//关键，输出数据

If(string) \*stream << string;//输出间隔符

Return \*this;

}

//注意一下三个操作

Ostream\_iterator<T>& operator\*() {return \*this;}

Ostream\_iterator<T>& operator++() {return \*this;}

Ostream\_iterator<T>& operator++(int) {return \*this;}

};

## 8.4 function adapters

我们知道：

容器是以 class template 完成的；

算法是以 function template 完成的；

仿函数是一种 operator()重载的 class template；

迭代器是一种将 operator++和 operator\*等指针习惯常行为重载的 class template；

应用于容器和迭代器身上的 adapter 也是一种 class template；

就像 container adapter 内藏一个 container member， reverse iterator adapter 内藏一个 iterator member， stream iterator adapter 内藏一个 point to container 一样，每一个 function adapters 也内藏了一个 member object，其型别等同于它所要配接的对象（那个对象当然是一个“可配接的仿函数”），见图 8-6.当 function adapter 有了完全属于自己的一份修饰对象（的副本）在手，他就成了该修饰对象（的副本）的主任，也就有资格调用该修饰对象（一个仿函数）。

辅助函数 (helper function)	实际产生的配接器对象形式	内藏成员的类型
<code>bind1st(cost Op&amp; op,           const T&amp; x);</code>	<code>binder1st&lt;Op&gt; (op, arg1_type(x))</code>	<code>Op</code> (二元仿函数)
<code>bind2nd(cost Op&amp; op,           const T&amp; x);</code>	<code>binder2nd&lt;Op&gt; (op, arg2_type(x))</code>	<code>Op</code> (二元仿函数)
<code>not1(cost Pred&amp; pred);</code>	<code>unary_negate&lt;Pred&gt; (pred)</code>	<code>Pred</code> 返回布尔值的仿函数
<code>not2(cost Pred&amp; pred);</code>	<code>binary_negate&lt;Pred&gt; (pred)</code>	<code>Pred</code> 返回布尔值的仿函数
<code>compose1(const Op1&amp; op1,           const Op2&amp; op2);</code>	<code>unary_compose&lt;Op1,Op2&gt; (op1, op2)</code>	<code>Op1, Op2</code>
<code>compose2(const Op1&amp; op1,           const Op2&amp; op2,           const Op3&amp; op3);</code>	<code>binary_compose &lt;Op1,Op2,Op3&gt; (op1, op2, op3)</code>	<code>Op1, Op2, Op3</code>
<code>ptr_fun (Result(*fp)(Arg));</code>	<code>pointer_to_unary_function &lt;Arg, Result&gt;(f)</code>	<code>Result(*fp)(Arg)</code>
<code>ptr_fun (Result(*fp)(Arg1,Arg2));</code>	<code>pointer_to_binary_function &lt;Arg1, Arg2, Result&gt;(f)</code>	<code>Result(*fp) (Arg1,Arg2)</code>
<code>mem_fun(S (T::*f)());</code>	<code>mem_fun_t&lt;S,T&gt;(f)</code>	<code>S (T::*f)()</code>
<code>mem_fun(S (T::*f)() const);</code>	<code>const_mem_fun_t&lt;S,T&gt;(f)</code>	<code>S (T::*f)() const</code>
<code>mem_fun_ref(S (T::*f)());</code>	<code>mem_fun_ref_t&lt;S,T&gt;(f)</code>	<code>S (T::*f)()</code>
<code>mem_fun_ref (S (T::*f)() const);</code>	<code>const_mem_fun_ref_t&lt;S,T&gt; (f)</code>	<code>S (T::*f)() const</code>
<code>mem_fun1(S (T::*f)(A));</code>	<code>mem_fun1_t&lt;S,T,A&gt;(f)</code>	<code>S (T::*f)(A)</code>
<code>mem_fun1(S (T::*f)(A) const);</code>	<code>const_mem_fun1_t&lt;S,T,A&gt;(f)</code>	<code>S (T::*f)(A) const</code>
<code>mem_fun1_ref(S (T::*f)(A));</code>	<code>mem_fun1_ref_t&lt;S,T,A&gt;(f)</code>	<code>S (T::*f)(A)</code>
<code>mem_fun1_ref (S (T::*f)(A) const);</code>	<code>const_mem_fun1_ref_t &lt;S,T,A&gt;(f)</code>	<code>S (T::*f)(A) const</code>

⬢ `compose1` 和 `compose2` 不在 C++ *Standard* 规范之内。

⬢ 最后四个辅助函数在 C++ *Standard* 内已去除名称中的 '1'。

图 8-6 不同的 function adapters 内藏不同的成员

图 8-7 展示了 `count_if()` 搭配 `bind2nd(less<int>(), 12)` 的例子。

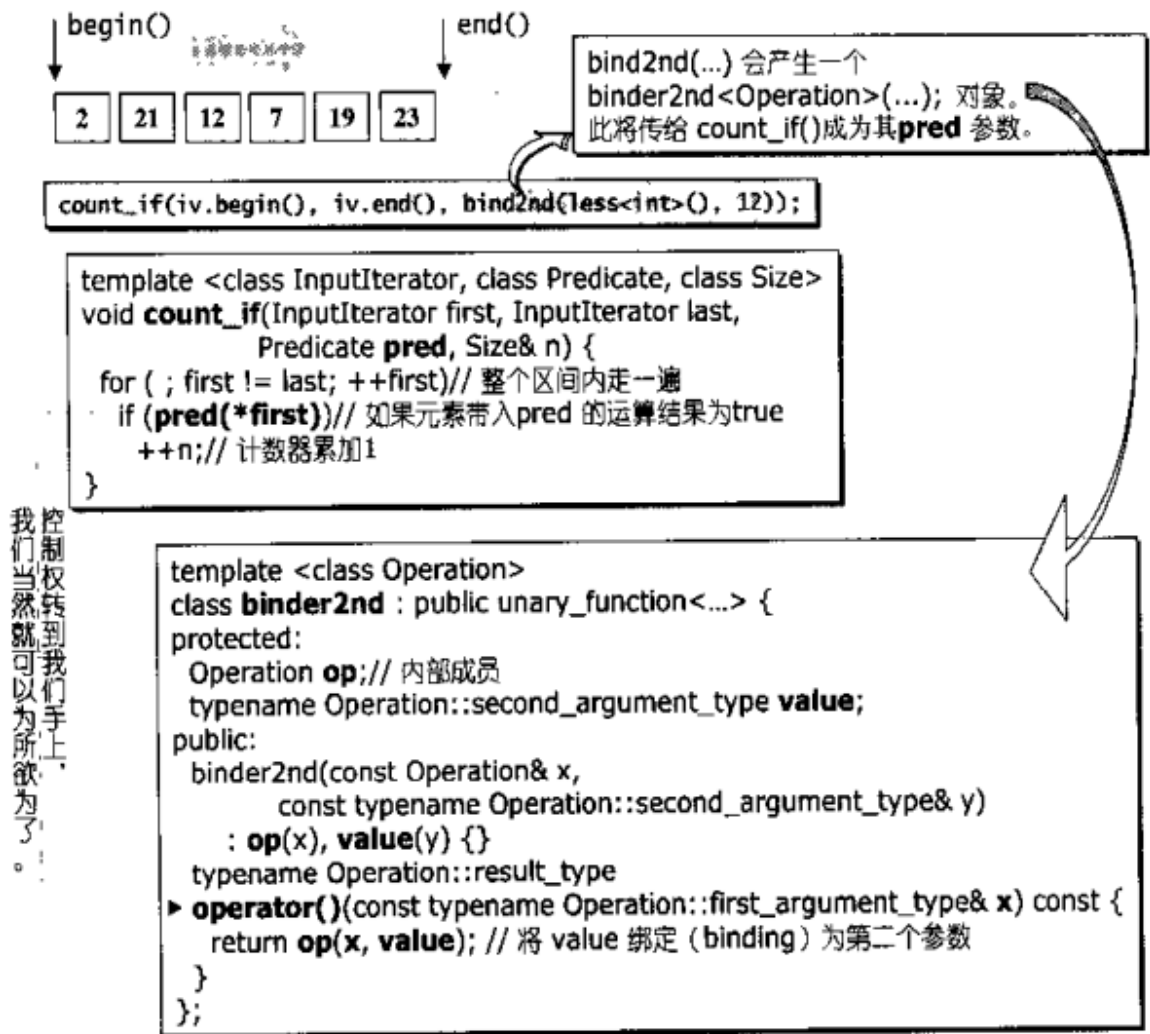


图 8-7 鸟瞰 count\_if() 和 bind2nd(less<int>(), 12) 的搭配实例。此图等于是相关源代码的接口整理，搭配 bind2nd() 以及 class binder2nd 源代码阅读，更得益处。图中浅色底纹方块为客户端调用 count\_if() 实况；循着箭头行进，便能理解的整个合作机制。

#### 8.4.1 对返回值进行逻辑否定：not1, not2

源代码中经常出现的 pred 一词，是 predicate 的缩写，意指会返回真假值的表达式。

```
Template<class Predicate>
```

```
Class unary_negate : public unary_function< typename Predicate::argument_type, bool>
```

```
{
```

```
Protected:
```

```
    Predicate pred;
```

```
Public:
```

```
    Explicit unary_negate( const Predicate& x): pred(x){}
```

```
    Bool operator()(const typename Predicate::argument_type& x) const
```

```

{
    Return !pred(x); //将 pred 的运算结果加上否定运算
}
};

```

Template<class Predicate>

Inline unary\_negate<Predicate> not1( const Predicate& pred)

```

{
    Return unary_negate< Predicate>(pred);
}

```

```

//-----
// 以下配接器用来表示某个 Adaptable Binary Predicate 的逻辑负值
template <class Predicate>
class binary_negate
    : public binary_function<typename Predicate::first_argument_type,
                           typename Predicate::second_argument_type,
                           bool> {
protected:
    Predicate pred;      // 内部成员
public:
    explicit binary_negate(const Predicate& x) : pred(x) {}
    bool operator()(const typename Predicate::first_argument_type& x,
                    const typename Predicate::second_argument_type& y) const {
        return !pred(x, y); // 将 pred 的运算结果加上否定 (negate) 运算
    }
};

// 辅助函数, 使我们得以方便使用 binary_negate<Pred>
template <class Predicate>
inline binary_negate<Predicate> not2(const Predicate& pred) {
    return binary_negate<Predicate>(pred);
}

```

#### 8.4.2 对参数进行绑定: bind1st, bind2nd

//以下配接器用来将某个 adaptable binary function 转换为 unary function

Template<class Operation>

Class binder1st: public unary\_function<typename Operation::second\_argument\_type,

typename Operation::result\_type>

```
{
```

Protected:

Operation op;

typename Operation::first\_argument\_type value;

Public:

Binder1st( const Operation& x, const typename Operation::first\_argument\_type& y)

:op(x),value(y) {} //将表达式和第一参数记录于内部成员

```

typename Operation::result_type
Operation() ( const typename Operation::second_argument_type& x)const
{
    return op(value,x);//实际调用表达式并将 value 绑定为第一参数
}
};

```

//辅助函数，让我们方便使用 binder1st<op>

//初始化时指定操作和绑定第一个参数，然后就可以像使用仿函数一样传入第二个参数

Template<class Operation,class T>

Inline binder1st<Operation> bind1st( const Operation& op, const T& x)

```

{
    Typedef typename Operation::first_argument_type arg1_type;
    Return binder1st<Operation>( op, arg1_type(x));
}

```

// 以下适配器用来将某个 Adaptable Binary function 转换为 Unary Function

template <class Operation>

class binder2nd

```

: public unary_function<typename Operation::first_argument_type,
                        typename Operation::result_type> {
protected:

```

```

    Operation op; // 内部成员

```

```

    typename Operation::second_argument_type value; // 内部成员

```

```

public:

```

```

    // constructor

```

```

    binder2nd(const Operation& x,

```

```

                const typename Operation::second_argument_type& y)

```

```

        : op(x), value(y) {} // 将表达式和第一参数记录于内部成员

```

```

    typename Operation::result_type

```

```

    operator()(const typename Operation::first_argument_type& x) const {

```

```

        return op(x, value); // 实际调用表达式，并将 value 绑定为第二参数

```

```

    }

```

```

};

```

// 辅助函数，让我们得以方便使用 binder2nd<Op>

Template <class Operation, class T>

inline binder2nd<Operation> bind2nd(const Operation& op, const T& x)

```

{

```

```

    typedef typename Operation::second_argument_type arg2_type;

```

```

    return binder2nd<Operation>(op, arg2_type(x));

```

```

    // 以上，注意，先把 x 转型为 op 的第二参数型别

```

```

}

```

#### 8.4.3 用于函数合成: compose1, compose2

//已知两个 adaptable unary functions f(), g(),以下配接器用来产生一个 h(),使得  $h(x) = f(g(x))$

```
Template< class Operation1, class Operation2>
```

```
Class unary_compose:
```

```
Public unary_function<typename Operation2::argument_type,
```

```
Typename Operation1::result_type>
```

```
{
```

```
Protected:
```

```
    Operation1 op1;
```

```
    Operation2 op2;
```

```
Public:
```

```
    Unary_compose( const Operation1& x, const Operation2& y)
```

```
    : op1(x),op2(y){}
```

```
    Typename Operation1::result_type operation()(consti typename Operation2::argument_type&
    x)const{
```

```
        return op1(op2(x));}
```

```
};
```

```
//辅助函数，让我们方便运用 unary_compose<op1, op2>
```

```
//初始化传入两个 op 方法，调用（）操作是则使用 op2(x)的返回值作为 op1 的参数
```

```
Template<class Operation1, class Operation2>
```

```
Inline unary_compse<Operation1,Operation2>
```

```
Compose1(const Operation1& op1, const Operation2& op2)
```

```
{
```

```
    Return unary_compose< Operation1, Operation2>(op1,op2);
```

```
}
```

```

// 已知一个 Adaptable Binary Function f 和
// 两个 Adaptable Unary Functions g1,g2.
// 以下配接器用来产生一个 h, 使 h(x) = f(g1(x),g2(x))
template <class Operation1, class Operation2, class Operation3>
class binary_compose
    : public unary_function<typename Operation2::argument_type,
                           typename Operation1::result_type> {
protected:
    Operation1 op1;      // 内部成员
    Operation2 op2;      // 内部成员
    Operation3 op3;      // 内部成员
public:
    // constructor, 将三个表达式记录于内部成员
    binary_compose(const Operation1& x, const Operation2& y,
                   const Operation3& z) : op1(x), op2(y), op3(z) {}

    typename Operation1::result_type
    operator()(const typename Operation2::argument_type& x) const {
        return op1(op2(x), op3(x));    // 函数合成
    }
};

// 辅助函数, 让我们得以方便运用 binary_compose<Op1,Op2,Op3>
template <class Operation1, class Operation2, class Operation3>
inline binary_compose<Operation1, Operation2, Operation3>
compose2(const Operation1& op1, const Operation2& op2,
          const Operation3& op3) {
    return binary_compose<Operation1, Operation2, Operation3>
        (op1, op2, op3);
}

```

#### 8.4.4 用于函数指针: ptr\_fun

这种配接器使我们可以讲一般函数当做仿函数使用, 一般函数当做仿函数传递给 STL 算法, 如果不是用这里所说的两个配接器先包装, 所使用的一般函数将无配接能力, 野菊无法和前面小节介绍过的其它配接器接轨。

//以下配接器其实就是把一个一元函数指针包起来: 当仿函数被使用时, 就调用该函数指针

```

Template<class Arg, class Result>

```

```

Class pointer_to_unary_function:public unary_function<Arg,Result>

```

```

{

```

```

Protected:

```

```

    Result (*ptr)(Arg); //内部成员 一个函数指针, 返回 Result 类型, 参数为 arg

```

```

Public:

```

```

    Pointer_to_unary_function(){}

```

```

    //以下函数将函数指针记录于内部成员之中

```

```

    Explicit pointer_to_unary_function( Result(*x)(arg)) : ptr(x){}

```

```

    //以下通过函数指针执行函数

```

```

    Result operation()(Arg x) const{return ptr(x);}

```



```
};
```

```
Template< class Arg, class Result>
Inline pointer_to_unary_function<Arg,Result>
Ptr_fun(Result(*x) (Arg))
{
    Return pointer_to_unary_function<Arg,Result>(x);
}
```

```
//-----
// 以下配接器其实就是把一个二元函数指针包起来:
// 当仿函数被使用时,就调用该函数指针
template <class Arg1, class Arg2, class Result>
class pointer_to_binary_function
    : public binary_function<Arg1, Arg2, Result> {
protected:
    Result (*ptr)(Arg1, Arg2); // 内部成员,一个函数指针
public:
    pointer_to_binary_function() {}
    // 以下 constructor 将函数指针记录于内部成员之中
    explicit pointer_to_binary_function(Result (*x)(Arg1, Arg2))
        : ptr(x) {}

    // 以下,通过函数指针执行函数
    Result operator()(Arg1 x, Arg2 y) const { return ptr(x, y); }
};

// 辅助函数,让我们得以方便使用 pointer_to_binary_function
template <class Arg1, class Arg2, class Result>
inline pointer_to_binary_function<Arg1, Arg2, Result> // 返回值型别
ptr_fun(Result (*x)(Arg1, Arg2)) {
    return pointer_to_binary_function<Arg1, Arg2, Result>(x);
}
```

#### 8.4.5 用于成员函数指针: mem\_fun, mem\_fun\_ref

这种配接器使我们能够将成员函数当做仿函数来使用,于是成员函数可以搭配各种泛型算法。当容器的元素类型是 X&或 X\*,而我们又以虚拟成员函数作为仿函数时,便可以借由泛型算法完成所谓的多态调用。这是泛型与多态之间的重要接轨。

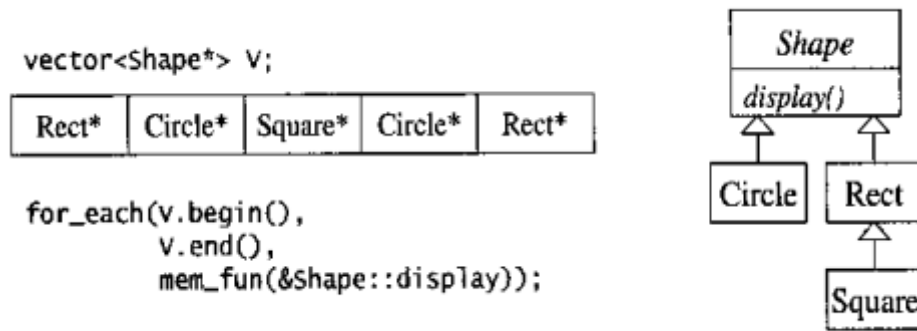


图 8-8 图右是类阶层体系，图左是实例所产生的容器状态

```

#include<iostream>
#include <vector>
#include <algorithm>
#include <functional>
Using namespace std;

Class shape
{public: virtual void display()=0;};

Class rect : public shape
{ public: virtual void display() {cout << "Rect";}};

Class circle : public shape
{ public: virtual void display() {cout << "Circle";}};

Class square : public rect
{ public: virtual void display() {cout << "Square";}};

Int main(){
    Vector<shape*> v;
    v.push_back(new rect);
    v.push_back(new circle);
    v.push_back(new square);
    v.push_back(new circle);
    v.push_back(new rect);

    for(int I = 0; I < v.size(); i++)
        (v[i])>display();
    Cout<<endl;// Rect Circle Square Circle Rect

    For_each(v.begin(), v.end(), mem_fun(&shape::display));
    Cout<<endl; // Rect Circle Square Circle Rect
}

```

请注意，就语法而言，你不能写：

```
For_each(v.begin(), v.end(), &shape::display);
```

也不能写：

```
For_each(v.begin(), v.end(), shape::display);
```

一定要配接器 `mem_fun` 修饰成员函数，才能被算法 `for_each` 接受。

另一个需要注意的是，虽然多态支持 `pointer` 或 `reference` 起作用，但是很可惜，STL 容器只支持“实值语义”，不支持“引用语义”，所以下面代码无法通过编译：

```
Vector<shape &> v;
```

```
Template< class S, class T>
```

```
Class mem_fun_t:public unary_function<T*, S>
```

```
{
```

```
Public:
```

```
Explicit mem_fun_t (S {T::*pf}()): f(pf){}
```

```
S operator()(T* p) const {return (p->*f)();}
```

```
Private:
```

```
S (T::*f)();
```

```
};
```

// 知道返回类型 S 和类类型 T，使用 `T::*p` 表示类内部的成员函数地址，作为参数传入，记录该成员函数。

```
// “无任何参数”、“通过 pointer 调用”、“non-const 成员函数”
template <class S, class T>
class mem_fun_t : public unary_function<T*, S> {
public:
    explicit mem_fun_t(S (T::*pf)()) : f(pf) {} // 记录下来
    S operator()(T* p) const { return (p->*f)(); } // 转调用
private:
    S (T::*f)(); // 内部成员, pointer to member function
};

// “无任何参数”、“通过 pointer 调用”、“const 成员函数”
template <class S, class T>
class const_mem_fun_t : public unary_function<const T*, S> {
public:
    explicit const_mem_fun_t(S (T::*pf)() const) : f(pf) {}
    S operator()(const T* p) const { return (p->*f)(); }
private:
    S (T::*f)() const; // 内部成员, pointer to const member function
};
```

```

// “无任何参数”、“通过 reference 调用”、“non-const 成员函数”
template <class S, class T>
class mem_fun_ref_t : public unary_function<T, S> {
public:
    explicit mem_fun_ref_t(S (T::*pf)()) : f(pf) {} // 记录下来
    S operator()(T& r) const { return (r.*f)(); } // 转调用
private:
    S (T::*f)(); // 内部成员, pointer to member function
};

// “无任何参数”、“通过 reference 调用”、“const 成员函数”
template <class S, class T>
class const_mem_fun_ref_t : public unary_function<T, S> {
public:
    explicit const_mem_fun_ref_t(S (T::*pf)() const) : f(pf) {}
    S operator()(const T& r) const { return (r.*f)(); }
private:
    S (T::*f)() const; // 内部成员, pointer to const member function
};

```

```

// “有一个参数”、“通过 pointer 调用”、“non-const 成员函数”
template <class S, class T, class A>
class mem_fun1_t : public binary_function<T*, A, S> {
public:
    explicit mem_fun1_t(S (T::*pf)(A)) : f(pf) {} // 记录下来
    S operator()(T* p, A x) const { return (p->*f)(x); } // 转调用
private:
    S (T::*f)(A); // 内部成员, pointer to member function
};

// “有一个参数”、“通过 pointer 调用”、“const 成员函数”
template <class S, class T, class A>
class const_mem_fun1_t : public binary_function<const T*, A, S> {
public:
    explicit const_mem_fun1_t(S (T::*pf)(A) const) : f(pf) {}
    S operator()(const T* p, A x) const { return (p->*f)(x); }
private:
    S (T::*f)(A) const; // 内部成员, pointer to const member function
};

```

```

// “有一个参数”、“通过 reference 调用”、“non-const 成员函数”
template <class S, class T, class A>
class mem_fun1_ref_t : public binary_function<T, A, S> {
public:
    explicit mem_fun1_ref_t(S (T&::*pf)(A)) : f(pf) {} // 记录下来
    S operator()(T& r, A x) const { return (r.*f)(x); } // 转调用
private:
    S (T&::*f)(A); // 内部成员, pointer to member function
};

// “有一个参数”、“通过 reference 调用”、“const 成员函数”
template <class S, class T, class A>
class const_mem_fun1_ref_t : public binary_function<T, A, S> {
public:
    explicit const_mem_fun1_ref_t(S (T::*pf)(A) const) : f(pf) {}
    S operator()(const T& r, A x) const { return (r.*f)(x); }
private:
    S (T::*f)(A) const; // 内部成员, pointer to const member function
};

```

//mem\_func adapter 的辅助函数: mem\_fun, mem\_fun\_ref

```

Template <class S, class T>

```

```

Inline mem_fun_t<S, T> mem_fun(S (T::*f)()) {

```

```

    Return mem_fun_t<S, T>(f);

```

```

}

```

```

Template <class S, class T>

```

```

Inline const_mem_fun_t<S, T> mem_fun(S (T::*f)() const) {

```

```

    Return const_mem_fun_t<S, T>(f);

```

```

}

```

```

Template <class S, class T>

```

```

Inline mem_fun_ref_t<S, T> mem_fun(S (T::*f)()) {

```

```

    Return mem_fun_t<S, T>(f);

```

```

}

```

```

Template <class S, class T>

```

```

Inline const_mem_fun_ref_t<S, T> mem_fun(S (T::*f)() const) {

```

```

    Return const_mem_fun_ref_t<S, T>(f);

```

```

}

```

```

Template <class S, class T, class A>

```

```

Inline mem_fun1_t<S, T, A> mem_fun1(S (T::*f)(A)) {

```

```

    Return mem_fun1_t<S, T, A>(f);

```

```

}

```

```

Template <class S, class T, class A>

```

```
Inline const_mem_fun1_t<S, T, A> mem_fun1(S (T::*f)(A) const){  
    Return const_mem_fun1_t<S, T, A>(f);  
}
```

```
Template <class S, class T, class A>  
Inline mem_fun1_ref_t<S, T, A> mem_fun1(S (T::*f)(A)){  
    Return mem_fun1_ref_t<S, T, A>(f);  
}
```

```
Template <class S, class T, class A>  
Inline mem_fun1_ref_t<S, T, A> mem_fun1(S (T::*f)(A) const){  
    Return const_mem_fun1_ref_t<S, T, A>(f);  
}
```