Chapter 1. STL 概论与版本简介

1.1.STL 概论

1.2.STL 六大组件 功能与运用、

1、六大组件

1) 容器
2) 算法
3) 迭代器
4) 仿函数
5) 配接器
6) 配置器

1.3.GUN 源代码开放精神

1.4.HP 实现版本

1.5.P.J.Plauger 实现版本

1.6.Rouge Wave 实现版本

1.7.STLport 实现版本

1.8.SGI STL 实现版本

1.8.1. GUN C++ headers 文件分布(按字母排序)

1.8.2. SGI STL 文件分布与简介

1.8.3. SGI STL 的编译器组态设置(configuration)

1、__STL_STATIC_TEMPLATE_MEMBER_BUG

1) 组态 3
2) 是否允许类拥有静态成员

Chapter 2. 空间配置器

2.1. 空间配置器的标准接口

1、根据 STL 规范，以下是 allocator 的必要接口

1) allocator::value_type
2) allocator::pointer
3) allocator::const_pointer
4) allocator::reference
5) allocator::const_reference
6) allocator::size_type
7) allocator::difference_type
8) allocator::rebind：一个嵌套的 class template，class rebind<U>拥有唯一成员 other，这是一个 typedef，代表 allocator<U>
9) allocator::allocator()
10) allocator::allocator(const allocator&)
11) template<class U> allocator::allocator(const allocator<U>&)
12) allocator::~allocator()
13) pointer allocator::address(reference x) const//这里也有 const 么
14) const_pointer allocator::address(const_reference x) const
15) pointer allocator::allocate(size_type n,const void*=0)
16) void allocator:deallocate(pointer p,size_type n)
17) size_type allocator::max_size() const
18) void allocator::construct(pointer p,const T&x)
19) void allocator::destroy(pointer x)

2.2. 具备次配置力(sub-allocation)的 SGI 空间配置器

1、SGI STL 的配置器与标准规范不同，其名称是 alloc 而非 allocator，而且不接受任何参数

2.2.1. SGI 标准的空间配置器

1、SGI 也定义有一个符合部分标准、名为 allocator 的配置器，但 SGI 从未使用过它，也不建议使用，主要原因是其效率不佳，因为只是把 C++的::operator new 和::operator delete 做一层封装而已

2.2.2. SGI 特殊的空间配置器，std::alloc

2.2.2.1. new、::operator new、placement new 的区别

1、new 和 delete 操作符(又可称为 new operator/delete operator)

- 它们是对堆中的内存进行申请和释放，new operator 与 delete operator 的行为是不能够也不应该被改变，这是 C++标准作出的承诺
- new 操作符实际上是执行如下 3 个过程：
  1) 调用::operator new 分配内存
  2) 调用构造函数生成类对象

3) 返回相应指针

- 要实现不同的内存分配行为，需要重载::operator new，而不是 new 和 delete

2、operator new

- operator new 与 operator delete 与 C 语言中的 malloc 与 free 对应，只负责分配及释放空间，与其他可重载操作符(例如 operator +)一样，是可以重载的
- 不能在全局对原型为 void ::operator new(size_t size)这个原型进行重载
- 一般只能在类中进行重载
  1) 重载时，返回类型必须声明为 void*
  2) 重载时，第一个参数类型必须为表达要求分配空间的大小(字节)，类型为 size_t
  3) 重载时，可以带其它参数
- 如果类中没有重载::operator new，那么调用的就是全局的::operator new 来完成堆的分配
- 同理，::operator new[]、::operator delete、::operator delete[]也是可以重载的
- 一般你重载了其中一个，那么最好把其余三个都重载一遍

3、::placement new

void *operator new(size_t, void * p) throw() { return p; }

- placement new 是重载 operator new 的一个标准、全局的版本，它不能够被自定义的版本代替，即不能重载
- placement new 的执行忽略了 size_t 参数，只返还第二个参数。其结果是允许用户把一个对象放到一个特定的地方，达到调用构造函数的效果。和其他普通的 new 不同的是，它在括号里多了另外一个参数(指向已分配内存的指针)
- 如果你想在已经分配的内存中创建一个对象，使用 new 是不行的。也就是说::placement new 允许你在一个已经分配好的内存中(栈或堆中)构造一个新的对象。原型中 void*p 实际上就是指向一个已经分配好的内存缓冲区的的首地址
-

4、我们知道使用 new 操作符分配内存需要在堆中查找足够大的剩余空间，这个操作速度是很慢的，而且有可能出现无法分配内存的异常(空间不够)。::placement new 就可以解决这个问题。我们构造对象都是在一个预先准备好了的内存缓冲区中进行，不需要查找内存，内存分配的时间是常数；而且不会出现在程序运行中途出现内存不足的异常。所以，::placement new 非常适合那些对时间要求比较高，长时间运行不希望被打断的应用程序

5、::placement new 如何使用

new (p) T();//其中 p 是一块已经分配但未初始化的内存

::__PLACEMENT_NEW_INLINE new(a) T();//PJ 版本的显式调用

2.2.3. 构造和析构基本工具：construct, destroy

1、下面给出部分源码(<stl_construct.h>)(已核对)

#include<new.h>//现在该头文件内容与原来的头文件内容有很大出入

```
template <class T>
inline void destroy(T* pointer) {//该方法会调用指定的析构函数
        pointer->~T();
}

template <class T1, class T2>
inline void construct(T1* p, const T2& value) {
        new (p) T1(value);//使用了 placement new
}

template <class ForwardIterator>
inline void
__destroy_aux(ForwardIterator first, ForwardIterator last, __false_type) {
        for ( ; first < last; ++first)
                destroy(&*first);
}
//被 destroy 的类型，其析构函数不可忽略，不是 trivial 的析构函数，因此必须老老实实得调用析构函数

template <class ForwardIterator>
inline void __destroy_aux(ForwardIterator, ForwardIterator, __true_type) {}
//被 destroy 的类型，其析构函数可以忽略，是 trivial 的析构函数，因此直接忽略不执行析构函数，提高效率

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());//根据 trivial_destructor()返回值选择合适的重载版本，返回的是一个对象，用于静态重载分派
}

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

//以下两个版本特例化 char 类型和 wchar_t 类型的 destroy 方法
inline void destroy(char*, char*) {}
inline void destroy(wchar_t*, wchar_t*) {}
```

2.2.3.1. __type_traits<>

1、在 STL 中为了提供通用的操作而又不损失效率，我们用到了一种特殊的技巧，叫 traits 编程技巧。具体的来说，traits 就是。通过定义一些结构体或类，并利用

模板类特化和偏特化的能力，给类型赋予一些特性，这些特性根据类型的不同而异。在程序设计中可以使用这些 traits 来判断一个类型的一些特性，引发 C++的函数重载机制，实现同一种操作因类型不同而异的效果。traits 的编程技巧极度弥补了 C++语言的不足

- 这里的不足是：C++本身并不直接支持对"指针所指之物"的类型判断，也不支持对"对象析构函数是否为 trivial"的判断

2、__type_traits 提供了一种机制，允许针对不同的型别属性，在编译时期完成函数派送决定(如果用 bool 值，那么无法根据不同类型调用不同的重载函数，如果返回一个对象，那么可以根据对象的类型进行重载)。这对于撰写 template 很有帮助。例如当我们对一个型别未知的数组进行 copy 时，如果我们事先知道该元素型别的构造函数是否是不重要的，我们可能可以使用 memcpy 或是 memmove 等函数快速处理

3、源码说明(<type_traits.h>)(已核对)

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

template <class _Tp>
struct __type_traits {
    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;
};
```

- 在没有模板特例化的情况下，任意类型的默认构造函数，拷贝构造函数，赋值运算符，析构函数都被标记为非 trivial，即不可忽略
- 如果需要忽略某个类型的上述某几个函数，那么需要进行特例化，如下

```
template <>
struct __type_traits<int> {
    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;
};
```

2.2.4. 空间的配置与释放，std::alloc

1、对象构造前的空间配置和对象析构后的空间释放，由<stl_alloc.h>负责，SGI 对此的设计哲学如下

1) 向 system heap 要求空间
2) 考虑多线程(multi-threads)的状态
3) 考虑内存不足时的应变策略

4) 考虑多"小型区块"可能造成的内存碎片(fragment)问题

2、C++的内存配置基本操作是::operator new()，内存释放基本操作是::operator delete()，这两个全局函数相当于 C 的 malloc()和 free()函数。SGI 正是以 malloc()和 free()完成内存的配置与释放

3、考虑到小型区块可能造成的内存破碎问题，SGI 设计了双层级配置器

1) 第一级配置器直接使用 malloc()和 free()

2) 第二级配置器则视情况采用不同的策略

- 当区块超过 128bytes 时，视为足够大，调用第一级配置器
- 当区块小于 128bytes 时，视为过小，为了降低额外负担，采用复杂的 memory pool 整理方式
- 整个设计究竟只开放第一级配置器，或是同时开放第二级配置器(即 alloc 是第一级配置器还是两级配置器)，取决于__USE_MALLOC 是否被定义(SGI STL 并未定义__USE_MALLOC)

4、无论 alloc 被定义为第一级或第二级配置器，SGI 还为它再包装一个接口，源码如下(<stl_alloc.h>)(已核对)

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

2.2.5. 第一级配置器 __malloc_alloc_template 剖析

1、源代码(<stl_alloc.h>)(已核对)

```
#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

template <int inst>
class __malloc_alloc_template {
```

```
private:

    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
        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);
        if (0 == result) result = oom_realloc(p, new_sz);
        return result;
    }

    //以下仿真 C++的 set_new_handler()，换句话说，你可以通过它指定自己的 out-of-memoty handler
    static void (* set_malloc_handler(void (*f)()))(){
        void (* old)() = __malloc_alloc_oom_handler;//旧的处理函数
        __malloc_alloc_oom_handler = f;//赋值新的处理函数
        return(old);//返回旧处理函数
    }
```

- static void (* set_malloc_handler(void (*f)()))()解析
  - 找到名字 set_malloc_handler
  - set_malloc_handler 右边有形参列表，因此它是一个函数
  - 返回类型首先是一个指针，指向的类型是函数，为 void ()
  - set_malloc_handler 形参列表中的形参为 f，f 首先是一个指针，指向的是 void ()的函数

```
};

//处理器初始化为空指针
```

```
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，如果该__malloc_alloc_oom_handler 未被设定，即 static void (*set_malloc_handler(void (*f)()))()未被客户端调用，那么会直接抛出异常
        my_malloc_handler = __malloc_alloc_oom_handler;
        if (0 == my_malloc_handler) { __THROW_BAD_ALLOC; }
        (*my_malloc_handler)();//调用处理，企图释放内存
        result = malloc(n);//再次尝试分配内存
        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，如果该__malloc_alloc_oom_handler 未被设定，即 static void (*set_malloc_handler(void (*f)()))()未被客户端调用，那么会直接抛出异常
        my_malloc_handler = __malloc_alloc_oom_handler;
        if (0 == my_malloc_handler) { __THROW_BAD_ALLOC; }
        (*my_malloc_handler)();
        result = realloc(p, n);
        if (result) return(result);
    }
}

typedef __malloc_alloc_template<0> malloc_alloc;
```

2、第一级配置器以 malloc()、free()、realloc()等 C 函数执行实际的内存配置、释放、重配置操作，并实现处类似 C++ new-handler 的机制(它不能直接运用 C++new-handler 机制，因为它并非使用::operator new 来配置内存)

3、所谓 C++ new handler 机制是，你可以要求系统在内存配置需求无法被满足时，调用一个你所指定的函数。也就是说，一旦::operator new 无法完成任务，在丢出 std::bad_alloc 异常状态之前，会先调用客户端指定的处理例程，该处理例程通常称为 new-handler。new-handler 解决内存不足的做法有特定的模式

4、SGI 以 malloc 而非::operator new 来配置内存，因此 SGI 不能直接使用 C++的 set_new_handler()，必须仿真一个类似的 set_malloc_handler()

2.2.6. 第二级配置器 __default_alloc_template 剖析

1、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，配置时的额外负担(overhead)也是一个大问题。额外负担永远无法避免，毕竟系统要考这多出来的空间来管理内存

2、SGI 第二级配置器的做法是

- 如果区块足够大，超过 128bytes，就移交第一级配置器处理
- 当区块小于 128bytes，则以内存池(memory pool)管理，此法又称层次配置：
  - 每次配置一大块内存，并维护对应之自由链表(free-list)
  - 若下次再有相同大小的内存需求，就直接从 free-lists 中拔出
  - 如果客户端释还小额区块，就由配置器回收到 free-lists 中

3、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上调至 8 的倍数(例如要求 30bytes，主动调整为 32bytes)，并维护 16 个 free-lists，各自管理大小分别为 8，16,24,32,....,128bytes 的小额区块。free-lists 的结构如下

```
union obj{
    union obj * free_list_link;
    char client_data[1];
}
```

- 为了维护链表，每个节点需要额外的指针，这会造成另一种额外负担
- 上述 obj 采用 union，由于 union 之故，obj 可被视为一个指针，指向相同形式的另一个 obj，obj 又可被视为一个指针，指向实际区块

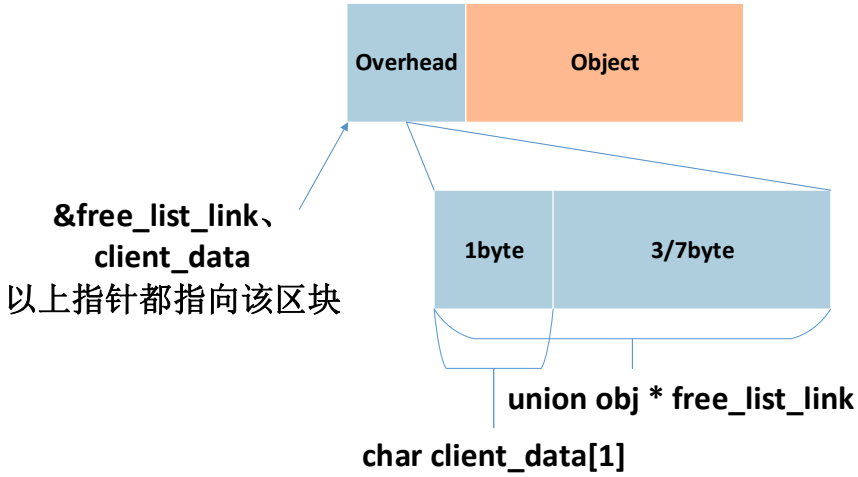
4、第二级配置器源码(<stl_alloc.h>)(已核对)

```
enum {__ALIGN = 8};
enum {__MAX_BYTES = 128};
enum {__NFREELISTS = __MAX_BYTES/__ALIGN};

template <bool threads, int inst>
class __default_alloc_template {

private:
    //向上补足至 8 的倍数，例如 7 返回 8，30 返回 32 等
    //__ALIGN-1=7 也就是 00000...000111，再取反就是 1111...111000
    static size_t ROUND_UP(size_t bytes) {
        return (((bytes) + __ALIGN-1) & ~(__ALIGN - 1));
    }
private:
    //obj 详解
      1) free_list_link 成员是个指针，即该 obj 对象中存放了一个指向
         obj 对象的指针，即存的是指针本身(32 位 4 字节，64 位 8 字节)
      2) client_data[1]是个长度为 1 的 char 数组，即存的是数组本身，
         在 obj 对象中占用了 1byte，当我们在使用 client_data 时，
         client_data 会退化为指针，指向了 client_data 所占据的内存空
```

间，也就是 obj 对象的起始地址，而这个退化的指针据我理解是个临时量

```
union obj {
    union obj * free_list_link;//自由链表的下一个节点
    char client_data[1];// client_data 是个 char*指针，表示当前节点指向的实际内存空间
};
```

//obj 内存模型详见下图，其中 overhad 还包括了一些其他系统负载，比如记录内存区域的大小(其中 Overhead 部分是否在对象初始化后始终作为系统开销，详见 2.2.9 详解部分)



1) free_list_link 是个指针，而指针的地址才是 obj 所占内存的地址
2) client_data 数组，因此它的地址就是 obj 所占内存的地址
   - 综上&_obj(假设_obj 是 obj 的对象)，&free_list_link，client_data 指向同一区域

```
private:
    static obj * volatile free_list[__NFREELISTS];
    // free_list 是个大小为 NFREELISTS 的数组，数组存放的元素是 obj *，且元素为 volatile
    static size_t FREELIST_INDEX(size_t bytes) {
        return (((bytes) + __ALIGN-1)/__ALIGN - 1);
    }

    static void *refill(size_t n);
    // Allocates a chunk for nobjs of size "size".  nobjs may be reduced
    // if it is inconvenient to allocate the requested number
    static char *chunk_alloc(size_t size, int &nobjs);

    // Chunk allocation state.
    static char *start_free;//内存起始位置，只在 chunk_alloc()中变化
    static char *end_free;//内存结束位置，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);
};

typedef __default_alloc_template<__NODE_ALLOCATOR_THREADS, 0> alloc;
template <bool threads, int inst>
char *__default_alloc_template<threads, inst>::start_free = 0;

template <bool threads, int inst>
char *__default_alloc_template<threads, inst>::end_free = 0;

template <bool threads, int inst>
size_t __default_alloc_template<threads, inst>::heap_size = 0;

template <bool threads, int inst>
__default_alloc_template<threads, inst>::obj * volatile
__default_alloc_template<threads, inst> ::free_list[__NFREELISTS] =
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, };
```

2.2.6.1. union

1、什么是联合

- "联合"是一种特殊的类，也是一种构造类型的数据结构
- 在一个"联合"内可以定义多种不同的数据类型，一个被说明为该"联合"类型的变量中，允许装入该"联合"所定义的任何一种数据，这些数据共享同一段内存，已达到节省空间的目的(还有一个节省空间的类型：位域)。这是一个非常特殊的地方，也是联合的特征
- 另外，同 struct 一样，联合默认访问权限也是公有的，并且，也具有成员函数

2、联合与结构的区别

- "联合"与"结构"有一些相似之处。但两者有本质上的不同
- 在结构中各成员有各自的内存空间，一个结构变量的总长度是各成员长度之和(空结构除外，同时不考虑边界调整)
- 而在"联合"中，各成员共享一段内存空间，一个联合变量的长度等于各成员中最长的长度
- 应该说明的是，这里所谓的共享不是指把多个成员同时装入一个联合变量内，而是指该联合变量可被赋予任一成员值，但每次只能赋一种值，赋入新值则冲去旧值。

3、联合里面的东西共享内存，所以静态、引用都不能用，因为他们不可能共享内存。

4、因为联合里不允许存放带有构造函数、析够函数、复制拷贝操作符等的类，因为他们共享内存，编译器无法保证这些对象不被破坏，也无法保证离开时调用

析够函数
5、这种技巧在强型语言如 Java 中行不通，但是在非强型语言如 C++中十分普遍

2.2.6.2. 为什么用 char*来表示指向内存地址的指针

1、内存以 byte 为单位进行分配，而 char 在 C++中，在任何编译器(16 位、32 位、64 位)中所占用的内存都是 1byte，方便进行偏移运算

2.2.7. 空间配置函数 allocate()

1、身为一个配置器，__default_alloc_template 拥有配置器的标准接口函数 allocate()
2、__default_alloc_template::allocate()源码(<stl_alloc.h>)(已核对)

```
static void * allocate(size_t n) {
    obj * volatile * my_free_list;
    obj * result;

    //大于 128 就调用第一级配置器
    if (n > (size_t) __MAX_BYTES) {
        return(malloc_alloc::allocate(n));
    }

    //根据字节数，找到合适的 free list
    my_free_list = free_list + FREELIST_INDEX(n);
    // my_free_list 是一个二维指针 obj**，指向当前区块的链表头指针，因此下面的 result 是 obj*，即指向的是 obj 对象，而函数在 result 非空时返回的就是 result，即指向 obj 对象的指针，客户会在该 obj 所占的内存空间上构造元素
    result = *my_free_list;
    if (result == 0) {//如果对应的 free_list 没有可用区块，那么调用 refill
        void *r = refill(ROUND_UP(n));//该函数下节详述
        return r;
    }
    *my_free_list = result -> free_list_link;//将当前 result 指针的下一个节点作为该区块的头指针
    return (result);
};
```

2.2.8. 空间释放函数 deallocate()

1、身为一个配置器，__default_alloc_template 拥有配置器的标准接口函数 deallocate()
2、__default_alloc_template::deallocate()源码(<stl_alloc.h>)(已核对)

```
static void deallocate(void *p, size_t n) {
    obj *q = (obj *)p;
    obj * volatile * my_free_list;
```

```
        //大于 128 就调用第一级配置器
        if (n > (size_t) __MAX_BYTES) {
            malloc_alloc::deallocate(p, n);
            return;
        }

        //寻找对应的 free list，同理，my_free_list 为 obj**类型，指向对应区块的链表头指针，并将 q(即被释放的地址)插入链表到头部
        my_free_list = free_list + FREELIST_INDEX(n);
        q -> free_list_link = *my_free_list;
        *my_free_list = q;

    }
```

2.2.9. 重新填充 free lists

1、当 free list 中没有可用区块时，就调用 refill()，准备为 free list 重新填充空间。新的空间将取自内存池(由 chunk_alloc()完成，默认取得 20 个新节点，万一内存池不够用，获得的节点数可能小于 20)

2、 __default_alloc_template::refill 源码(<stl_alloc.h>)(已核对)

```
    template <bool threads, int inst>
    void* __default_alloc_template<threads, inst>::refill(size_t n) {
        int nobjs = 20;
        //调用 chunk_alloc()，尝试取得 nobjs 个区块作为 free list 的新节点
        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，这个区块就分配给调用者，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 free list
        result = (obj *)chunk;
        //以下引导 free list 指向新配置的空间(取自内存池)
        *my_free_list = next_obj = (obj *)(chunk + n);
        //以下将 free list 各节点串接起来
        for (i = 1; ; i++) {//从 1 开始，因为第 0 个返回给客户端，剩余的才插入 free list
            current_obj = next_obj;
            //关键语句详解
              1)  由于 char 为一个字节，因此 char*指针 next_obj 加上 1 相当
```

于移动一个字节

2) 而分配的对象是 n 字节的，因此加 n(从这里可以看出，next_obj 所指向的内存区域大小是 n+obj 对象的大小，因此 obj 对象所占的空间至始至终作为 overhead，没有被用户对象所重用)

3) 然后转型为联合对象 obj 的指针

```
            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. 内存池(memory pool)

1、_default_alloc_template::chunk_alloc 源码(<stl_alloc.h>)(已核对)

```
//假设 size 已经上调至 8 的倍数
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;//内存池剩余空间
    if (bytes_left >= total_bytes) {
        //内存剩余空间完全满足需求
        result = start_free;
        start_free += total_bytes;
        return(result);
    } else if (bytes_left >= size) {
      //内存池剩余空间不能完全满足需求，但足够供应一个(含)以上区块
        nobjs = bytes_left/size;
        total_bytes = size * nobjs;
        result = start_free;
        start_free += total_bytes;
        return(result);
    } else {//内存池剩余空间连一个区块的大小都无法提供
        //新内存量的大小为需求量的两倍，加上一个随着配置次数增加而越来越大的附加量(每次分配的内存必然是 8 的倍数，每次取的内存也是 8 的倍数，因此①处这种情况不会发生)
        size_t bytes_to_get = 2 * total_bytes + ROUND_UP(heap_size >> 4);
        //以下尝试让内存池中的残余零头还有利用价值
```

```
if (bytes_left > 0) {
        //内存池还有一些零头，先配给适当的 free list
        //先寻找适当的 free list(①这里奇怪，因为 FREELIST_INDEX 本身含有向上扩充到 8byte 倍数的含义，比如剩余 1byte，那么会将这个 1byte 的区块插入到自由链表的第一个链表中，也就是 8byte 的表中，这样不是出问题了么?可能 bytes_left 永远是 8 的倍数，因此这种情况不会发生)
        obj * volatile * my_free_list =
        free_list + FREELIST_INDEX(bytes_left);
        //调整 free list，将内存池中的残余空间编入
        ((obj *)start_free) -> free_list_link = *my_free_list;
        *my_free_list = (obj *)start_free;
}

//配置 heap 空间，用来补充内存池
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;
        //试着检验我们手上拥有的东西，这不会造成伤害，我们不打算配置较小的区块，因为那在多进程(multi-process)机器上容易导致灾难
        //以下搜索适当的 free list，所谓适当是指"尚未拥有区块，切区块足够大"的 free list，(例如需要 7byte 的空间，本来会从 8byte 的自由链表中查找，但现在会往更高的自由链表中查找，比方说 128byte 的表中还有，那么也会将其分配给本次需要 7byte 的客户)
        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 内尚有未用区块
                        //调整 free list 以释放区块
                        *my_free_list = p -> free_list_link;
                        start_free = (char *)p;
                        end_free = start_free + i;
                        //递归调用自己，为了修正 nobjs(此时只是将该链表表头抽出，然后修改内存池首位地址)
                        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(没有任何内存可用)
```

```
            start_free = (char *)malloc_alloc::allocate(bytes_to_get);
            //这将导致抛出异常，或内存不足的情况得以改善
        }
        //下面两句意思不懂
        heap_size += bytes_to_get;
        end_free = start_free + bytes_to_get;
        //递归调用自己，为了修正 nobjs
        return(chunk_alloc(size, nobjs));
    }
}
```

2.3. 内存处理基本工具（Important）

1、STL 定义五个全局函数，作用于未初始化空间上(已经分配的内存，但是内存区域尚未初始化)

1) construct()
2) destroy()
3) uninitialized_copy()
4) uninitialized_fill()
5) uninitialized_fill_n()

- 包含<memory>，定义于<stl_uninitialized>中

2.3.1. uninitialized_copy

1、uninitialized_copy()能够将内存的配置与对象的构造行为分开

- 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区域，则 uninitialized_copy 会使用 copy constructor，给身为输入来源的[first,last)范围内的每一个对象产生一份复制品，放进输出范围中
- 针对输入范围内的每一个迭代器 i，该函数会调用 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，"result+(i-first))"是个迭代器，因此需要解引用，然后取地址来获取指针

2、源码(<stl_uninitialized.h>)(已核对)

```
template <class InputIterator, class ForwardIterator>
ForwardIterator
uninitialized_copy(InputIterator first, InputIterator last,
                   ForwardIterator result) {
    return __uninitialized_copy(first, last, result, value_type(result));
}
```

- 迭代器 first 指向输入端的起始位置(闭)
- 迭代器 last 指向输入端的结束位置(开)
- 迭代器 result 指向输出端(欲初始化空间)的起始处

```
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());
}
```

- 这个函数进行的逻辑是，首先萃取出迭代器 result 的 value type，然后判断该型别是否为 POD 型别
  - POD 指 Plain Old Data，也就是标量类型，或传统的 C struct 类型
  - POD 型别必然拥有 trivial ctor/dtor/copy/assignment 函数，因此可以对 POD 型别采用最有效的初值填写手法
  - 对 non-POD 型别采取最保险安全的手法

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

template <class InputIterator, class ForwardIterator>
ForwardIterator
__uninitialized_copy_aux(InputIterator first, InputIterator last,
                         ForwardIterator result,
                         __false_type) {
    ForwardIterator cur = result;
    for ( ; first != last; ++first, ++cur)
        construct(&*cur, *first);//必须一个一个构造，无法批量进行
    return cur;
}
```

3、针对 char*和 wchar_t*两种型别，可以采用最具效率的做法 memmove(直接移动内存内容)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本

- char：一个字节，只能表示 256 个字符
- wchar_t：2 或 4 字节，用于存储其他字符，例如中文等，unicode 编码

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

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);
```

}

4、如果需要实现一个容器，uninitialized_copy()这样的函数会为你带来很大帮助，因为容器的全区间构造函数，通常以两个步骤完成

1) 配置内存区块，足以包含范围内的所有元素
2) 使用 uninitialized_copy()，在该内存上构造元素

5、C++标准规格书要求 uninitialized_copy()具有"commit or rollback"，意为要么"构造出所有必要元素"，要么"不构造任何东西"

2.3.2. uninitialized_fill

1、uninitialized_fill()能够使我们将内存配置与对象的构造行为分离开

- 如果[first,last)范围内的每个迭代器都指向未初始化的内存，那么 uninitialized_fill()会在该范围内产生 x 的复制品，即调用 construct(&*i,x)

2、源码(<stl_uninitialized.h>)(已核对)

```
template <class ForwardIterator, class T>
inline void uninitialized_fill(ForwardIterator first, ForwardIterator last,
                               const T& x) {
    __uninitialized_fill(first, last, x, value_type(first));
}
```

- 迭代器 first 指向输出端(欲初始化空间)的起始处(闭)
- 迭代器 last 指向输出端(预初始化空间)的结束处(开)
- x：表示初值

```
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());
}
```

- 这个函数进行的逻辑是，首先萃取出迭代器 result 的 value type，然后判断该型别是否为 POD 型别
  - POD 指 Plain Old Data，也就是标量类型，或传统的 C struct 类型
  - POD 型别必然拥有 trivial ctor/dtor/copy/assignment 函数，因此可以对 POD 型别采用最有效的初值填写手法
  - 对 non-POD 型别采取最保险安全的手法

```
template <class ForwardIterator, class T>
inline void
__uninitialized_fill_aux(ForwardIterator first, ForwardIterator last,
                         const T& x, __true_type){
    fill(first, last, x);//调用 STL 算法 fill()
}

template <class ForwardIterator, class T>
void
```

```
__uninitialized_fill_aux(ForwardIterator first, ForwardIterator last,
                         const T& x, __false_type){
    ForwardIterator cur = first;
    for ( ; cur != last; ++cur)
        construct(&*cur, x);//必须一个一个元素构造，无法批量进行
}
```

3、与 uninitialized_copy()一样，uninitialized_fill()必须具备 commit or rollback 语义，要么产生所有必要元素，要么不产生任何元素

2.3.3. uninitialized_fill_n

1、uninitialized_fill_n()能够使我们将内存分配与对象构造行为分离开来

- 它会为指定范围内的所有元素设定相同的初值
- 如果[first,first+n)范围内的每一个迭代器都指向未初始化的内存，那么 uninitialized_fill_n()会调用 copy constructor，在该范围内产生 x 的复制品，即调用 construct(&*i,x)，在对应位置处产生 x 的复制品

2、源码(<stl_uninitialized.h>)(已核对)

```
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));
}
```

- 迭代器 first 指向预初始化空间的起始处
- n 表示预初始化空间的大小
- x 表示初值

```
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());
}
```

- 函数进行的逻辑是：先萃取出迭代器 first 的 value type，然后判断该类型是否为 POD 类型
  - POD 指 Plain Old Data，也就是标量类型，或传统的 C struct 类型
  - POD 型别必然拥有 trivial ctor/dtor/copy/assignment 函数，因此可以对 POD 型别采用最有效的初值填写手法
  - 对 non-POD 型别采取最保险安全的手法

```
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);//交由高阶函数执行
```

```
}

template <class ForwardIterator, class Size, class T>
ForwardIterator
__uninitialized_fill_n_aux(ForwardIterator first, Size n,
                           const T& x, __false_type) {
    ForwardIterator cur = first;
    for ( ; n > 0; --n, ++cur)
        construct(&*cur, x);
    return cur;
}
```

3、uninitialized_fill_n()也具有 commit or rollback 的语义，要么产生所有必要元素，要么不产生任何元素

Chapter 3. 迭代器(iterators)概念与 traits 编程技法

1、iterator 模式定义：提供一种方法，使之能够依序访问某个聚合物(容器)所含的各个元素，而又无需暴露该聚合物的内部表述方式

3.1. 迭代器设计思维---STL 关键所在

1、STL 的中心思想：将数据容器与算法分开，彼此独立设计，最后再将它们撮合在一起

3.2. 迭代器(iterator)是一种 smartpointer

1、迭代器是一种行为类似指针的对象，而指针的各种行为中最常见的也最重要的便是内容提领(dereference)和成员访问(member access)。因此迭代器最重要的工作就是对 operator *和 operator->进行重载
2、要针对某一个特定容器额外实现一个迭代器，就必须对该容器有充分的了解，既然这无法避免，干脆就把迭代器开发工作交给容器设计者，这也就是每一种 STL 容器都提供专属迭代器的原因

3.3. 迭代器相应型别

1、在运用迭代器时，可能会用到其相应型别(associated type)。迭代器所指之物的型别便是其一
2、获取迭代器所指类型型别的解决办法：利用 function template 的参数推导(argument deducation)机制

```
void func(I iter){
    func_impl(iter,*iter);
}

template <class T,class T>
void func_impl(I iter, T t){
    T tmp;
    ...
}
```

3、迭代器相应型别，常用的有 5 中，并不是所有都能通过 template 参数推导机制来取得

3.4. Traits 编程技法---STL 源代码门钥

1、迭代器所指对象的型别，称为该迭代器的 value type，上一节的参数型别推导技巧可用于 value type，但也无法用于所有场景，例如，value type 必须用于函数的传回值，就没有办法了
2、解决方法：声明内嵌型别

```
template <class T>
```

```
struct MyIter{
    typedef T value_type;
    T* ptr;
    MyIter(T* p=0): ptr(p) { }
    T& operator*() const { return *ptr; }
    ...
};

template <class I>
typename I::value_type//函数返回值
func(I ite){
    return *ite;
}
```

- func()返回型别必须加上 typename 关键词，因为 T 是一个 template 参数，在它被编译器具现化之前，编译器对 T 一无所悉，换句话说，编译器此时并不知道 MyIter<T>::value_type 代表的是一个型别或是一个 member function 或者一个 data member
- 关键词 typename 用于告诉编译器这是一个型别，如此才能顺利通过编译

```
~~template <class I>~~
~~*I function(I ite){~~//参数推导机制不能推导返回类型!!!
    ~~return *ite;~~
~~}~~
```

3、上述解决方法看起来不错，但是有个隐晦陷阱：并不是所有迭代器都是 class type，原生指针就不是，如果不是 class type 就无法为其定义内嵌型别。但 STL(以及整个泛型思维)绝对必须接受原生指针作为一种迭代器

4、template partial specialization

1) Partial specialization 的意义：如果 class template 拥有一个以上的 template 参数，我们可以针对其中某个(或数个，但非全部)template 参数进行特化工作。即我们可以在泛化设计中提供一个特化版本(也就是将特化版本中的某些 template 参数赋予明确的指定或者进行进一步的条件限定)

```
template<typename T>
class C{...};//这个版本接受 T 为任何类型

template<typename T>
class C<T*> {...};//这个版本仅适用于 T 为原生指针的情况
```

2) 有了 partial specialization，就可以解决"内嵌型别"无法解决的问题

5、iterator_traits

1) 下面这个 class template 专门用来"萃取"迭代器特性，而 value type 正是迭代器的特性之一

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

```
    ...
}
```

- 所谓 traits，其意义是：如果 I 定义有自己的 value type，那么通过这个 traits 的作用，萃取出来的 value_type 就是 I::value_type

2) 如果 I 定义有自己的 value_type，先前那个 func()可以进行如下改写

```
template<class T>
typename iterator_traits<I>::value_type
func(I ite){
    return *ite;
}
```

- 多了一层间接性，带来什么好处呢?好处是 traits 可以拥有特化版本

3) 我们令 iterator_traits 拥有一个 partial specializations 如下

```
template<class T>
struct iterator_traits<T*>{
    typedef T value_type;
};
```

- 于是原生指针 int*虽然不是一种 class Type，也可通过 traits 取其中 values type，这就解决了先前的问题

4) 但是对于指向常量对象的指针例如 const int*，通过上述萃取得到的是 const int 而非 int，于是可以再定义一个偏特化版本

```
template<class T>
struct iterator_itraits<const T*>{
    typedef T value_type;
}
```

5) 若要特性萃取机 traits 有效运作，每一个迭代器必须遵循约定，自行以内嵌型别定义(nested typedef)的方式定义出相应型别，这是一个约定，不遵循这个约定，就无法兼容 STL

6、常用的迭代器相应型别有五种

1) value type
2) difference type
3) pointer
4) reference
5) iterator
6) category

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

3.4.1. 迭代器相应型别之一：value type

1、value type 是指迭代器所指对象的型别，任何一个打算与 STL 算法有完美搭配

的 class，都应该定义自己的 value type 内嵌型别

3.4.2. 迭代器相应型别之二：difference type

1、difference type 用来表示两个迭代器之间的距离，因此它也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最大容量

2、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

3、针对相应型别 difference type，traits 的如下两个(针对原生指针而写的)特化版本与，以 C++内建的 ptrdiff_t(定义于<cstddef>)作为原生指针的 difference type

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

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

3.4.3. 迭代器相应型别之三：reference type

1、从迭代器所指之物内容是否允许改变的角度来看，迭代器分为两种：

1) 不允许改变"所指对象内容"，称为 constant iterators
2) 允许改变"所指对象之内容"，称为 mutable iterators

2、当我们对一个 mutable iterators 进行提领操作时，获得的不应该时一个右值(rvalue)，应该是一个左值(lvalue)，因为右值不允许赋值操作

3、在 C++中，函数如果要传回左值，都是以 by reference 的方式进行，所以

1) 当 p 是个 mutable iterators 时，如果其 value type 是 T，那么*p 的型别不应该时 T，应该是 T&
2) 当 p 是个 constant iterators 时，其 value type 是 T，那么*p 的型别不应该时 const T，而应该时 const T&

3.4.4. 迭代器相应型别之四：point type

1、pointers 和 references 在 C++中有非常密切的关联，如果传回一个左值，令它代表 p 所指之物是可能的，那么传回一个左值，令它代表 p 所指之物的地址也一定可以，也就是说，我们能够传回一个 pointer，指向迭代器所指之物

2、本章例子 ListIter class 的片段

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

- 如果 listIter->member 会如何
  - 如果 ptr 是个指针，那么调用(*ptr).member
  - 如果 ptr 指向了一个重载了->运算符的类的对象，那么继续调用 ptr.operator->().member

- 重复解析 ptr.operator->().member

3、对于原生指针，同样有两个偏特化版本

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

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

3.4.5. 迭代器相应型别之五：iterator_category

1、根据移动特性与施行操作，迭代器被分为五类：

1) Input Iterator:
2) Output Iterator:唯写(write only)
3) Forward Iterator:允许"写入型"算法
4) Bidirectional Iterator：可双向移动
5) Random Access Iterator：前四种迭代器都只供应一部分指针算数能力(前三种支持 operator++，第四种再加上 operator--，第五种则涵盖所有指针算数能力，包括 p+n,p-n,p[n],p1-p2,p1<p2

2、设计算法时，如果可能，尽量针对某种迭代器提供一个明确定义，并针对更强化的某种迭代器提供另一种定义，这样才能在不同情况下提供最大效率

3.4.5.1. 以 advanced () 为例

1、这是许多算法内部常用的一个函数，该函数有两个参数，迭代器 p 和数值 n；函数内部将 p 进行 n 次(前进 n 距离)

2、下面提供三份定义，一份针对 Input Iterator(Forward Iterator 和 Input Iterator 一样)，一份针对 Bidirectional Iterator，另一份针对 Random Access Iterator

```
template <class InputIterator, class Distance>
void advance_II(InputIterator& i,Distance n){
    while(n--) ++i;
}

template <class InputIterator, class Distance>
void advance_BI(InputIterator& i,Distance n){
    if(n>=0)
        while (n--) ++i;
    else
        while (n++) --i;
```

```
}

template <class InputIterator, class Distance>
void advance_RAI(InputIterator& i,Distance n){
    i+=n;
}
```

3、将三者合为一

```
template <class InputIterator, class Distance>
void advance (InputIterator& i,Distance n){
    if(is_random_access_iterator(i))
        advance_RAI(i,n);
    else if(is_bidirectional_iterator(i))
        advance_BI(i,n);
    else
        advance_II(i,n);
```

- 但是这样做，只有在执行时期才会决定使用哪个版本，会影响效率，最好能在编译期就选择正确版本，重载函数机制可以达成这个目标

4、考虑如下设计：如果 traits 有能力萃取出迭代器的种类，我们便可以利用这个"迭代器类型"相应型别作为 advanced()第三个参数，这个相应型别一定必须是一个 class type，而不能是数值(例如 int 或者 bool 之类运行时才能判断其值)的东西，因为编译器需要依赖(一个型别)来进行重载决议。下面定义五个 classes

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

- 这些 classes 只作为标记，所以不需要任何成员

5、重新设计__advance()(由于只在内部使用，所以函数名加上特定的前导符号)，并加上第三参数，使之形成重载

```
template <class InputIterator, class Distance>
inline void __advance(InputIterator& i, Distance n,
                      input_iterator_tag){
    while (n--) ++i;
}

template <class ForwardIterator, class Distance>
inline void __advance(ForwardIterator& i, Distance n,
                      forward_iterator_tag){
    __advance(i,n,input_iterator_tag());
}

template <class BidiectionalIterator, class Distance>
inline void __advance(BidiectionalIterator& i, Distance n,
                      bidirectional_iterator_tag){
```

```
        if(n>=0)
            while (n--) ++i;
        else
            while (n++) --i;

    }

    template <class RandomAccessIterator, class Distance>
    inline void __advance(RandomAccessIterator & i, Distance n,
                          random_access_iterator_tag){
        n+=i;
    }
```

6、还需要一个对外开放的上层控制接口，调用上述各个重载的__advance()，这个接口只需要两个参数，它将工作转发给__advance()时才会加上第三个参数：迭代器类型，自然交给 traits 机制

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

7、对于原生指针的偏特化版本

```
    template <class T>
    struct iterator_traits<T*>{
        ...
        typedef random_access_iterator_tag iterator_category;
    };

    template <class T>
    struct iterator_traits<const T*> {
        ...
        typedef random_access_iterator_tag iterator_category;
    };
```

8、任何一个迭代器，其类型永远应该落在"该迭代器所隶属之各种类型中最强化的那个"

9、advance、__advance 的模板参数名字不同(标记蓝色的部分)，这是 STL 算法的一个规则，以算法所能接受的最低阶迭代器类型，来为其迭代器型别参数命名

3.4.5.2. 消除"单纯传递调用的函数"

1、以 class 来定义迭代器的各种分类标签，不仅可以促成重载机制的成功运作，另一个好处是，通过继承，我们可以不必在写"单纯只做传递调用"的函数，例如前面的__advance()的 Forward Iterator 版本，这个版本其实可以删除

3.4.5.3. 以 distance() 为例

1、源码

```
    template <class InputIterator>
```

```
inline iterator_traits<InputIterator>::difference_type
__distance(InputIterator first, InputIterator last, input_iterator_tag) {
        iterator_traits<InputIterator>::difference_type n = 0;
        while (first != last) {
                ++first; ++n;
        }
        return n;
}

template <class RandomAccessIterator>
inline iterator_traits<RandomAccessIterator>::difference_type
__distance(RandomAccessIterator first, RandomAccessIterator last,
                random_access_iterator_tag) {
        return last - first;
}

template <class InputIterator>
inline iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last) {
        typedef typename iterator_traits<InputIterator>::iterator_category
        category;
        return __distance(first, last, category());
}

template <class InputIterator, class Distance>
inline void distance(InputIterator first, InputIterator last, Distance& n) {
        __distance(first, last, n, iterator_category(first));
}
```

2、当客户端调用 distance()并使用 Forward Iterators 或 Bidirectional Iterators 时都会传递调用 Input Iterator 版本的__distance()

3.5. std::iterator 的保证

1、为了符合规范，任何迭代器都应该提供五个内嵌相应型别，以利于 traits 萃取，否则便是自别于整个 STL 架构，可能无法与其他 STL 组件顺利搭配

2、STL 提供了一个 iterator class，只要继承它，就可以保证符合 STL 所需的规范，当然也可以不继承，但必须提供五个内嵌型别

```
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;
};
```

3.6. iterator 源代码完整重列

1、源码(<stl_iterator.h>)(已核对)

```
//五种迭代器类型
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
struct bidirectional_iterator_tag : public forward_iterator_tag {};
struct random_access_iterator_tag : public bidirectional_iterator_tag {};

//为避免写代码时错误，自行发开迭代器最好继承下面这个 std::iterator
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;
};

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;
};

//原生指针的偏特化版本
template <class T>
struct iterator_traits<T*> {
    typedef random_access_iterator_tag      iterator_category;
    typedef T                               value_type;
    typedef ptrdiff_t                       difference_type;
    typedef T*                              pointer;
    typedef T&                              reference;
```

```
};

//原生指针(pointer-to-const)的偏特化版本
template <class T>
struct iterator_traits<const T*> {
    typedef random_access_iterator_tag      iterator_category;
    typedef T                               value_type;
    typedef ptrdiff_t                       difference_type;
    typedef const T*                        pointer;
    typedef const T&                        reference;
};

//这个函数可以方便地决定某个迭代器的类型 category
template <class Iterator>
inline typename iterator_traits<Iterator>::iterator_category
iterator_category(const Iterator&) {
    typedef typename iterator_traits<Iterator>::iterator_category category;
    return category();
}
//这里为什么不返回指针类型：我的猜想是，category()本身是一个内容为空的 class，返回该类型的对象也占用不了多少内存

//这个函数可以方便地决定某个迭代器 distance type
template <class Iterator>
inline typename iterator_traits<Iterator>::difference_type*
distance_type(const Iterator&) {
    return static_cast<typename iterator_traits<Iterator>::difference_type*>(0);
}
//为什么要返回指针类型：我的猜想是，这里返回的如果是 difference_type 类型的对象，可能该对象占用的空间很大，返回一个该对象还需要构造一个该对象，会造成性能的降低。但是通过将 0 强制转型为该类型的指针，没有额外的开销，效率高

//这个函数可以方便地决定某个迭代器 value_type
template <class Iterator>
inline typename iterator_traits<Iterator>::value_type*
value_type(const Iterator&) {
    return static_cast<typename iterator_traits<Iterator>::value_type*>(0);
}
//为什么要返回指针类型：我的猜想是，如果返回的是 value_type 类型的对象，可能该对象占用的空间很大，返回一个该对象还需要构造一个该对象，会造成性能的降低。但是通过将 0 强制转型为该类型的指针，没有额外的开销，效率高
```

3.7.SGI STL 的私房菜：__type_traits

1、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

2、iterator_traits 负责萃取迭代器的特性，__type_traits 负责萃取型别特性。这个型别将会影响到我们是否可以在对这个型别进行构造、析构、拷贝、赋值等操作时，采用最有效的措施

3、源码<type_traits.h>

```
struct __true_type {};

struct __false_type {};
```

- 真假采用对象，而非数值(例如 int 或 bool)，因为我们想要利用其来进行参数推导，而编译器只有面对 class object 形式的参数才会做参数推导
- 这两个空白 classes 没有任何成员，不会带来额外负担，却又能标识真假

```
template <class type>
struct __type_traits {
    typedef __true_type        this_dummy_member_must_be_first;
    //不要移出这个成员，它通知"有能力自动将__type_traits 特化"的编译器，我们现在所看到的的这个__type_traits template 是特殊的。这是为了确保万一编译器也使用一个名为__type_traits 而其实与此处定义并无任何关联的 template 时，所有事情仍将顺利运作

    //以下条件应被遵守，因为编译器有可能自动为各型别产生专属的__type_traits 特化版本
      -你可以重新排列以下的成员顺序
      -你可以移出以下任何成员
      -绝对不可以将以下成员重新命名却没有改变在编译器中的对应名称
      -新加入的成员会被视为一般成员，除非你在编译器中加上适当支持
    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;
};
```

- 一般具现体(general instantiation)，内含对所有型别都必定有效的保守值。上述各个 has_trivial_xxx 型别都被定义为__false_type，就是对所有型别都必定有效的保守值
- 经过声明的特化版本，例如<type_traits.h>内对所有 C++标量型别提供了对应的特化声明
- 某些编译器会自动为所有型别提供适当的特化版本

4、__types_traits 在 SGI STL 中的应用很广

1) uninitialized_fill_n

```
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));
}
//value_type()：迭代器所指类型的萃取方法，返回的是该类型的指针，避免创建对象，造成额外的开销
//首先萃取出迭代器所指对象的类型

template <class ForwardIterator, class Size, class T, class T1>
inline ForwardIterator __uninitialized_fill_n(ForwardIterator first, Size n,
                                            const T& x, T1*) {
    typedef typename __type_traits<T1>::is_POD_type is_POD;
    return __uninitialized_fill_n_aux(first, n, x, is_POD());//构造了一个 is_POD 类型的对象，没有什么开销
}
//获取了迭代器所指对象的类型之后，可以萃取出其型别特性

template <class ForwardIterator, class Size, class T>
ForwardIterator
__uninitialized_fill_n_aux(ForwardIterator first, Size n,
                           const T& x, __false_type) {
    ForwardIterator cur = first;
    for ( ; n > 0; --n, ++cur)
        construct(&*cur, x);
    return cur;
}

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);//交由高阶函数执行，见如下
}

//定义于<stl_algobase.h>
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;
}
```

5、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?一个简

单的判断是：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 就需要实现自己的 non-trivial-xxx

3. 8. 总结

1、__type_traits<T>::xxx：所萃取出来的是一个作为"真假"判断的类型，这种类型没有任何成员，构造这样类型的成员，没有额外的开销

2、iterator_traits<T>::xxx：

1) 只有 iterator_category 类型成员是空 class
2) 其他类型成员(value_type、difference_type、pointer、reference)并不是作为标志的类型，而是与容器本身结构，或者保存元素的类型相关的类型，因此萃取时最好返回其类型的指针，避免返回该类型的对象而造成需要额外构造的操作

3、为了充分发挥 C++静态编译器的重载机制，在编译器完成重载分派，那么必须传入类型对象，或者指针，而不能是数值(int 或 bool 等运行时才能知道结果的东西)

1) 对于__true_type 或者__false_type，传入对象即可，因为构造这种类型的对象没有什么开销
2) 对于 value_type，传入该类型的指针即可，因为构造这种类型的对象可能产生较大的开销

4、每种容器的迭代器都可以理解为完全不同的，但是接受迭代器的算法或者方法都是模板方法，任何类型都可以作为迭代器，只要它能萃取出相应的型别即可

Chapter 4. 序列式容器

4.1. 容器的概观与分类

1、研究数据的特定排列方式，以利于搜寻或排序或其他特殊目的，这一专门学科称为数据结构
2、常用的数据结构

1) array
2) list
3) tree
4) stack
5) queue
6) hash table
7) set
8) map

➢ 根据在容器中排列的特性，这些数据结构分为序列式和关联式两种

4.1.1. 序列式容器(sequential containers)

1、所谓序列式容器，其中元素都可序(ordered)，但未必有序(sorted)
2、C++本身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。其中 stack 和 queue 只是将 deque 进行了封装，技术上被归类为配接器(adapter)

4.2. vector

4.2.1. vector 概述

1、vector 技术的实现，关键在于其对大小的控制以及重新配置时数据移动效率

4.2.2. vector 的迭代器

1、vector 维护的是一个连续线性空间，所以不论其元素型别为何，普通指针都可以作为 vector 的迭代器而满足所有必要条件

1) operator *
2) operator ->
3) operator ++
4) operator --
5) operator +
6) operator -
7) operator +=
8) operator -=

2、源码摘要

```
template <class T, class Alloc = alloc>
class vector {
public:
    //vector 的嵌套型别定义
```

```
        typedef T value_type;
        //vector 的迭代器就是指针类型，由于指针指向的就是 vector 中的元素，
        因此仅仅靠这个就把 vector 的迭代器和 vector 本身建立了关联
        typedef value_type* iterator;
    ...
    };
```

3、迭代器为指针时，其型别类型由 iterator_traits 的偏特化版本保证

4.2.3. vector 定义概要

1、部分源码(摘取了部分) (<stl_vector.h>)(已核对)

```
template <class T, class Alloc = alloc>
class vector {
public:
    //vector 的嵌套型别定义
    typedef T value_type;
    typedef value_type* pointer;
    typedef value_type* iterator;
    typedef value_type& reference;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

protected:
    //simple_alloc 是 SGI STL 的空间配置器，详见 2.2.4
    typedef simple_alloc<value_type, Alloc> data_allocator;
    iterator start;//表示目前使用空间头
    iterator finish;//表示目前使用空间尾
    iterator end_of_storage;//表示目前可用空间尾

    void insert_aux(iterator position, const T& x);
    void deallocate() {
        if (start)
            data_allocator::deallocate(start, end_of_storage - start);
    }

    void fill_initialize(size_type n, const T& value) {
        start = allocate_and_fill(n, value);
        finish = start + n;
        end_of_storage = finish;
    }
public:
    iterator begin() { return start; }
    iterator end() { return finish; }
    size_type size() const { return size_type(end() - begin()); }
    size_type capacity() const { return size_type(end_of_storage - begin()); }
```

```
bool empty() const { return begin() == end(); }
reference operator[](size_type n) { return *(begin() + n); }

vector() : start(0), finish(0), end_of_storage(0) {}
vector(size_type n, const T& value) { fill_initialize(n, value); }
vector(int n, const T& value) { fill_initialize(n, value); }
vector(long n, const T& value) { fill_initialize(n, value); }
explicit vector(size_type n) { fill_initialize(n, T()); }

~vector() {
    destroy(start, finish);//全局函数，详见 2.2.3
    deallocate();
}
reference front() { return *begin(); }
reference back() { return *(end() - 1); }
void push_back(const T& x) {
    if (finish != end_of_storage) {
        construct(finish, x);//全局函数，详见 2.2.3
        ++finish;
    }
    else
        insert_aux(end(), x);
}

void pop_back() {
    --finish;
    destroy(finish);//全局函数，详见 2.2.3
}

iterator erase(iterator position) {
    if (position + 1 != end())
        copy(position + 1, finish, position);//后续元素往前移动
    --finish;
    destroy(finish);//全局函数，详见 2.2.3
    return position;
}
void resize(size_type new_size, const T& x) {
    if (new_size < size())
        erase(begin() + new_size, end());
    else
        insert(end(), new_size - size(), x);
}
void resize(size_type new_size) { resize(new_size, T()); }
void clear() { erase(begin(), end()); }
```

```
protected:
    iterator allocate_and_fill(size_type n, const T& x) {
        iterator result = data_allocator::allocate(n);
        uninitialized_fill_n(result, n, x);
        return result;
    }
    //...
};
```

4.2.4. vector 的数据结构

1、vector 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 指向整块连续空间的尾端

2、源码摘要

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

3、为了降低空间配置时的速度成本，vector 实际配置的大小可能比客户端需求量更大一些，以备将来可能的扩充，这便是容量的概念。换句话说，vector 的容量大小永远大于等于其大小

4、运用 start、finish、end_of_storage 三个迭代器，便可轻易提供首尾标示，大小，容量，空容器判断，下标云算子，最前端元素值，最后端元素值等功能

```
template <class T, class Alloc = alloc>
class vector {
...
public:
    iterator begin() { return start; }
    iterator end() { return finish; }
    size_type size() const { return size_type(end() - begin()); }
    size_type capacity() const { return size_type(end_of_storage - begin()); }
    bool empty() const { return begin() == end(); }
    reference operator[](size_type n) { return *(begin() + n); }

    vector() : start(0), finish(0), end_of_storage(0) {}
    vector(size_type n, const T& value) { fill_initialize(n, value); }
    vector(int n, const T& value) { fill_initialize(n, value); }
    vector(long n, const T& value) { fill_initialize(n, value); }
```

```
        explicit vector(size_type n) { fill_initialize(n, T()); }
    ...
    };
```

4.2.5. vector 的构造与内存管理：constructor、push_back

1、vector 缺省使用 alloc 作为空间配置器，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 是 SGI STL 的空间配置器，详见 2.2.4
        typedef simple_alloc<value_type, Alloc> data_allocator;
    ...
    };
```

2、vector 提供了许多 constructors，其中一个允许我们指定空间大小以及初值

```
    vector(size_type n, const T& value) { fill_initialize(n, value); }

    //填充并初始化
    void fill_initialize(size_type n, const T& value) {
        start = allocate_and_fill(n, value);
        finish = start + n;
        end_of_storage = finish;
    }

    //配置而后填充
    iterator allocate_and_fill(size_type n, const T& x) {
        iterator result = data_allocator::allocate(n);
        uninitialized_fill_n(result, n, x);//全局函数，详见 2.3
        return result;
    }
```

3、push_back

```
    void push_back(const T& x) {
        if (finish != end_of_storage) {
            construct(finish, x);//全局函数，详见 2.2.3
            ++finish;
        }
        else//已无备用空间
            insert_aux(end(), x);
    }

    template <class T, class Alloc>
    void vector<T, Alloc>::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;
            //如果原大小为 0，则配置 1，若不为 0，则配置原大小的两倍，前
            半段用于存放原有数据，后半段用于存放新数据

            iterator new_start = data_allocator::allocate(len);
            iterator new_finish = new_start;

            __STL_TRY{
                //将原 vector 的前半段数据拷贝到新 vector
                new_finish = uninitialized_copy(start, position, new_start);
                //构造即将插入的元素到新 vector
                construct(new_finish, x);
                ++new_finish;
                //将原 vector 的后半段数据拷贝到新 vector
                new_finish = uninitialized_copy(position, finish, new_finish);
            }
            catch(...) {
                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;
        }
    }
```

4、动态增加大小，并不是在原来空间之后续新空间，而是以原大小的两倍另外配置一块较大空间，然后将原内容拷贝过去，**因此，对 vector 的操作如果引起空间重新配置，那么原有迭代器将会失效**

4.2.6. vector 的元素操作：pop_back、erase、clear、insert

1、源码摘要

```
void pop_back() {
    --finish;
    destroy(finish);
}
iterator erase(iterator position) {
    if (position + 1 != end())
        copy(position + 1, finish, position);//全局函数，见第六章
    --finish;
    destroy(finish);
    return position;
}

iterator erase(iterator first, iterator last) {
    iterator i = copy(last, finish, first);
    destroy(i, finish);
    finish = finish - (last - first);
    return first;
}

void clear() { erase(begin(), end()); }
```

2、insert 实现源码

```
//从 position 开始，插入 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 不为 0 才进行以下操作
        if (size_type(end_of_storage - finish) >= n) {
            //当备用空间个数大于等于新增元素个数
            T x_copy = x;
            //计算当前插入点之后现有元素个数
            const size_type elems_after = finish - position;
            iterator old_finish = finish;
            //为什么要这样再细分 elems_after 与 n 的关系：position-old_finish 之间是已经初始化过的内存，而 finish-end_of_storage 是已分配但未初始化的内存，对于这两种内存的操作方式是不同的!!!
            if (elems_after > n) {
                //插入点之后的现有元素个数比 n 大
                //将插入点之后的部分元素(finish-n,finish)共 n 个，使用 uninitialized_copy 在未初始化的区域进行初始化拷贝
                uninitialized_copy(finish - n, finish, finish);
                finish += n;//更新 finish
```

```
            //将插入点之后的部分元素(position,old_finish-n)通过
            copy_backward 转移到之前已经初始化过的内存中
            copy_backward(position, old_finish - n, old_finish);
            //在已经初始化过的区域中，调用全局函数 fill
            fill(position, position + n, x_copy);
        }
        else {
            //插入点之后的现有元素个数小于等于 n
            //先构造多余现有元素的部分
            uninitialized_fill_n(finish, n - elems_after, x_copy);
            finish += n - elems_after;//更新 finish
            //将现有元素挪到以 finish 为起始的地址中
            uninitialized_copy(position, old_finish, finish);
            finish += elems_after;//更新 finish
            fill(position, old_finish, x_copy);//在原有内存上填上插入
            的值，填的数量为 elems_after，而多余的 n- elems_after
            个元素，在第一步已经构造了
        }
    }
    else {
        //备用空间小于新增元素个数，就必须配置额外内存
        //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 {
            //将旧 vector 的插入点之前的元素复制到新空间
            new_finish = uninitialized_copy(start, position, new_start);
            //在将新增元素填入新空间
            new_finish = uninitialized_fill_n(new_finish, n, x);
            //再将旧 vector 的插入点之后的元素，复制到新空间
            new_finish = uninitialized_copy(position, finish, new_finish);
        }
        catch(...) {
            //如果有异常，实现 commit or rollback
            destroy(new_start, new_finish);
            data_allocator::deallocate(new_start, len);
            throw;
        }

        //清除并释放旧的 vector
        destroy(start, finish);
```

```
                deallocate();
                start = new_start;
                finish = new_finish;
                end_of_storage = new_start + len;
            }
        }
    }
```

4.3. list

4.3.1. list 概述

1、list 与 vector 不同，每次插入或删除一个元素，就配置或释放一个元素空间

4.3.2. list 的节点

1、list 本身和 list 节点是不同的结构，需要分开设计，以下是 list 节点的结构

```
template <class T>
struct __list_node {
    typedef void* void_pointer;
    //节点的 prev 和 next 指针指向的类型都是 void*，在使用时需要转换
    void_pointer next;
    void_pointer prev;
    T data;
};
```

4.3.3. list 的迭代器

1、list 不再能够像 vector 一样以普通指针作为迭代器，因为其节点不保证在存储空间中连续存在。list 迭代器必须有能力指向 list 节点，并有能力进行正确的递增、递减、取值、成员存取等操作

1) 递增时指向下一个节点
2) 递减时指向上一个节点
3) 取值时取得是节点的数据值
4) 成员取用时取用的是节点的成员

2、list 有一个重要的性质：插入操作和接合操作都不会造成原有的 list 迭代器失效，而 vector 却不行，因为 vector 可能会由于扩张导致内存重新配置

3、list 迭代器的设计

```
template<class T, class Ref, class Ptr>
struct __list_iterator {
    ...
    //iterator 和 self 有什么含义
    //iterator 的意义是什么?
    //在 list 中迭代器内建类型定义的如下
        typedef __list_iterator<T, T&, T*>          iterator;
        那么对于 iterator 和 self 有什么区别呢
    //self 就是迭代器本身
```

```
typedef __list_iterator<T, T&, T*>              iterator;
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;//指向节点的指针
typedef size_t size_type;
typedef ptrdiff_t difference_type;

//与 list 容器建立关联
link_type node;//迭代器内部有一个普通指针，指向 list 的节点

//该构造函数提供了从 link_type 向 iterator 转型
__list_iterator(link_type x) : node(x) {}
__list_iterator() {}
__list_iterator(const iterator& x) : node(x.node) {}

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

reference operator*() const { return (*node).data; }//对迭代器取值

pointer operator->() const { return &(operator*()); }//迭代器重载->的标准做法

self& operator++() {
        node = (link_type)((*node).next);
        return *this;
}
self operator++(int) {
        self tmp = *this;
        ++*this;
        return tmp;
}
self& operator--() {
        node = (link_type)((*node).prev);
        return *this;
}
self operator--(int) {
        self tmp = *this;
        --*this;
        return tmp;
```

```
    }
};
```

4.3.4. list 定义概要

1、部分源码(<stl_list.h>)(已核对)

```
template <class T, class Alloc = alloc>
class list {
protected:
    typedef void* void_pointer;
    typedef __list_node<T> list_node;//节点类型
    typedef simple_alloc<list_node, Alloc> list_node_allocator;
public:
    typedef T value_type;//泛型类型
    typedef value_type* pointer;
    typedef value_type& reference;
    typedef list_node* link_type;//节点指针类型
    typedef size_t size_type;
    ptrdiff_t difference_type;

public:
    typedef __list_iterator<T, T&, T*>              iterator;
    //其他迭代器
    ...
protected:
    //配置一个节点并传回
    link_type get_node() { return list_node_allocator::allocate(); }
    //释放一个节点
    void put_node(link_type p) { list_node_allocator::deallocate(p); }

    //产生(配置并构造)一个节点，带有元素值
    link_type create_node(const T& x) {
        link_type p = get_node();
        construct(&p->data, x);//全局函数，构造/析构基本工具
        return p;
    }
    //销毁(析构并释放)一个节点
    void destroy_node(link_type p) {
        destroy(&p->data); //全局函数，构造/析构基本工具
        put_node(p);
    }


protected:
    void empty_initialize() {
```

```
        node = get_node();
        node->next = node;//令头尾都指向自己，不设元素值
        node->prev = node;
    }

    void fill_initialize(size_type n, const T& value) {
        empty_initialize();
        insert(begin(), n, value);
    }

    template <class InputIterator>
    void range_initialize(InputIterator first, InputIterator last) {
        empty_initialize();
        insert(begin(), first, last);
    }

protected:
    link_type node;

public:
    //构造一个空链表
    list() { empty_initialize(); }

    iterator begin() { return (link_type)((*node).next); }
    iterator end() { return node; }

    bool empty() const { return node->next == node; }
    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()); }

    void swap(list<T, Alloc>& x) { __STD::swap(node, x.node); }
    iterator insert(iterator position, const T& x) {
        link_type tmp = create_node(x);
        tmp->next = position.node;
        tmp->prev = position.node->prev;
        (link_type(position.node->prev))->next = tmp;
        //position 是个 iterator 类型，node 是 link_type 类型，prev 是 void*，
        要将 void*转为 link_type
```

```
        position.node->prev = tmp;
        return tmp;
}
//其他 insert 不再介绍
...

//插入一个节点作为头节点
void push_front(const T& x) { insert(begin(), x); }
//插入一个节点作为尾节点
void push_back(const T& x) { insert(end(), x); }

//移出迭代器 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);
}

void clear();

void pop_front() { erase(begin()); }
void pop_back() {
        iterator tmp = end();
        erase(--tmp);//如果链表是空，那么好像有问题
}

//构造函数
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); }
explicit list(size_type n) { fill_initialize(n, T()); }

//接受迭代器的构造函数
template <class InputIterator>
list(InputIterator first, InputIterator last) {
        range_initialize(first, last);
}

//拷贝构造函数
list(const list<T, Alloc>& x) {
        range_initialize(x.begin(), x.end());
```

```
    }
    ~list() {
        clear();
        put_node(node);
    }
    list<T, Alloc>& operator=(const list<T, Alloc>& x);

protected:
    //将[first,last)内的所有元素移到 position 之前，如果 position 处于[first,last)之间怎么办
    void transfer(iterator position, iterator first, iterator last) {
        if (position != last) {
            //[first,last)的有效尾部与 position 相连
            (*(link_type((*last.node).prev))).next = position.node;
            //将缝合住被取走的部分
            (*(link_type((*first.node).prev))).next = last.node;
            //将 position 原来的前继节点连到 first 上
            (*(link_type((*position.node).prev))).next = first.node;
            //以下将双向另一根补全
            link_type tmp = link_type((*position.node).prev);
            (*position.node).prev = (*last.node).prev;
            (*last.node).prev = (*first.node).prev;
            (*first.node).prev = tmp;
        }
    }
    //差异在于这种方式充分利用了变更顺序，以用最少的额外空间完成一个交换

public:
    //将 x 接合与 position 所指位置之前，x 必须不同于*this
    void splice(iterator position, list& x) {
        if (!x.empty())
            transfer(position, x.begin(), x.end());
    }
    //将 i 所指元素接合与 position 所指位置之前，position 与 i 可指向同一个 list
    void splice(iterator position, list&, iterator i) {
        iterator j = i;
        ++j;
        if (position == i || position == j) return;
        transfer(position, i, j);
    }

    //将[first,last)内的所有元素接合与 position 所指位置之前
```

```
        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();
};

template <class T, class Alloc>
void list<T, Alloc>::remove(const T& value) {
        iterator first = begin();
        iterator last = end();
        while (first != last) {
            iterator next = first;
            ++next;
            if (*first == value) erase(first);
            first = next;
        }
}

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;//保证循环开始前，next 与 fist 指向同一个
        }
        //以上这段思路很好
}

//合并有序链表，将 x 合并到当前链表中
template <class T, class Alloc>
void list<T, Alloc>::merge(list<T, Alloc>& x) {
        iterator first1 = begin();
        iterator last1 = end();
```

```
        iterator first2 = x.begin();
        iterator last2 = x.end();
        while (first1 != last1 && first2 != last2)
            if (*first2 < *first1) {//x 中的当前节点小于本链表当前节点
                iterator next = first2;
                transfer(first1, first2, ++next);
                first2 = next;
            }
            else
                ++first1;
        if (first2 != last2) transfer(last1, first2, last2);//合并剩余的
}

template <class T, class Alloc>
void list<T, Alloc>::reverse() {
        //判断是否为空或者仅有一个元素
        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);
        }
}

template <class T, class Alloc>
void list<T, Alloc>::sort() {
        //判断是否为空或者仅有一个元素
        if (node->next == node || link_type(node->next)->next == node) return;
        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]);
    }
```

4.3.5. list 的数据结构

1、SGI list 不仅是一个双向链表，还是一个环状双向链表，所以它只需要一个指针便可以完整表现整个链表

```
    template<class T, class Ref, class Ptr>
    struct __list_iterator {
    ...
    protected:
        link_type node;
    ...
    };
```

2、如果让指针 node 指向刻意置于尾端的空白节点，node 便能符合 STL 对于前闭后开区间的要求，成为 last 迭代器

```
    template <class T, class Alloc = alloc>
    class list {
    protected:
        typedef void* void_pointer;
        typedef __list_node<T> list_node;
        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;//还是指向节点的指针
        typedef size_t size_type;
        typedef ptrdiff_t difference_type;

        public:
        typedef __list_iterator<T, T&, T*>              iterator;
    ...
        iterator begin() { return (link_type)((*node).next); }
        //link_type 如何转为 iterator 类型???
        iterator end() { return node; }
        bool empty() const { return node->next == node; }
        size_type size() const {
            size_type result = 0;
            distance(begin(), end(), result);//第三个形参类型是引用
            return result;
        }
        reference front() { return *begin(); }
        reference back() { return *(--end()); }
```

```
...
};
```

4.3.6. list 的构造与内存管理:constructor,push_back,insert

1、list 缺省使用 alloc(2.2.4)作为空间配置器

```
template <class T, class Alloc = alloc>
class list {
...
protected:
    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;
...
protected:
    //配置一个节点并传回
    link_type get_node() { return list_node_allocator::allocate(); }
    //释放一个节点
    void put_node(link_type p) { list_node_allocator::deallocate(p); }

    //产生(配置并构造)一个节点，带有元素值
    link_type create_node(const T& x) {
        link_type p = get_node();
        construct(&p->data, x);//全局函数，构造/析构基本工具
        return p;
    }
    //销毁(析构并释放)一个节点
    void destroy_node(link_type p) {
        destroy(&p->data); //全局函数，构造/析构基本工具
        put_node(p);
    }
..
public:
    list() { empty_initialize(); }

protected:
    void empty_initialize() {
        node = get_node();
        node->next = node;//令头尾都指向自己，不设元素值
        node->prev = node;
    }
..
public:
    void push_back(const T& x) { insert(end(), x); }
```

```
        iterator insert(iterator position, const T& x) {
            link_type tmp = create_node(x);
            tmp->next = position.node;
            tmp->prev = position.node->prev;
            (link_type(position.node->prev))->next = tmp;
            //position 是个 iterator 类型，node 是 link_type 类型，prev 是 void*，
            要将 void*转为 link_type
            position.node->prev = tmp;
            return tmp;
        }
        //插入完成后，新节点将位于哨兵节点(标示出插入点)所指节点的前方--
        --这是 STL 对于"插入操作"的规范
        //由于 list 不像 vector 那样有可能在空间不足时做重新配置、数据移动
        的操作，所以插入前的所有迭代器在插入操作之后都仍然有效

    };
```

4.3.7. list 的元素操作

1、元素操作有

1) push_front
2) push_back
3) erase
4) pop_front
5) pop_back
6) clear
7) remove
8) unique
9) eplice
10) merge
11) reverse
12) sort

2、其实就是链表的标准操作，已经蛮熟悉了，不过看看人家怎么写的吧

```
    template <class T, class Alloc = alloc>
    class list {
    ...
    public:
        //插入一个节点作为头节点
        void push_front(const T& x) { insert(begin(), x); }
        //插入一个节点作为尾节点
        void push_back(const T& x) { insert(end(), x); }

        //移出迭代器 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);
    }

protected:
    //将[first,last)内的所有元素移到 position 之前，如果 position 处于[first,last)之间怎么办
    void transfer(iterator position, iterator first, iterator last) {
        if (position != last) {
            //[first,last)的有效尾部与 position 相连
            (*(link_type((*last.node).prev))).next = position.node;
            //将缝合住被取走的部分
            (*(link_type((*first.node).prev))).next = last.node;
            //将 position 原来的前继节点连到 first 上
            (*(link_type((*position.node).prev))).next = first.node;
            //以下将双向另一根补全
            link_type tmp = link_type((*position.node).prev);
            (*position.node).prev = (*last.node).prev;
            (*last.node).prev = (*first.node).prev;
            (*first.node).prev = tmp;
        }
    }
    //差异在于这种方式充分利用了变更顺序，以用最少的额外空间完成一个交换

public:
    //将 x 接合与 position 所指位置之前，x 必须不同于*this
    void splice(iterator position, list& x) {
        if (!x.empty())
            transfer(position, x.begin(), x.end());
    }
    //将 i 所指元素接合与 position 所指位置之前，position 与 i 可指向同一个 list
    void splice(iterator position, list&, iterator i) {
        iterator j = i;
        ++j;
        if (position == i || position == j) return;
        transfer(position, i, j);
    }

    //将[first,last)内的所有元素接合与 position 所指位置之前
```

```
        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();
};

template <class T, class Alloc>
void list<T, Alloc>::remove(const T& value) {
        iterator first = begin();
        iterator last = end();
        while (first != last) {
            iterator next = first;
            ++next;
            if (*first == value) erase(first);
            first = next;
        }
}

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;//保证循环开始前，next 与 fist 指向同一个
        }
        //以上这段思路很好
}

//合并有序链表，将 x 合并到当前链表中
template <class T, class Alloc>
void list<T, Alloc>::merge(list<T, Alloc>& x) {
        iterator first1 = begin();
        iterator last1 = end();
```

```
        iterator first2 = x.begin();
        iterator last2 = x.end();
        while (first1 != last1 && first2 != last2)
            if (*first2 < *first1) {//x 中的当前节点小于本链表当前节点
                iterator next = first2;
                transfer(first1, first2, ++next);
                first2 = next;
            }
            else
                ++first1;
        if (first2 != last2) transfer(last1, first2, last2);//合并剩余的
}

template <class T, class Alloc>
void list<T, Alloc>::reverse() {
        //判断是否为空或者仅有一个元素
        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);
        }
}

template <class T, class Alloc>
void list<T, Alloc>::sort() {
        //判断是否为空或者仅有一个元素
        if (node->next == node || link_type(node->next)->next == node) return;
        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

4. 4. 1. deque 概述

1、vector 是单向开口的连续线性空间，deque 是一种双向开口的连续线性空间

2、deque 和 vector 的最大差异

1) deque 允许于常数时间内对起头端进行元素的插入操作或移出操作

2) deque 没有所谓容量概念，因为它是动态地以分段连续空间组合而成，随时可以增加一段新的空间并链接起来，因此不会出现像 vector 一样因旧空间不足而重新分配一块更大空间，然后复制元素，再释放旧空间的行为

➢ 这就意味着，deque 的空间并非一直连续，而是分段连续的

3、虽然 deque 也提供 Random Access Iterator，但它的迭代器并不是普通指针，复杂度和 vector 不同。因此除非必要，我们应该尽量选择 vector 而非 deque

4、对 deque 的最高效的排序：将 deque 复制到 vector，然后排序后再复制回 deque

4. 4. 2. deque 的中控器

1、deque 是逻辑上的连续空间，deque 由一段段的定量连续空间构成，一旦有必要在 deque 的前端或尾端增加新空间，便配置一段定量连续空间，串接在整个 deque 的头端或尾端

2、deque 的最大任务便是在这些分段定量连续空间上，维护整体连续的假象，并提供随机存取的接口，避开了"重新配置、复制、释放"的轮回，代价则是复杂的迭代器架构

3、deque 采用一块所谓的 map(不是 STL 的 map 容器)作为主控，这里的 map 指的是一小块连续空间，其中每个元素(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 的存储空间主体，SGI STL 允许指定缓冲区大小，默认值 0 表示使用 512bytes 缓冲区

4. 4. 3. deque 迭代器

1、deque 是分段连续空间，维持其"整体连续"假象的任务，落在了迭代器的 operator++和 operator--两个运算子上

2、deque 迭代器必须具备的结构

1) 必须能够指出分段连续空间(缓冲区)在哪里

2) 其次它必须能够判断自己是否处于其所在缓冲区的边缘

- 如果是，一旦前进或后退，就必须跳跃到下一个或上一个缓冲区
- 为了能够跳跃，deque 必须能够掌握管控中心(map)

3、下面给出 deque 迭代器源码

```
template <class T, class Ref, class Ptr, size_t BufSiz>
struct __deque_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
    typedef random_access_iterator_tag iterator_category;
```

```
typedef T value_type;
typedef Ptr pointer;
typedef Ref reference;
typedef size_t size_type;
typedef ptrdiff_t difference_type;
typedef T** map_pointer;

typedef __deque_iterator self;

//保持与 deque 容器的联系，迭代器指向容器中的某一个元素，该元素位于且仅位于其中一个缓冲区
T* cur;//迭代器现在所指向的元素
T* first;//迭代器所指向的缓冲区头
T* last;//迭代器所指向的缓冲区尾
map_pointer node;//指向管控中心，以便进行跨缓冲区操作

__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; }
pointer operator->() const { return &(operator*()); }

difference_type operator-(const self& x) const {
    return difference_type(buffer_size()) * (node - x.node - 1) +
        (cur - first) + (x.last - x.cur);
}

self& operator++() {
    ++cur;
    if (cur == last) {
        set_node(node + 1);
        cur = first;
    }
    return *this;
}
self operator++(int)    {
    self tmp = *this;//this 是指向当前迭代器的指针
    ++*this;//调用前置递增版本
    return tmp;
}
```

```
self& operator--() {
    if (cur == first) {
        set_node(node - 1);
        cur = last;
    }
    --cur;
    return *this;
}
self operator--(int) {
    self tmp = *this;
    --*this;
    return tmp;
}

//本身是要改变的
self& operator+=(difference_type n) {
    difference_type offset = n + (cur - first);
    if (offset >= 0 && offset < difference_type(buffer_size()))
        //仍然处于同一个缓冲区
        cur += n;
    else {
        //计算出缓冲区偏移量，切换到正确的缓冲区
        difference_type node_offset =
            offset > 0 ? offset / difference_type(buffer_size())
            : -difference_type((-offset - 1) / buffer_size()) - 1;
        set_node(node + node_offset);
        //此时 first 指向正确的缓冲区头，再加上在该缓冲区内的偏移
        量，即 cur 的指向
        cur = first + (offset - node_offset * difference_type(buffer_size()));
    }
    return *this;
}

//本身是不变的，返回的是一个临时量
self operator+(difference_type n) const {
    self tmp = *this;
    return tmp += n;//显式调用 operator +=
}

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 {
            //首先判断是否在同一个缓冲区，如果是在缓冲区内比，若不是，
            比较缓冲区的关系即可
            return (node == x.node) ? (cur < x.cur) : (node < x.node);
        }

        void set_node(map_pointer new_node) {
            node = new_node;
            first = *new_node;
            last = first + difference_type(buffer_size());
        }
};

//每个缓冲区放置多少个元素
//若 n 不为 0，传回 n，表示 buffer size 由用户自定义，即一个缓冲区放置 n
个元素
//若 n 为 0，表示 buffer size 使用默认值，那么
  1)  若 sz 小于 512，传回 512/sz，每个缓冲区放置 512/sz 个元素
  2)  若 sz 不小于 512，传回 1，每个缓冲区放置一个元素
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));
}
```

4.4.4. deque 定义概要

1、部分源码(摘取了部分)(已核对)

```
template <class T, class Alloc = alloc, size_t BufSiz = 0>
class deque {
public:// Basic types
    typedef T value_type;
    typedef value_type* pointer;
    typedef value_type& reference;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

public: // Iterators
    typedef __deque_iterator<T, T&, T*, BufSiz>                    iterator;
```

```
protected: // Internal typedefs
    typedef pointer* map_pointer;
    typedef simple_alloc<value_type, Alloc> data_allocator;
    typedef simple_alloc<pointer, Alloc> map_allocator;

    static size_type buffer_size() {
        return __deque_buf_size(BufSiz, sizeof(value_type));
    }
    static size_type initial_map_size() { return 8; }

protected: // Data members
    iterator start;//第一个节点
    iterator finish;//表现最后一个节点

    map_pointer map;//指向 map，map 是块连续空间，每个元素都是指针，
    指向一个节点(缓冲区)
    size_type map_size;

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

    reference operator[](size_type n) { return start[difference_type(n)]; }

    reference front() { return *start; }

    reference back() {
        iterator tmp = finish;
        --tmp;
        return *tmp;
        //以上三行为何不改为 return *(finish-1);
    }

    //最后两个分号???
    size_type size() const { return finish - start;; }
    size_type max_size() const { return size_type(-1); }
    bool empty() const { return finish == start; }

public: // Constructor, destructor.
    deque()
        : start(), finish(), map(0), map_size(0)
    {
        create_map_and_nodes(0);
```

```
}

deque(const deque& x)
      : start(), finish(), map(0), map_size(0)
{
      create_map_and_nodes(x.size());
      uninitialized_copy(x.begin(), x.end(), start);//详见 2.3.1
}

deque(size_type n, const value_type& value)
      : start(), finish(), map(0), map_size(0)
{
      fill_initialize(n, value);
}

deque(int n, const value_type& value)
      : start(), finish(), map(0), map_size(0)
{
      fill_initialize(n, value);
}

deque(long n, const value_type& value)
      : start(), finish(), map(0), map_size(0)
{
      fill_initialize(n, value);
}

explicit deque(size_type n)
      : start(), finish(), map(0), map_size(0)
{
      fill_initialize(n, value_type());
}


template <class InputIterator>
deque(InputIterator first, InputIterator last)
      : start(), finish(), map(0), map_size(0)
{
      range_initialize(first, last, iterator_category(first));
}

~deque() {
      destroy(start, finish);
      destroy_map_and_nodes();
```

```
    }

    deque& operator= (const deque& x) {
        const size_type len = size();
        if (&x != this) {//保证自赋值的正确性
            if (len >= x.size())
                erase(copy(x.begin(), x.end(), start), finish);//???
            else {
                const_iterator mid = x.begin() + difference_type(len);
                copy(x.begin(), mid, start);
                insert(finish, mid, x.end());
            }
        }
        return *this;
    }

    void swap(deque& x) {
        __STD::swap(start, x.start);
        __STD::swap(finish, x.finish);
        __STD::swap(map, x.map);
        __STD::swap(map_size, x.map_size);
    }

public: // push_* and pop_*

    void push_back(const value_type& t) {
        if (finish.cur != finish.last - 1) {
            //最后缓冲区尚有两个以上元素的备用空间
            construct(finish.cur, t);//详见 2.2.3
            ++finish.cur;
        }
        else
            //最后缓冲区只剩一个元素备用空间
            push_back_aux(t);
    }

    void push_front(const value_type& t) {
        if (start.cur != start.first) {
            //第一缓冲区尚有备用空间，与 push_back 不同，有 1 个就行
            construct(start.cur - 1, t);
            --start.cur;
        }
        else
            //第一缓冲区已无备用空间
```

```
            push_front_aux(t);
        }

        void pop_back() {
            if (finish.cur != finish.first) {
                //删除最后一个元素后，最后缓冲区有一个或更多元素
                --finish.cur;
                destroy(finish.cur);
            }
            else
                //当前元素是缓冲区的唯一一个元素，删除该元素后需要释放
                缓冲区
                pop_back_aux();
        }

        void pop_front() {
            if (start.cur != start.last - 1) {
                //第一个缓冲区有两个或更多元素
                destroy(start.cur);
                ++start.cur;
            }
            else
                //当前元素是缓冲区的唯一一个元素，删除该元素后需要释放
                缓冲区
                pop_front_aux();
        }

    public: // Insert

        //返回插入元素的迭代器
        iterator insert(iterator position, const value_type& x) {
            if (position.cur == start.cur) {
                //插入的位置是 deque 的头部
                push_front(x);
                return start;
            }
            else if (position.cur == finish.cur) {
                //插入的位置是 deque 的尾部
                push_back(x);
                iterator tmp = finish;
                --tmp;
                return tmp;
            }
            else {
```

```
            //在中间插入，开销会很大
            return insert_aux(position, x);
        }
    }

    iterator insert(iterator position) { return insert(position, value_type()); }

    void insert(iterator pos, size_type n, const value_type& x);

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


    template <class InputIterator>
    void insert(iterator pos, InputIterator first, InputIterator last) {
        insert(pos, first, last, iterator_category(first));
    }

    void resize(size_type new_size, const value_type& x) {
        const size_type len = size();
        if (new_size < len)
            erase(start + new_size, finish);
        else
            insert(finish, new_size - len, x);
    }

    void resize(size_type new_size) { resize(new_size, value_type()); }

public: // Erase
    iterator erase(iterator pos) {
        iterator next = pos;
        ++next;
        difference_type index = pos - start;//清点之前元素个数
        if (index < (size() >> 1)) {//如果清点之前的元素比较少，就移动清楚点之前的元素
            copy_backward(start, pos, next);
            pop_front();//移动完毕，最前面的元素冗余，除去
        }
        else {//清除点之后的元素比较少
            copy(next, finish, pos);//移除清楚点之后的元素
```

```
                pop_back();//移动完毕，最后一个元素冗余，除去
            }
            return start + index;
        }

        iterator erase(iterator first, iterator last);
        void clear();

protected: // Internal construction/destruction

        void create_map_and_nodes(size_type num_elements);
        void destroy_map_and_nodes();
        void fill_initialize(size_type n, const value_type& value);


        template <class InputIterator>
        void range_initialize(InputIterator first, InputIterator last,
                              input_iterator_tag);

        template <class ForwardIterator>
        void range_initialize(ForwardIterator first, ForwardIterator last,
                              forward_iterator_tag);

protected: // Internal push_* and pop_*

        void push_back_aux(const value_type& t);
        void push_front_aux(const value_type& t);
        void pop_back_aux();
        void pop_front_aux();

protected: // Internal insert functions


        template <class InputIterator>
        void insert(iterator pos, InputIterator first, InputIterator last,
                    input_iterator_tag);

        template <class ForwardIterator>
        void insert(iterator pos, ForwardIterator first, ForwardIterator last,
                    forward_iterator_tag);

        iterator insert_aux(iterator pos, const value_type& x);
        void insert_aux(iterator pos, size_type n, const value_type& x);
```

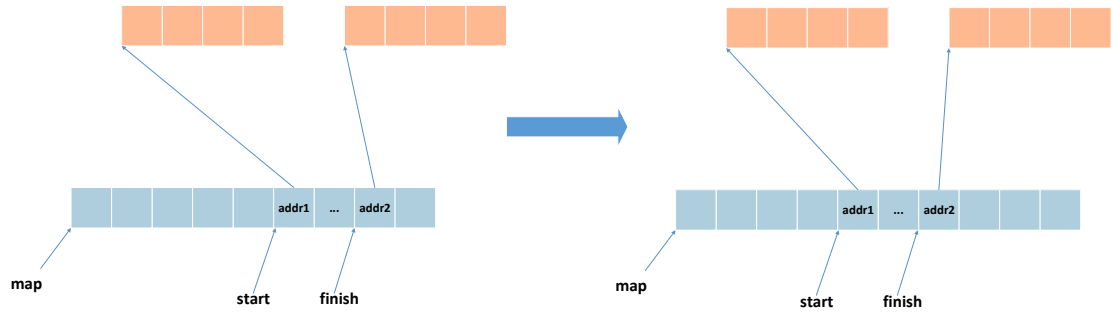
```
        template <class ForwardIterator>
        void insert_aux(iterator pos, ForwardIterator first, ForwardIterator last,
                        size_type n);


        iterator reserve_elements_at_front(size_type n) {
            size_type vacancies = start.cur - start.first;
            if (n > vacancies)
                new_elements_at_front(n - vacancies);
            return start - difference_type(n);
        }

        iterator reserve_elements_at_back(size_type n) {
            size_type vacancies = (finish.last - finish.cur) - 1;
            if (n > vacancies)
                new_elements_at_back(n - vacancies);
            return finish + difference_type(n);
        }

        void new_elements_at_front(size_type new_elements);
        void new_elements_at_back(size_type new_elements);

        void destroy_nodes_at_front(iterator before_start);
        void destroy_nodes_at_back(iterator after_finish);

    protected: // Allocation of map and nodes

        //Makes sure the map has space for new nodes. Does not actually
        //add the nodes. Can invalidate map pointers. (And consequently,
        //deque iterators.)

        void reserve_map_at_back (size_type nodes_to_add = 1) {
            //如果 map 尾端的节点备用空间不足，必须重新换个 map
            if (nodes_to_add + 1 > map_size - (finish.node - map))
                reallocate_map(nodes_to_add, false);
        }

        void reserve_map_at_front (size_type nodes_to_add = 1) {
            //如果 map 前端的节点备用空间不足，必须重新换个 map
            if (nodes_to_add > start.node - map)
                reallocate_map(nodes_to_add, true);
        }

        void reallocate_map(size_type nodes_to_add, bool add_at_front);
```

```
        pointer allocate_node() { return data_allocator::allocate(buffer_size()); }
        void deallocate_node(pointer n) {
            data_allocator::deallocate(n, buffer_size());
        }

        public:
        bool operator==(const deque<T, Alloc, 0>& x) const {
            return size() == x.size() && equal(begin(), end(), x.begin());
        }
        bool operator!=(const deque<T, Alloc, 0>& x) const {
            return size() != x.size() || !equal(begin(), end(), x.begin());
        }
        bool operator<(const deque<T, Alloc, 0>& x) const {
            return lexicographical_compare(begin(), end(), x.begin(), x.end());
        }
};

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::fill_initialize(size_type n,
                                               const value_type& value) {
        create_map_and_nodes(n);
        map_pointer cur;
        __STL_TRY {
            //为每个节点的缓冲区设定初始值，cur 指向缓冲区
            for (cur = start.node; cur < finish.node; ++cur)
                //cur 指向缓冲区，因此*cur 指向指定缓冲区的首元素，
                buffer_size()返回每个缓冲区的元素个数
                uninitialized_fill(*cur, *cur + buffer_size(), value);
            //最后一个节点的设定稍不同，因为尾端有备用空间，不必设初值
            uninitialized_fill(finish.first, finish.cur, value);
        }
        catch(...) {
            for (map_pointer n = start.node; n < cur; ++n)
                destroy(*n, *n + buffer_size());
            destroy_map_and_nodes();
            throw;
        }
}

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(缓冲区个数)
```

```
    size_type num_nodes = num_elements / buffer_size() + 1;

    //每个 map 管理几个节点，最少 8 个，最多所需节点加 2
    map_size = max(initial_map_size(), num_nodes + 2);
    map = map_allocator::allocate(map_size);

    //下面让 nstart 和 nfinish 指向 map 所拥有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，每个节点对应一个缓冲区
    map_pointer nstart = map + (map_size - num_nodes) / 2;//中间的缓冲区
    map_pointer nfinish = nstart + num_nodes - 1;//起始缓冲区加偏移量

    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有一些富裕)
        for (cur = nstart; cur <= nfinish; ++cur)
            *cur = allocate_node();//分配一个缓冲区的内存，返回该块缓冲区的头指针，赋值给*cur，因为*cur 就是指向指定缓冲区的头元素的指针
    }
    catch(...) {
        //commit or rollback
        for (map_pointer n = nstart; n < cur; ++n)
            deallocate_node(*n);
        map_allocator::deallocate(map, map_size);
        throw;
    }

    start.set_node(nstart);
    finish.set_node(nfinish);
    start.cur = start.first;
    //如果刚好整除会多配一个节点
    finish.cur = finish.first + num_elements % buffer_size();
}

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();//符合某个条件则必须重新换一个 map
    *(finish.node + 1) = allocate_node();//配置一个新节点
    __STL_TRY {
        construct(finish.cur, t_copy);//在原来缓冲区最后一个位置安置元素
        finish.set_node(finish.node + 1);//改变 finish，令其指向新缓冲区
```

```
            finish.cur = finish.first;//设定 finish 状态
        }
        __STL_UNWIND(deallocate_node(*(finish.node + 1)));
}

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();//若符合某种条件必须更换一个 map
        *(start.node - 1) = allocate_node();//配置一个新节点
        __STL_TRY {
            start.set_node(start.node - 1);//改变 start，指向新节点
            start.cur = start.last - 1;//设定 start 状态
            construct(start.cur, t_copy);//针对标的元素设值
        }
        catch(...) {
            start.set_node(start.node + 1);
            start.cur = start.first;
            deallocate_node(*(start.node - 1));
            throw;
        }
}

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) {
            //如果 map 现有空间足够大，将缓冲区调整到中间的位置，移动并
            不会改变元素相对于缓冲区的偏移量
            new_nstart = map + (map_size - new_num_nodes) / 2
                        + (add_at_front ? nodes_to_add : 0);
            if (new_nstart < start.node)
                //向前移动，注意此处的移动只是改变 map 的指向而已，缓冲
                区的实际地址没有发生改变(原有元素仍然在原来的地址上)，
                因此迭代器不会失效，迭代器指向的是元素而非 map，如下图
                所示
                copy(start.node, finish.node + 1, new_nstart);
            else
                //向后移动
                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;

            map_pointer new_map = map_allocator::allocate(new_map_size);
            //同样，将使用的缓冲区挪到 map 的中间区域，移动并不会改变元素相对于缓冲区的偏移量
            new_nstart = new_map + (new_map_size - new_num_nodes) / 2
                    + (add_at_front ? nodes_to_add : 0);
            copy(start.node, finish.node + 1, new_nstart);
            map_allocator::deallocate(map, map_size);

            map = new_map;
            map_size = new_map_size;
        }

        //重新设置迭代器 start 和 finish
        start.set_node(new_nstart);
        finish.set_node(new_nstart + old_num_nodes - 1);
}
```




```
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>:: pop_back_aux() {
        deallocate_node(finish.first);//释放掉最后一个缓冲区
        finish.set_node(finish.node - 1);//调整 finish 的状态，使指向上一个缓冲区的最后一个元素
        finish.cur = finish.last - 1;
        destroy(finish.cur);
}

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::pop_front_aux() {
        destroy(start.cur);//将缓冲区第一个(唯一一个)元素析构
```

```
        deallocate_node(start.first);//释放第一个缓冲区
        start.set_node(start.node + 1);//调整 start 的状态
        start.cur = start.first;
}

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::clear() {
        //以下针对头尾意外每一个缓冲区(除了头尾节点一定是饱满的)
        for (map_pointer node = start.node + 1; node < finish.node; ++node) {
                destroy(*node, *node + buffer_size());
                data_allocator::deallocate(*node, buffer_size());
        }

        if (start.node != finish.node) {//至少含有头尾两个缓冲区
                destroy(start.cur, start.last);
                destroy(finish.first, finish.cur);
                data_allocator::deallocate(finish.first, buffer_size());
        }
        else//只有一个缓冲区
                destroy(start.cur, finish.cur);

        finish = start;
}

template <class T, class Alloc, size_t BufSize>
deque<T, Alloc, BufSize>::iterator
deque<T, Alloc, BufSize>::erase(iterator first, iterator last) {
        //如果清除区间就是整个 deque，调用 clear 即可
        if (first == start && last == finish) {
                clear();
                return finish;
        }
        else {
                difference_type n = last - first;//清除区间长度
                difference_type elems_before = first - start;//清除区间前方元素个数
                if (elems_before < (size() - n) / 2) {//如果前方元素比较少
                        copy_backward(start, first, last);//向后移动前方元素
                        iterator new_start = start + n;
                        destroy(start, new_start);//析构冗余元素
                        //下面将缓冲区释放
                        for (map_pointer cur = start.node; cur < new_start.node; ++cur)
                                data_allocator::deallocate(*cur, buffer_size());
                        start = new_start;//设置 deque 新起点
                }
```

```
            else {//如果清除区间后方元素比较少
                copy(last, finish, first);//向前移动后方元素
                iterator new_finish = finish - n;
                destroy(new_finish, finish); //析构冗余元素
                //下面将缓冲区释放
                for (map_pointer cur = new_finish.node + 1; cur <= finish.node;
                ++cur)
                    data_allocator::deallocate(*cur, buffer_size());
                finish = new_finish; //设置 deque 新尾点
            }
            return start + elems_before;
        }
}

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个元素同值得元素
        iterator front1 = start;
        ++front1;
        iterator front2 = front1;
        ++front2;
        pos = start + index;//插入元素的位置
        iterator pos1 = pos;
        ++pos1;
        copy(front2, pos1, front1);
    }
    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;
        --back1;
        iterator back2 = back1;
        --back2;
        pos = start + index;//插入元素的位置
        copy_backward(pos, back2, back1);
    }
    *pos = x_copy;
    return pos;
}
```

4.4.5. deque 的数据结构

1、deque 除了维护一个指向 map 外的指针外，还维护 start，finish 两个迭代器

1) start 指向第一个缓冲区的第一个元素
2) finish 指向最后一个缓冲区的周后一个元素的下一个位置

2、除此之外，deque 还必须维护 map 的大小，以便在 map 所提供的节点不足时，进行 map 的重新配置

3、源码如下：

```
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
        typedef pointer* map_pointer;

protected: // Data members
        iterator start;//第一个节点
        iterator finish;//表现最后一个节点

        map_pointer map;//指向 map，map 是块连续空间，每个元素都是指针，
        指向一个节点(缓冲区)
        size_type map_size;

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

        reference operator[](size_type n) { return start[difference_type(n)]; }

        reference front() { return *start; }

        reference back() {
                iterator tmp = finish;
                --tmp;
                return *tmp;
                //以上三行为何不改为 return *(finish-1);
        }

        //最后两个分号???
```

```
        size_type size() const { return finish - start;; }
        size_type max_size() const { return size_type(-1); }
        bool empty() const { return finish == start; }
    };
```

4.4.6. deque 的构造与内存管理 ctor、push_back、push_front

1、deque 自定义两个专属的空间配置器

2、源码详解

```
    template <class T, class Alloc = alloc, size_t BufSiz = 0>
    class deque {
    ...
    protected: // Internal typedefs
        typedef simple_alloc<value_type, Alloc> data_allocator;
        typedef simple_alloc<pointer, Alloc> map_allocator;

        deque()
            : start(), finish(), map(0), map_size(0)
        {
            create_map_and_nodes(0);
        }
    ...
    public:
        void push_back(const value_type& t) {
            if (finish.cur != finish.last - 1) {
                //最后缓冲区尚有两个以上元素的备用空间
                construct(finish.cur, t);//详见 2.2.3
                ++finish.cur;
            }
            else
                //最后缓冲区只剩一个元素备用空间
                push_back_aux(t);
        }

        void push_front(const value_type& t) {
            if (start.cur != start.first) {
                //第一缓冲区尚有备用空间，与 push_back 不同，有 1 个就行
                construct(start.cur - 1, t);
                --start.cur;
            }
            else
                //第一缓冲区已无备用空间
                push_front_aux(t);
        }
```

```
protected:
    void reserve_map_at_back (size_type nodes_to_add = 1) {
        //如果 map 尾端的节点备用空间不足，必须重新换个 map
        if (nodes_to_add + 1 > map_size - (finish.node - map))
            reallocate_map(nodes_to_add, false);
    }

    void reserve_map_at_front (size_type nodes_to_add = 1) {
        //如果 map 前端的节点备用空间不足，必须重新换个 map
        if (nodes_to_add > start.node - map)
            reallocate_map(nodes_to_add, true);
    }

...
};

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::fill_initialize(size_type n,
                                               const value_type& value) {
    create_map_and_nodes(n);
    map_pointer cur;
    __STL_TRY {
        //为每个节点的缓冲区设定初始值，cur 指向缓冲区
        for (cur = start.node; cur < finish.node; ++cur)
            //cur 指向缓冲区，因此*cur 指向指定缓冲区的首元素，
            buffer_size()返回每个缓冲区的元素个数
            uninitialized_fill(*cur, *cur + buffer_size(), value);
        //最后一个节点的设定稍不同，因为尾端有备用空间，不必设初值
        uninitialized_fill(finish.first, finish.cur, value);
    }
    catch(...) {
        for (map_pointer n = start.node; n < cur; ++n)
            destroy(*n, *n + buffer_size());
        destroy_map_and_nodes();
        throw;
    }
}

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(缓冲区个数)
    size_type num_nodes = num_elements / buffer_size() + 1;
```

```
    //每个 map 管理几个节点，最少 8 个，最多所需节点加 2
    map_size = max(initial_map_size(), num_nodes + 2);
    map = map_allocator::allocate(map_size);

    //下面让 nstart 和 nfinish 指向 map 所拥有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，每个节点对应一个缓冲区
    map_pointer nstart = map + (map_size - num_nodes) / 2;//中间的缓冲区
    map_pointer nfinish = nstart + num_nodes - 1;//起始缓冲区加偏移量

    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有一些富裕)
        for (cur = nstart; cur <= nfinish; ++cur)
            *cur = allocate_node();//分配一个缓冲区的内存，返回该块缓冲区的头指针，赋值给*cur，因为*cur 就是指向指定缓冲区的头元素的指针
    }
    catch(...) {
        //commit or rollback
        for (map_pointer n = nstart; n < cur; ++n)
            deallocate_node(*n);
        map_allocator::deallocate(map, map_size);
        throw;
    }

    start.set_node(nstart);
    finish.set_node(nfinish);
    start.cur = start.first;
    //如果刚好整除会多配一个节点
    finish.cur = finish.first + num_elements % buffer_size();
}

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();//符合某个条件则必须重新换一个 map
    *(finish.node + 1) = allocate_node();//配置一个新节点
    __STL_TRY {
        construct(finish.cur, t_copy);//在原来缓冲区最后一个位置安置元素
        finish.set_node(finish.node + 1);//改变 finish，令其指向新缓冲区
        finish.cur = finish.first;//设定 finish 状态
    }
```

```
        __STL_UNWIND(deallocate_node(*(finish.node + 1)));
    }

    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();//若符合某种条件必须更换一个 map
        *(start.node - 1) = allocate_node();//配置一个新节点
        __STL_TRY {
            start.set_node(start.node - 1);//改变 start，指向新节点
            start.cur = start.last - 1;//设定 start 状态
            construct(start.cur, t_copy);//针对标的元素设值
        }
        catch(...) {
            start.set_node(start.node + 1);
            start.cur = start.first;
            deallocate_node(*(start.node - 1));
            throw;
        }
    }
```

3、map 什么时候需要重新整治

```
    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) {
            //如果 map 现有空间足够大，将缓冲区调整到中间的位置，移动并不会改变元素相对于缓冲区的偏移量
            new_nstart = map + (map_size - new_num_nodes) / 2
                             + (add_at_front ? nodes_to_add : 0);
            if (new_nstart < start.node)
                //向前移动，注意此处的移动只是改变 map 的指向而已，缓冲区的实际地址没有发生改变(原有元素仍然在原来的地址上)，因此迭代器不会失效，迭代器指向的是元素而非 map，示意图详见 4.4.4
                copy(start.node, finish.node + 1, new_nstart);
            else
                //向后移动
                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;

            map_pointer new_map = map_allocator::allocate(new_map_size);
            //同样，将使用的缓冲区挪到 map 的中间区域，移动并不会改变元素相对于缓冲区的偏移量
            new_nstart = new_map + (new_map_size - new_num_nodes) / 2
                                + (add_at_front ? nodes_to_add : 0);
            copy(start.node, finish.node + 1, new_nstart);
            map_allocator::deallocate(map, map_size);

            map = new_map;
            map_size = new_map_size;
        }

        //重新设置迭代器 start 和 finish
        start.set_node(new_nstart);
        finish.set_node(new_nstart + old_num_nodes - 1);
    }
```

4.4.7. deque 的元素操作：pop_back, pop_front, clear, erase, insert

1、源码

```
    template <class T, class Alloc = alloc, size_t BufSiz = 0>
    class deque {
    ...
    public:
        void pop_back() {
            if (finish.cur != finish.first) {
                //删除最后一个元素后，最后缓冲区有一个或更多元素
                --finish.cur;
                destroy(finish.cur);
            }
            else
                //当前元素是缓冲区的唯一一个元素，删除该元素后需要释放缓冲区
                pop_back_aux();
        }

        void pop_front() {
            if (start.cur != start.last - 1) {
                //第一个缓冲区有两个或更多元素
                destroy(start.cur);
```

```
            ++start.cur;
        }
        else
            //当前元素是缓冲区的唯一一个元素，删除该元素后需要释放
            缓冲区
            pop_front_aux();
    }

public: // Erase
    iterator erase(iterator pos) {
        iterator next = pos;
        ++next;
        difference_type index = pos - start;//清点之前元素个数
        if (index < (size() >> 1)) {//如果清点之前的元素比较少，就移动清楚
        点之前的元素
            copy_backward(start, pos, next);
            pop_front();//移动完毕，最前面的元素冗余，除去
        }
        else {//清除点之后的元素比较少
            copy(next, finish, pos);//移除清楚点之后的元素
            pop_back();//移动完毕，最后一个元素冗余，除去
        }
        return start + index;
    }
public: // Insert

    //返回插入元素的迭代器
    iterator insert(iterator position, const value_type& x) {
        if (position.cur == start.cur) {
            //插入的位置是 deque 的头部
            push_front(x);
            return start;
        }
        else if (position.cur == finish.cur) {
            //插入的位置是 deque 的尾部
            push_back(x);
            iterator tmp = finish;
            --tmp;
            return tmp;
        }
        else {
            //在中间插入，开销会很大
            return insert_aux(position, x);
        }
```

```
    }
    ...
};

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>:: pop_back_aux() {
    deallocate_node(finish.first);//释放掉最后一个缓冲区
    finish.set_node(finish.node - 1);//调整 finish 的状态，使指向上一个缓冲区的最后一个元素
    finish.cur = finish.last - 1;
    destroy(finish.cur);
}

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::pop_front_aux() {
    destroy(start.cur);//将缓冲区第一个(唯一一个)元素析构
    deallocate_node(start.first);//释放第一个缓冲区
    start.set_node(start.node + 1);//调整 start 的状态
    start.cur = start.first;
}
```

2、deque 最初状态(无任何元素)保留一个缓冲区，clear()完成之后恢复初始状态，同样要保留一个缓冲区

```
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::clear() {
    //以下针对头尾意外每一个缓冲区(除了头尾节点一定是饱满的)
    for (map_pointer node = start.node + 1; node < finish.node; ++node) {
        destroy(*node, *node + buffer_size());
        data_allocator::deallocate(*node, buffer_size());
    }

    if (start.node != finish.node) {//至少含有头尾两个缓冲区
        destroy(start.cur, start.last);
        destroy(finish.first, finish.cur);
        data_allocator::deallocate(finish.first, buffer_size());
    }
    else//只有一个缓冲区
        destroy(start.cur, finish.cur);

    finish = start;
}


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

```
deque<T, Alloc, BufSize>::iterator
deque<T, Alloc, BufSize>::erase(iterator first, iterator last) {
    //如果清除区间就是整个 deque，调用 clear 即可
    if (first == start && last == finish) {
        clear();
        return finish;
    }
    else {
        difference_type n = last - first;//清除区间长度
        difference_type elems_before = first - start;//清除区间前方元素个数
        if (elems_before < (size() - n) / 2) {//如果前方元素比较少
            copy_backward(start, first, last);//向后移动前方元素
            iterator new_start = start + n;
            destroy(start, new_start);//析构冗余元素
            //下面将缓冲区释放
            for (map_pointer cur = start.node; cur < new_start.node; ++cur)
                data_allocator::deallocate(*cur, buffer_size());
            start = new_start;//设置 deque 新起点
        }
        else {//如果清除区间后方元素比较少
            copy(last, finish, first);//向前移动后方元素
            iterator new_finish = finish - n;
            destroy(new_finish, finish); //析构冗余元素
            //下面将缓冲区释放
            for (map_pointer cur = new_finish.node + 1; cur <= finish.node;
            ++cur)
                data_allocator::deallocate(*cur, buffer_size());
            finish = new_finish; //设置 deque 新尾点
        }
        return start + elems_before;
    }
}
```

4.5. stack

4.5.1. 概述

1、stack 是一种先进后出(FILO)的数据结构，允许新增元素、移除元素、取得最顶端元素

2、推入的操作称为 push，推出的操作称为 pop

4.5.2. stack 定义完整列表

1、源码如下(已核对)

```
template <class T, class Sequence = deque<T> >
```

```
class stack {
    //以下__STL_NULL_TMPL_ARGS 将展开为<>
    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，由此可见 stack 只是做了一层适配
public:
    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(); }
    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

1、stack 的所有元素都满足先进后出的条件，只有 stack 顶端的元素才有机会被外界取用，stack 不提供走访功能，也不提供迭代器

4.6. queue

4.6.1. queue 概述

queue 是一种先进先出(First In First Out,FIFO)的数据结构，queue 允许新增元素，移除元素，从最底端加入元素、取得最顶端元素。

4.6.2. queue 定义完整列表

1、由于 queue 以底部容器完成其所有工作，而具有这种修改某物接口，形成另

一种风貌，称为 adapter(配接器/适配器)，因此 STL queue 往往不被归类为 container(容器)，而被归类为 container adapter

2、源码如下(已核对)

```
template <class T, class Sequence = deque<T> >
class queue {
    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;
public:
    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(); }
    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

1、queue 所有元素的进出都必须符合"先进先出"的条件，只有 queue 顶端的元素，才有机会被外界取用。queue 不提供遍历功能，也不提供迭代器

4.7. heap

4.7.1. heap 概述

1、heap 并不归属于 STL 容器组件，它是 priority queue 实现的基础，priority queue 允许用户以任何次序将任何元素推入容器内，但取出时一定从优先级最高的元素开始取，binary max heap 正是具有这样的特性，适合作为 priority queue 的底层机制

2、可以使用 list 作为 priority queue 的底层机制

1) 元素插入操作享受常数时间，但是要找到极值，却必须遍历整个 list
2) 或者插入时排序，但是插入的复杂度就过高了

3、以 binary search tree 作为 priority queue 的底层机制，如此一来，元素插入和极值的取得就有 O(lgN)的表现，但是杀鸡用牛刀，并且 birany search tree 实现并不容易

4、binary heap 就是一种 complete binary tree(完全二叉树)，整棵树 binary tree 除了最底层的叶节点之外，是满的，并且叶节点从左到右，且没有空隙。正是由于这种无缝特性，我们可以用 array 来存储树的节点

1) 保留 array[0]，对于节点 i，其左子节点就是 2i，右子节点就是 2i+1(**后面的实现并未采用这种方式**)
2) 对于节点 i，左子节点就是 2i+1，右子节点就是 2i+2

5、如此一来，我们需要的工具就是一个 array 和一组 heap 算法(用来插入元素，删除元素，取极值)。array 的缺点是无法动态改变大小，而 heap 却需要这项功能，因此 vector 是更好的选择

4.7.2. heap 算法

4.7.2.1. push_heap 算法

1、源码如下(已核对)

```
template <class RandomAccessIterator, class Compare>
inline void push_heap(RandomAccessIterator first, RandomAccessIterator last,
                      Compare comp) {
    //萃取元素类型以及距离类型
    __push_heap_aux(first, last, comp, distance_type(first), value_type(first));
}

template <class RandomAccessIterator, class Compare, class Distance, class T>
inline void __push_heap_aux(RandomAccessIterator first,
                            RandomAccessIterator last, Compare comp,
                            Distance*, T*) {
    //最后一个元素的洞号：(last-1)-first
    //树根洞号：0
    __push_heap(first, Distance((last - first) - 1), Distance(0),
    T(*(last - 1)), comp);
}

//下面这个函数保证，当 holeIndex 为根的子数以满足堆性质(由于只会向上
```

与父节点交换，这就要求 value 的值必须比其左右孩子具有更高优先级，也就是以 holeIndex 为根的子树已满足堆性质)，在 holeIndex 插入值 value 能维护堆性质(最后插入的洞号未必是给定的 holeIndex)

```
template <class RandomAccessIterator, class Distance, class T, class Compare>
void __push_heap(RandomAccessIterator first, Distance holeIndex,
                 Distance topIndex, T value, Compare comp) {
    Distance parent = (holeIndex - 1) / 2;//找出父节点
    while (holeIndex > topIndex && comp(*(first + parent), value)) {
        //当尚未到达顶端(一定存在父节点)，并且父节点小于新值(于是不符合 heap 的次序特性)
        *(first + holeIndex) = *(first + parent);//令洞值为父值
        holeIndex = parent;//调整洞号
        parent = (holeIndex - 1) / 2;//新洞的父节点
    }
    *(first + holeIndex) = value;//令洞值为新值，完成插入操作
}
```

4.7.2.2. pop_heap 算法

1、源码如下(已核对)

```
template <class RandomAccessIterator>
inline void pop_heap(RandomAccessIterator first, RandomAccessIterator last) {
    //先萃取出元素类型
    __pop_heap_aux(first, last, value_type(first));
}

template <class RandomAccessIterator, class T>
inline void __pop_heap_aux(RandomAccessIterator first,
                           RandomAccessIterator last, T*) {
    //将 heap 边界减少 1
    __pop_heap(first, last - 1, last - 1, T(*(last - 1)), distance_type(first));
}

template <class RandomAccessIterator, class T, class Compare, class Distance>
inline void __pop_heap(RandomAccessIterator first, RandomAccessIterator last,
                       RandomAccessIterator result, T value, Compare comp,
                       Distance*) {
    //将头元素的值放到 result 处，而 result 处的元素就是参数 value
    *result = *first;
    //注意，边界已经减少了 1
    __adjust_heap(first, Distance(0), Distance(last - first), value, comp);
}
```

//下面的方法保证给指定洞号填上给定值时，满足堆的性质(最后填值得洞号未必是给定的洞号，会调整)，并且有个前提，holeIndex 以后的节点(不包括

```
holeIndex)以满足堆性质
//这个方法有更好的实现形式(算法导论上的那种方式)，下面这种方式太啰嗦，而且复杂度也高
template <class RandomAccessIterator, class Distance, class T, class Compare>
void __adjust_heap(RandomAccessIterator first, Distance holeIndex,
                   Distance len, T value, Compare comp) {
    Distance topIndex = holeIndex;
    Distance secondChild = 2 * holeIndex + 2;//右孩子
    //以下过程一定会将 holeIndex 调整为某个叶节点的洞号，以下的 while 循环与要安放的值 value 无关
    //为什么要将洞号调整为叶节点的洞号：因为只有是叶节点时，调用 push_aux 才是无害的，否则必须保证以洞号 holeIndex 为根的子树以满足堆性质，即 value 比 holeIndex 的左右孩子具有更高优先级
    while (secondChild < len) {
        //如果左孩子更'大'
        if (comp(*(first + secondChild), *(first + (secondChild - 1))))
            secondChild--;
        *(first + holeIndex) = *(first + secondChild);//将洞值更新为两个孩子中的较大值
        holeIndex = secondChild;//更新洞号，哪个孩子更大就更新成哪个
        secondChild = 2 * (secondChild + 1);//更新右子洞号
    }
    if (secondChild == len) {//如果没有右子节点，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;//此时一定是叶
    }
    //将待填值得洞号调整到叶节点后，在调用下面的函数，向上找到合适的洞号将 value 填入
    __push_heap(first, holeIndex, topIndex, value, comp);
}
```

2、pop_heap 之后，最大元素只是被置于底部容器的最尾端，尚未被取走，若要取走其值，可使用底部容器(vector)所提供的 back()操作函数，如果要移走它，可使用底部容器(vector)所提供的 pop_back()操作函数

4.7.2.3. sort_heap 算法

1、既然每次 pop_heap 可获得 heap 中建值最大的元素，如果持续对整个 heap 做 pop_heap 操作，每次将操作范围从后向前缩减一个元素，当整个程序执行完毕时，便有了一个递增序列

2、源码(已核对)

```
template <class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp) {
    while (last - first > 1) pop_heap(first, last--, comp);
```

```
}
```

4.7.2.4. make_heap 算法

1、源码(已核对)

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

template <class RandomAccessIterator, class Compare, class T, class Distance>
void __make_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp, T*, Distance*) {
    if (last - first < 2) return;
    Distance len = last - first;
    Distance parent = (len - 2)/2;

    while (true) {
        __adjust_heap(first, parent, len, T(*(first + parent)), comp);
        if (parent == 0) return;
        parent--;
    }
}
```

4.7.2.5. 总结：真的反人类

4.7.3. heap 没有迭代器

4.8. priority_queue

4.8.1. priority_queue 概述

1、priority_queue 是一个拥有权值概念的 queue，它允许加入新元素，移出旧元素，审视元素值等功能

2、priority_queue 带有权值观念，其内的元素并非依照被推入的次序排列，而是自动依照元素的权值排列，权值最高者，排在最前面

3、缺省情况下 priority_queue 利用一个 max_heap 完成，后者是一个以 vector 表现的 complete binary tree

4.8.2. priority_queue 定义完整列表

1、queue 以底部容器完成所有工作，具有这种"修改某物接口，形成另一种风貌"，称为 adapter(配接器)，因此 STL priority_queue 往往不被归类为 container(容器)，而被归类为 container adapter

2、源码(已核对)

```
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) {}

    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 {
            c.push_back(x);
            push_heap(c.begin(), c.end(), comp);
        }
        __STL_UNWIND(c.clear());
    }
    void pop() {
        __STL_TRY {
            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

Chapter 5. 关联式容器

1、容器可概分为序列式(sequence)和关联式(associative)两种
2、STL 关联式容器分为 set 集合和 map 两大类，以及这两大类的衍生体 multiset(多键集合)和 multimap(多键映射表)，这些容器的底层机制均以红黑树完成，RB-tree 也是一个独立的容器，但并不开放给外界使用
3、此外，SGI STL 还提供了一个不在标准规格之列的关联式容器：hash table(散列表)以及以此 hash table 为底层机制而完成的 hash_set(散列集合)、hash_map(散列映射表)、hash_multiset(散列多键集合)、hash_multimap(散列多键映射表)
4、序列式容器

1) array(build-in)
2) vector
3) heap(以算法形式呈现)
4) priority-queue
5) list
6) slist(非标准)
7) deque
8) stack(adapter)
9) queue(adapter)

5、关联式容器

1) RB-tree(非公开)
2) set
3) map
4) multiset
5) multimap
6) hashtable(非标准)
7) hash_set(非标准)
8) hash_map(非标准)
9) hash_multiset(非标准)
10) hash_multimap(非标准)

6、关联式容器

- 概念上类似关联式数据库：每笔数据(每个元素)都有一个键值(key)和一个实值(value)
- 当元素被插入到关联式容器中时，容器内部结构(RB-tree 或 hash-table)便依照其键值大小，以某种特定规则将这个元素放置于适当位置
- 关联式容器没有所谓头尾(只有最大元素或最小元素)，因此不会有 push_back()、push_fron()、pop_back()、pop_fron()、begin()、end()这样的操作行为

7、一般而言，关联式容器的内部结构是一个 balanced binary tree 平衡二叉树，以便获得良好的搜寻效率，balanced binary tree 有许多类型，包括 AVL-tree、RB-tree、AA-tree，其中最被广泛运用于 STL 的是 RB-tree

5.1. 树的导览

1、树(tree)，在计算机科学里，是一种非常基础的数据结构
   1) 几乎所有操作系统都将文件存放在树状结构里
   2) 几乎所有编译器都需要实现一个表达时树(expression tree)
   3) 文件压缩所用的哈夫曼算法(Huffman's Alogrithm)需要用到树状结构；数据库使用的 B-tree 则是一种相当复杂的树状结构(**我觉得 B 树比红黑树容易理解的多**)

2、树的概念
   1) 树由节点(nodes)和边(edges)构成
   2) 最上端的节点称为根节点(root)
   3) 每个节点可以拥有具方向性的边(directed edges)，用来和其他节点相连，相连节点之中，在上者称为父节点(parent)，在下者称为子节点(child)。无子节点称为叶节点(leaf)
   4) 不同节点如果拥有相同父节点，则彼此互为兄弟节点(siblings)
   5) 根节点到任意节点之间有唯一路径，路径所经过的边数，称为路径长度，根节点至任意节点的路径长度，即所谓该节点的深度。根节点的深度永远是 0，**某节点至其最深子节点(叶节点)的路径长度，称为该节点的高度(height)**

5.1.1. 二叉搜索树

1、所谓二叉树(binary tree)，其意义是：任何节点最多只允许有两个子节点，这两个子节点称为左子节点和右子节点

2、所谓二叉搜索树(binary search tree)，可提供对数时间(logarithmic time)的元素插入和访问
   1) 二叉搜索树的节点放置规则是：任何节点的键值一定大于其左子树中的每一个节点的键值，并小于其右子树中的每一个节点的键值

3、二叉搜索树的删除
   1) 如果被删除节点 A 最多只有一个子节点，那么该子节点移动到被删除节点处即可
   2) 如果有两个子节点，那么找到以被删除节点 A 为根的子树中的最小节点 B，将其抽出(若该节点 B 有右孩子，将其右孩子移动到节点 B 处)，并将其置于被删除节点 A 处

5.1.2. 平衡二叉搜索树

1、也许因为输入值不够随机，也许因为经过某些插入或删除操作，二叉搜索树可能会失去平衡，造成寻找效率低落的情况(可能退化为链表)

2、所谓树平衡与否，并没有一个绝对的衡量标准，平衡的大致意思是：没有任何一个节点过深，不同的平衡条件，造就出不同的效率表现，以及不同的实现复杂度。有数种特殊结构如 **AVL-tree、RB-tree、AA-tree**，均可实现平衡二叉搜索树，它们都比一般的(无法绝对维持平衡)二叉搜索树复杂，因此，插入节点和删除节点的平均时间也较长，但是它们可以避免极难应付的最坏(高度不平衡)情况

5.1.3. AVL tree(Adelson-Velskii-Landis tree)

1、AVL tree 是一个加上"加上额外平衡条件"的二叉搜索树，其平衡条件的建立是为了确保整棵树的深度为 O(lgN)。直观上的最佳平衡条件是每个节点的左右子树拥有相同的高度，但这未免太过严苛，很难插入新元素同时保持这样的平衡条件

2、AVL tree 退而求其次，要求任何节点的左右子树高度相差最多 1，这是一个较弱的条件，但是仍能够保证**"对数深度"平衡状态**

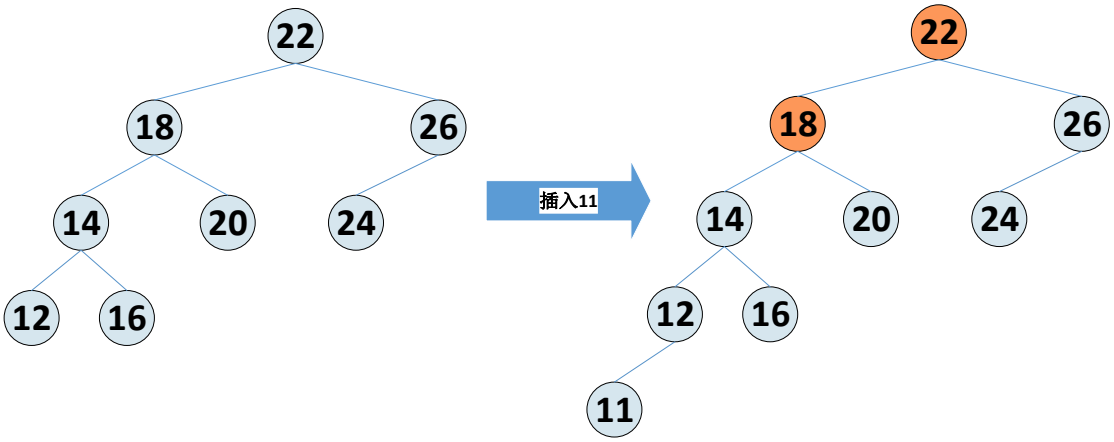
3、**由于只有"插入点至根节点"路径上各节点可能改变平衡状态，因此只需要调整其中最深的那个节点，便可使整棵树重新获得平衡**

4、由于只要调整"插入点至根节点"路径上，平衡状态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状态。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"平衡被破坏"意味着 X 的左右两棵子树的高度相差 2，因此我们可以轻易地将情况分为四种

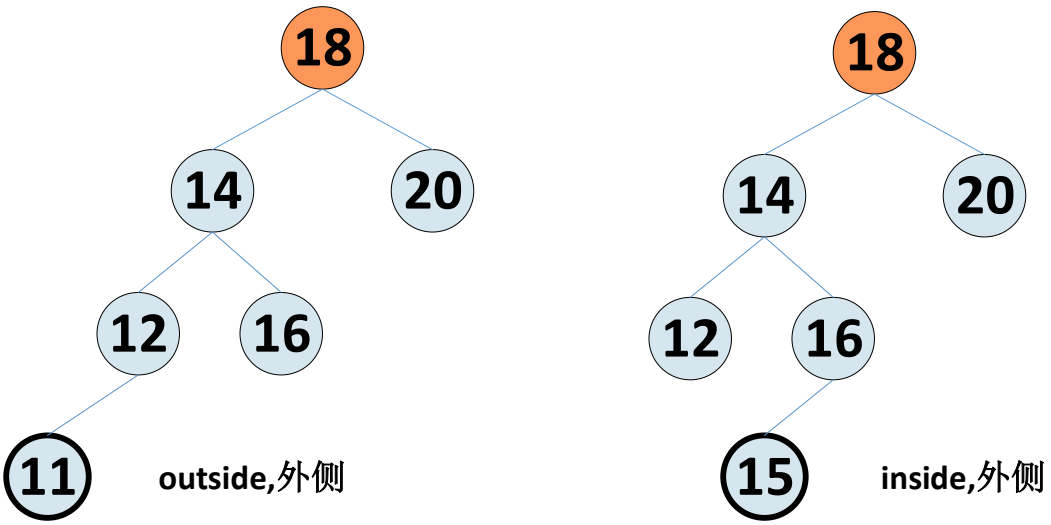
1) 插入点位于 X 的左子节点的左子树--左左
2) 插入点位于 X 的左子节点的右子树--左右
3) 插入点位于 X 的右子节点的左子树--右左
4) 插入点位于 X 的右子节点的右子树--右右

- 情况 1,4 彼此对称，称为外侧(outside)插入，可以采用单旋转操作(singlerotation)调整解决
- 情况 2,3 彼此对称，称为内侧(inside)插入，可以采用双旋转操作(double rotation)调整解决

5、图示详解



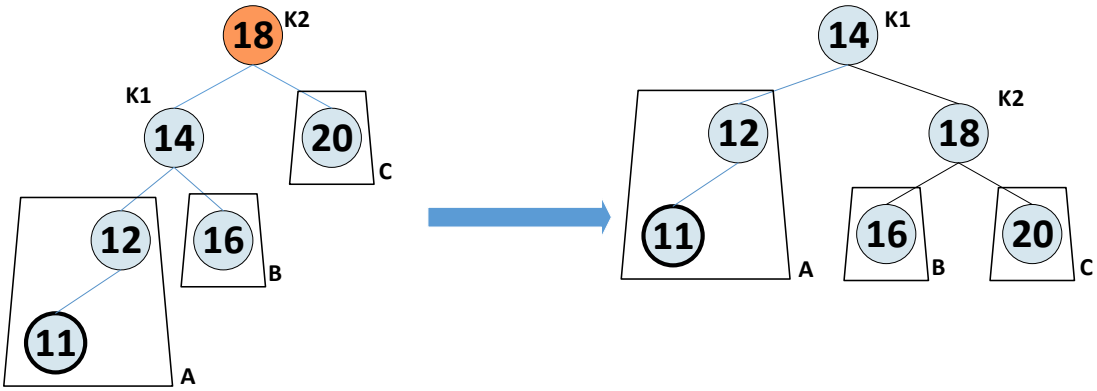
- 橘红色节点：违反 AVL tree 规则的节点，最深的节点是 18，因此只需要调整该节点，那么便可使得整棵树重新得到平衡状态



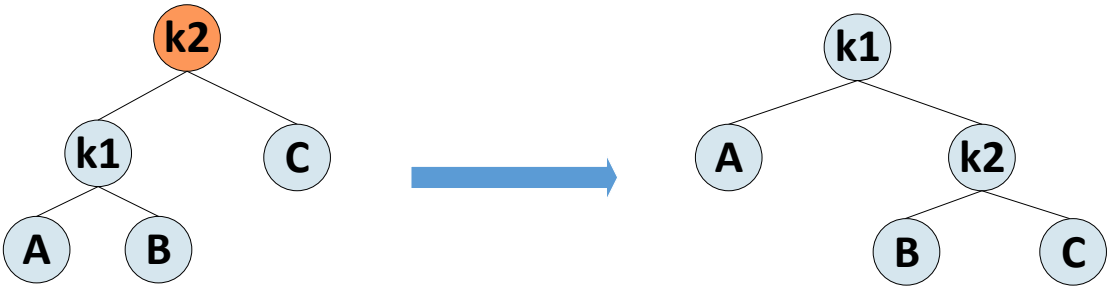
1) 插入点在 X 左子节点的左子树(左左)对称于插入点在 X 右子节点的右子树(右右)
2) 插入点在 X 左子节点的右子树(左右)对称于插入点在 X 右子节点的左子树(右左)

5.1.4. 单旋转(Single Rotation)

1、在外侧插入状态中，k2"插入前平衡，插入后不平衡"的唯一情况如下图



2、进一步抽象成如下状态



- $H_A+1=H_C+2$，即 $H_C=H_A-1$
- 必定满足 $H_B<H_A$，否则 k2 在插入前就不处于平衡状态($H_B+1=H_C+2$)
- 必定满足 $H_B+2>H_A$，否则第一违反平衡的节点是 k1，于是 $H_B=H_A-1$

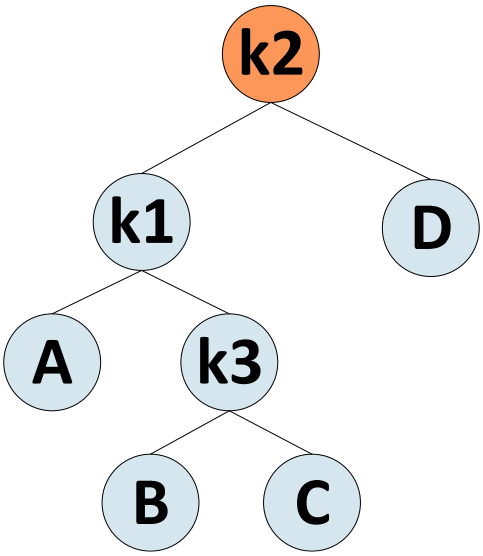
3、为了调整平衡状态，我们希望将 A 子树提高一层，并将 C 子树下降一层

1) 基于上述分析，B，C 子树树高相同，因此旋转后 k2 节点不会有问题

2) 且旋转后 A 子树和 K2 子树树高也相同

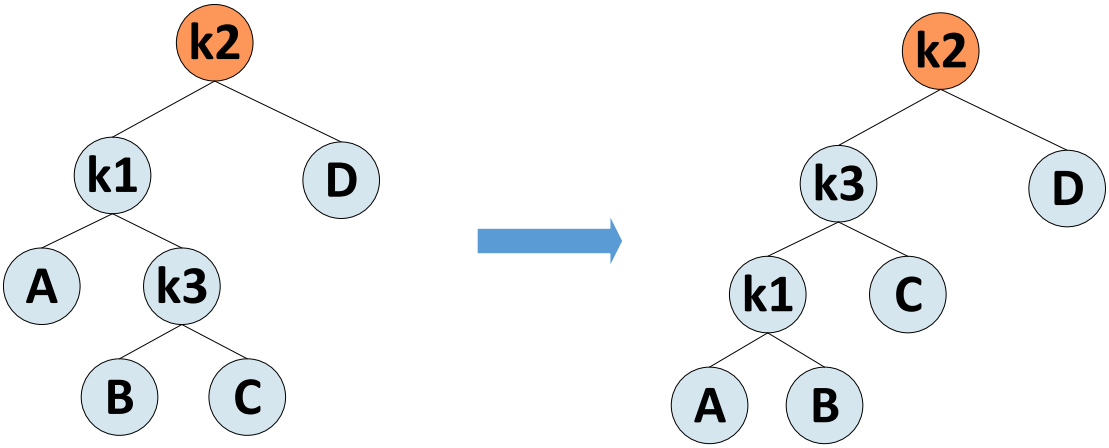
5.1.5. 双旋转

1、在内侧插入导致不平衡的情况如下

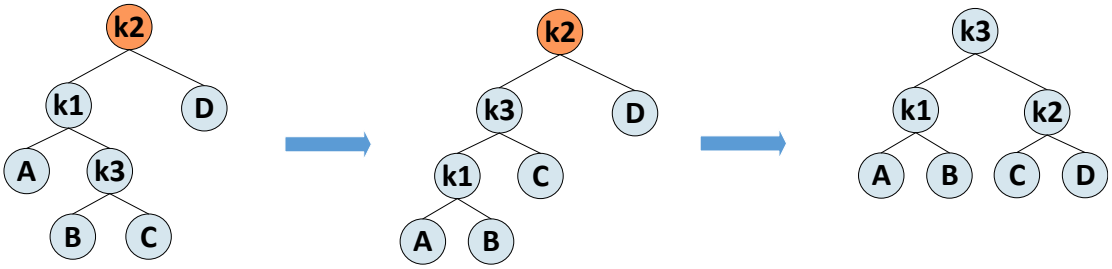


1) $H_{k3}+1=H_D+2$，即 $H_D=H_{k3}-1$
2) $H_A=H_{k3}-1$
3) H_B、H_C 都为 $H_{k3}-1$ 或一个为 $H_{k3}-1$ 另一个为 $H_{k3}-2$
4) **k2 在插入之前的高度为 H_D+2**

2、以 k1 为根节点进行一次左旋，旋转后如下



3、以 k2 为根节点再进行一次右旋，旋转后如下



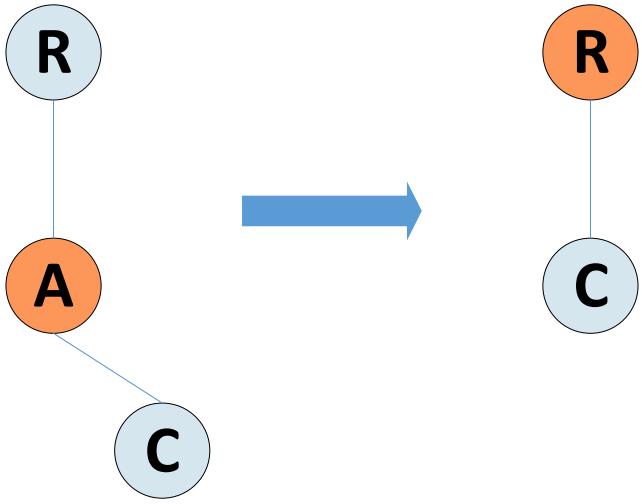
1) 对于 k1 节点，满足平衡条件，因为 H_B 要么与 H_A 相同，要么比 H_A 少 1
2) 对于 k2 节点，满足平衡条件，因为 H_C 要么与 H_D 相同，要么比 H_D 少 1

3) 对于 k3 节点，满足平衡条件，因为 H_{k1} 高度与 H_{k2} 相同
4) **在平衡之后 k3 节点的高度是 H_D+2，因此不会造成 k3 父节点的不平衡，因此只需要旋转两次即可终止**

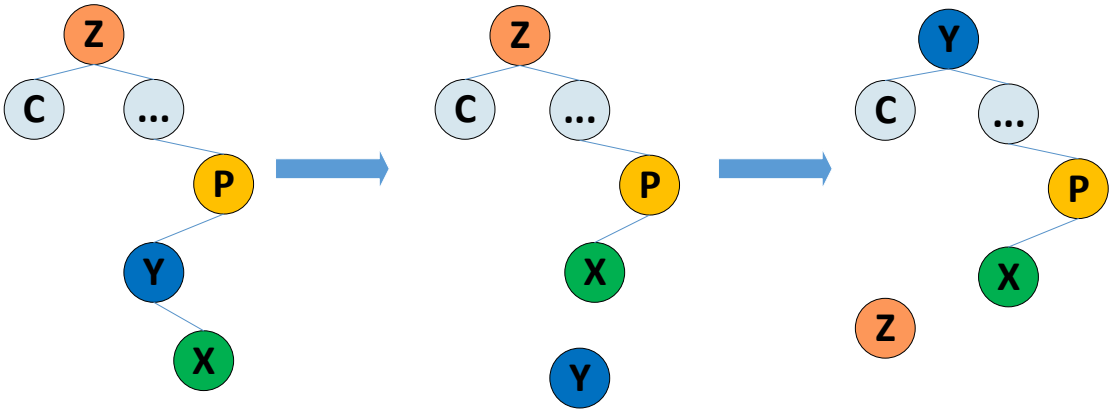
5.1.6. AVL 树删除节点

1、当被删除节点最多有一个孩子节点，被删除后节点 R 的高度可能改变

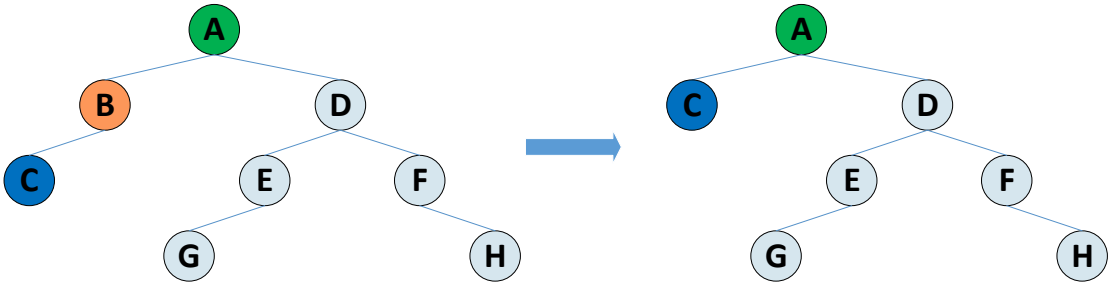
- 若节点 R 不平衡，那么旋转使其平衡，并且更新高度，若新高度与原告度不同，继续向上传递
- 若新高度与原高度相同，那么终止退出即可，整棵树已经平衡



2、被删除节点 A 有两个孩子节点，找出右子树中的最小节点 B，将其取出，代替 A 节点， 那么被删除的节点就变成了 B，转化为第一种情况



1) Z 节点为被删除的节点
2) Y 节点为 Z 节点的右子树中的最小节点，X 节点为 Y 节点的右孩子(可能为哨兵节点)
3) P 节点为 Y 节点的父节点
4) 从 P 节点到根节点都可能出现平衡被破坏的情况
5) 而且可能出现如下一种情况，这种情况在插入时是不可能出现的



- A 节点的平衡性被破坏了，并且 D 的两颗子树高度相同，插入时是不可能相同的，必定一高一低
- 这种情况应该被归纳到左左或者右右中去，即一次旋转即可调整到平衡状态

5.2. RB-tree(红黑树)

1、红黑树有如下性质

1) 每个节点不是红色就是黑色
2) 根节点为黑色
3) 如果节点为红色，其叶节点必定为黑色
4) 任一节点至叶节点(nil)的任何路径，所含的黑节点数量必须相同
5) 叶节点 nil 为黑色

2、根据规则 4，新增节点必须为红色；根据规则 3，新增节点的父节点必须是黑色。当新增节点根据二叉搜索树的规则到达插入点时，如果未能满足上述条件，就必须调整颜色并旋转树形

5.2.1. 插入节点

1、为了方便讨论，首先定义一些代名

1) X：新节点
2) P：新节点 X 的父节点
3) G：新节点 X 的祖父节点
4) S：新节点 X 的伯父节点
5) GG：新节点 X 的曾祖父节点

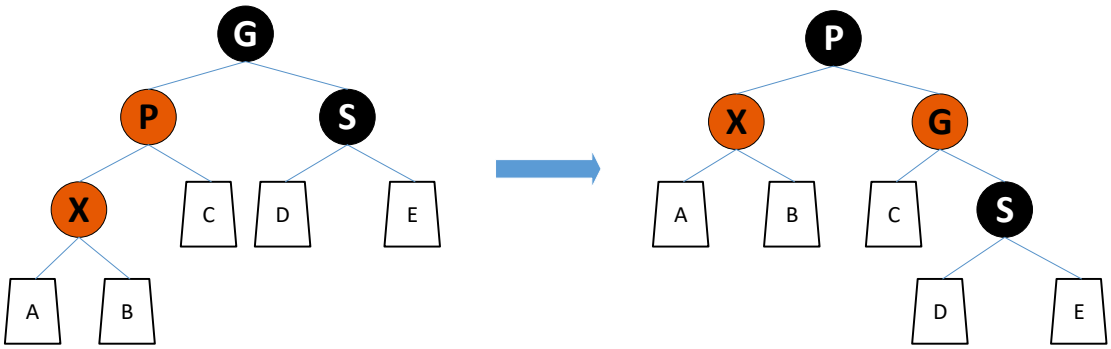
2、根据红黑树的规则

1) 新节点 X 必为叶节点，且为红色

3、根据 X 的插入位置以及外围节点的颜色，有了如下四种考虑

5.2.1.1. 情况 1

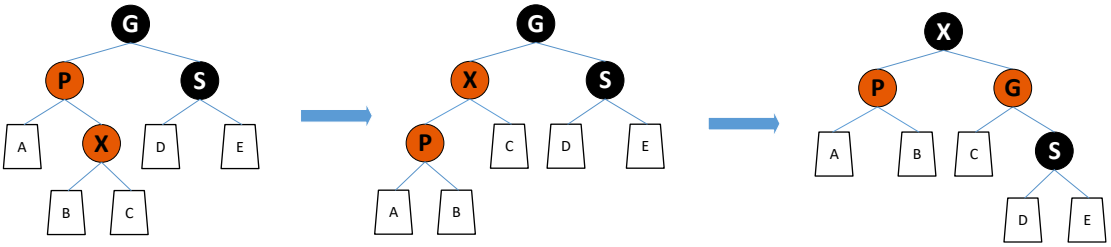
1、S 为黑且 X 为外侧插入，如下图



2、对此情况，我们先对 P、G 做一次但旋转，并更改 PG 颜色，即可重新满足红黑树的规则

5. 2. 1. 2. 情况 2

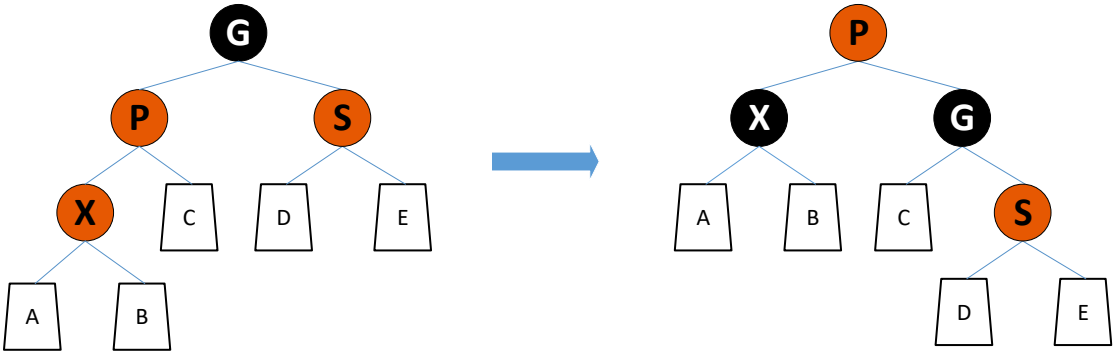
1、S 为黑色且 X 内侧插入，如下图



2、对此情况，我们必须先对 P，X 做一次单旋转，然后更改 G，X 颜色，再将结果对 G 做一次单旋转

5. 2. 1. 3. 情况 3

1、S 为红色且 X 为外侧插入，如下图



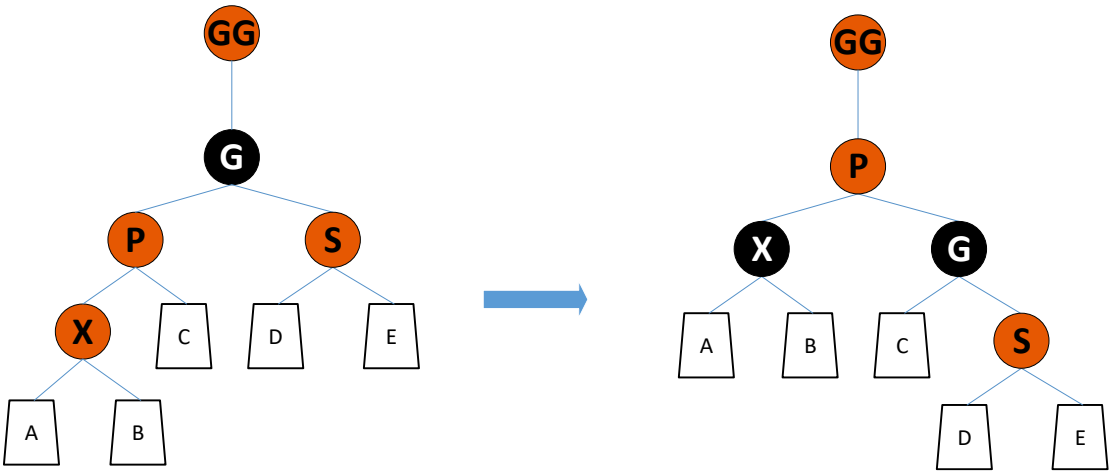
2、对此情况，先对 P 和 G 做一次单旋转，并改变 X 的颜色

1) 如果 GG 为黑色，则直接满足红黑树的所有性质
2) 如果 GG 为红色，则转为情况 4

3、算法导论上此种情况的处理方式与此处介绍的不一致，并且这种情况 3 与情况 4 好像是一样的

5. 2. 1. 4. 情况 4

1、S 为红色，且 X 为外侧插入，如下图



2、对此情况，先对 P 和 G 做一次单旋转，并改变 X 的颜色，如果此时 GG 还是为红色，得继续往上做，直到不再有父子连续为红色
3、算法导论上此种情况的处理方式与此处介绍的不一致，并且这种情况 3 与情况 4 好像是一样的
4、少了一种分析，当 S 为红色，且 X 为内侧插入
5、更偏向于算法导论那种做法

5.2.2. 一个由上而下的程序

1、为了避免状况 4：父子节点皆为红色的情况持续向 RB-tree 的上层结构发展，形成处理时效上的瓶颈，我们可以施行一个由上而下的程序

1) 假设新增节点为 A，那么我们就沿着 A 的路径，只要看到某节点 X 的两个子节点皆为红色，就把 X 改为红色，并把两个子节点改为黑色

2、如此一来插入操作变得只剩下情况 1 和情况 2，只需要 1 或 2 次旋转操作即可维护红黑树的性质

5.2.3. 节点删除

1、<未完成>：如何自上而下？

5.2.4. RB-tree 的节点设计

1、为了具有更大的弹性，节点分为两层
2、源码(已核对)(stl_tree.h)

```
typedef bool __rb_tree_color_type;
const __rb_tree_color_type __rb_tree_red = false;//红色为 0
const __rb_tree_color_type __rb_tree_black = true;//黑色为 1

struct __rb_tree_node_base
{
    typedef __rb_tree_color_type color_type;
    typedef __rb_tree_node_base* base_ptr;

    color_type color;//节点颜色
    base_ptr parent;//RB 树的许多操作，必须知道父节点
```

```
        base_ptr left;//指向左节点
        base_ptr right;//指向右节点

        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;
    };
```

5.2.5. RB-tree 的迭代器

1、要成功地将 RB-tree 实现成为一个泛型容器，迭代器的设计师一个关键。首先要考虑他的类别(category)，然后要考虑它的前进(increment)、后退(decrement)、提领(dereference)、成员访问(member access)等操作

2、为了更大的弹性，SGI 将 RB-tree 迭代器实现分为两层，这种设计理念和 slist 类似，其主要关系如下

1) __rb_tree_node 继承自__rb_tree_node_base
2) __rb_tree_iterator 继承自__rb_tree_base_iterator

3、RB-tree 迭代器属于双向迭代器，但不具备随机定位能力，其提领操作和成员访问操作与 list 十分相近，较为特殊的是前进和后退操作

- RB-tree 迭代器的前进操作 operator++()调用了基层迭代器的 increment()
- RB-tree 迭代器的后退操作 operator--()调用了基层迭代器的 decrement()
- 前进和后退的行为完全依据二叉树的节点排列法则

4、基层迭代器源码如下(已核对)(stl_tree.h)

```
    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;//它用来与容器之间产生一个连接关系
```

```
//其实可以实现与 operator++内，因为再无他处会调用次函数了
void increment()
{
	if (node->right != 0) {
		//情况 1 如果有右子节点，那么后继就是右子树的最小节点
		node = node->right;
		while (node->left != 0)
			node = node->left;
	}
	else {//情况 2 如果没有右子树，那么向上找到第一次满足如下性质(某节点是其父节点的左孩子)的节点，那么这个父节点就是后继
		base_ptr y = node->parent;
		while (node == y->right) {
			node = y;
			y = y->parent;
		}
		if (node->right != y)
			//情况 3 此时右子节点不等于此时的父节点，此时父节点为后继
			node = y;
		//情况 4
		//以上判断"若此时的右子节点不等于此时的父节点"，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节点，以上做法必须配合 RB-tree 根节点与特殊 header 之间的特殊关系
	}
}
//其实可以实现与 operator--内，因为再无他处会调用次函数了
void decrement()
{
	if (node->color == __rb_tree_red &&
	node->parent->parent == node)
		//情况 1 以上情况发生于 node 为 header 时，亦即 node 为 end()时，注意 header 之右子节点即 mostright，指向整棵树的 max 节点
		node = node->right;
	else if (node->left != 0) {
		//情况 2 如果有左子节点，那么前继就是左子树的最大节点
		base_ptr y = node->left;
		while (y->right != 0)
			y = y->right;
		node = y;
	}
```

```
            else {
                //情况 3 如果非根节点也没有左子树，那么向上找到第一次
                满足如下性质(某节点是其父节点的右孩子)的节点，那么这
                个父节点就是前继
                base_ptr y = node->parent;
                while (node == y->left) {
                    node = y;
                    y = y->parent;
                }
                node = y;
            }
        }
    };
```

5、上层迭代器源码如下(已核对)(stl_tree.h)

```
    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;

        __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; }
    #ifndef __SGI_STL_NO_ARROW_OPERATOR
        pointer operator->() const { return &(operator*()); }
    #endif /* __SGI_STL_NO_ARROW_OPERATOR */

        self& operator++() { 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;
    }
};
```

5.2.6. RB-tree 的数据结构

1、源码如下(已核对)(stl_tree.h)

```
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(); }
    void put_node(link_type p) { rb_tree_node_allocator::deallocate(p); }

    link_type create_node(const value_type& x) {
        link_type tmp = get_node();//配置空间
        __STL_TRY {
            construct(&tmp->value_field, x);//构造内容
        }
        __STL_UNWIND(put_node(tmp));// commit or rollback
        return tmp;
    }

    link_type clone_node(link_type x) {//复制一个节点(的值和颜色)
        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) {
        destroy(&p->value_field);//析构内容
        put_node(p);//释放内存
    }

protected:
    size_type node_count; //追踪记录树的大小
    link_type header;//这是实现上的一个技巧
    Compare key_compare;//节点间的键值大小比较准则

    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; }

    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); }

    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_field; }
    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);
    }

    static link_type minimum(link_type x) {
        return (link_type)  __rb_tree_node_base::minimum(x);
    }
    static link_type maximum(link_type x) {
        return (link_type) __rb_tree_node_base::maximum(x);
    }

public:
    typedef __rb_tree_iterator<value_type, reference, pointer> iterator;
```

```
    typedef __rb_tree_iterator<value_type, const_reference, const_pointer>
                                        const_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();//产生一个节点空间，令 header 指向它
        color(header) = __rb_tree_red; //令 header 为红色，用来区分 header
        和 root，在 iterator.operator--之中
        root() = 0;
        leftmost() = header;//令 header 的左子节点为自己
        rightmost() = header;//令 header 的右子节点为自己
    }
public:
    // allocation/deallocation
    rb_tree(const Compare& comp = Compare())
        : node_count(0), key_compare(comp) { init(); }

    rb_tree(const rb_tree<Key, Value, KeyOfValue, Compare, Alloc>& x)
        : node_count(0), key_compare(x.key_compare)
    {
        header = get_node();
        color(header) = __rb_tree_red;
        if (x.root() == 0) {
            root() = 0;
            leftmost() = header;
            rightmost() = header;
        }
        else {
            __STL_TRY {
                root() = __copy(x.root(), header);
            }
            __STL_UNWIND(put_node(header));
            leftmost() = minimum(root());
            rightmost() = maximum(root());
        }
        node_count = x.node_count;
    }
    ~rb_tree() {
        clear();
        put_node(header);
    }
```

```
        rb_tree<Key, Value, KeyOfValue, Compare, Alloc>&
        operator=(const rb_tree<Key, Value, KeyOfValue, Compare, Alloc>& x);

public:
        // accessors:
        Compare key_comp() const { return key_compare; }
        iterator begin() { return leftmost(); }//RB 树的起头为最左(最小)节点处
        const_iterator begin() const { return leftmost(); }
        iterator end() { return header; }//RB 树的终点为 header 所指处
        const_iterator end() const { return header; }
        reverse_iterator rbegin() { return reverse_iterator(end()); }
        const_reverse_iterator rbegin() const {
                return const_reverse_iterator(end());
        }
        reverse_iterator rend() { return reverse_iterator(begin()); }
        const_reverse_iterator rend() const {
                return const_reverse_iterator(begin());
        }
        bool empty() const { return node_count == 0; }
        size_type size() const { return node_count; }
        size_type max_size() const { return size_type(-1); }

        void swap(rb_tree<Key, Value, KeyOfValue, Compare, Alloc>& t) {
                __STD::swap(header, t.header);
                __STD::swap(node_count, t.node_count);
                __STD::swap(key_compare, t.key_compare);
        }

public:
        // insert/erase
        pair<iterator,bool> insert_unique(const value_type& x);//保持节点唯一
        iterator insert_equal(const value_type& x);//允许节点重复的插入操作

        iterator insert_unique(iterator position, const value_type& x);
        iterator insert_equal(iterator position, const value_type& x);

        template <class InputIterator>
        void insert_unique(InputIterator first, InputIterator last);
        template <class InputIterator>
        void insert_equal(InputIterator first, InputIterator last);

        void erase(iterator position);
        size_type erase(const key_type& x);
        void erase(iterator first, iterator last);
```

```
        void erase(const key_type* first, const key_type* last);
        void clear() {
            if (node_count != 0) {
                __erase(root());
                leftmost() = header;
                root() = 0;
                rightmost() = header;
                node_count = 0;
            }
        }

public:
        // set operations:
        iterator find(const key_type& x);
        const_iterator find(const key_type& x) const;
        size_type count(const key_type& x) const;
        iterator lower_bound(const key_type& x);
        const_iterator lower_bound(const key_type& x) const;
        iterator upper_bound(const key_type& x);
        const_iterator upper_bound(const key_type& x) const;
        pair<iterator,iterator> equal_range(const key_type& x);
        pair<const_iterator, const_iterator> equal_range(const key_type& x) const;

public:
        // Debugging
        bool __rb_verify() const;
};

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;
        link_type x = root();//从根节点开始
        while (x != 0) {//从根节点开始，往下寻找适当的插入点
            y = x;
            //遇大往左，遇小往右
            x = key_compare(KeyOfValue()(v), key(x)) ? left(x) : right(x);
        }
        return __insert(x, y, v);
        //以上，x 为新值插入点，y 为插入点之父节点，v 为新值
}

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();//从根节点开始
    bool comp = true;
    while (x != 0) {//从根节点开始，往下寻找适当的插入点
        y = x;
        comp = key_compare(KeyOfValue()(v), key(x));
        //为什么不在这里进行相等性判断???这样效率会更高的
        //遇大往左，遇小往右
        x = comp ? left(x) : right(x);
    }
    //明确 key_compare 函数当 x1>x2 时返回 true，当 x1<=x2 时返回 false
    //离开循环后，y 所指即插入点之父节点(此时它必为叶节点)
    iterator j = iterator(y);
    if (comp)//如果离开循环时，comp 为真，即表示遇大(v 严格小于父节
    点的关键字)，将插入于左侧
        if (j == begin())//如果插入节点之父节点为最左节点
            return pair<iterator,bool>(__insert(x, y, v), true);
        else//否则(插入节点之父节点不为最左节点)
            --j;//找到前继节点
    //这个判断处理的是
      1)  comp 返回 false，即 v>=j.key
      2)  comp 返回 true，向上找到第一个满足性质(该节点是其父节点的
          右孩子，因为只有向右走的过程会发生相等)的节点，而上一步
          的--j 操作会找到这个节点
    if (key_compare(key(j.node), KeyOfValue()(v)))
        //新键值不与既有节点之键值重复，于是执行安插操作
        return pair<iterator,bool>(__insert(x, y, v), true);
    //新值一定与树中键值重复，那么不该插入新值，并返回与新值相等
    值得迭代器
    return pair<iterator,bool>(j, false);
}

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) {
    link_type x = (link_type) x_;
    link_type y = (link_type) y_;
    link_type z;
```

```
    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()=x
        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;//设定新节点的父节点
    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);
    ++node_count;//节点数累加
    return iterator(z);//返回一个迭代器，指向新增节点
}

inline void
__rb_tree_rebalance(__rb_tree_node_base* x, __rb_tree_node_base*& root)
{
    x->color = __rb_tree_red;
    while (x != root && x->parent->color == __rb_tree_red) {
        if (x->parent == x->parent->parent->left) {
            __rb_tree_node_base* y = x->parent->parent->right;
            if (y && y->color == __rb_tree_red) {
                x->parent->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->right) {
                    x = x->parent;
                    __rb_tree_rotate_left(x, root);
                }
```

```
                x->parent->color = __rb_tree_black;
                x->parent->parent->color = __rb_tree_red;
                __rb_tree_rotate_right(x->parent->parent, root);
            }
        }
        else {
            __rb_tree_node_base* y = x->parent->parent->left;
            if (y && y->color == __rb_tree_red) {
                x->parent->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) {
                    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);
            }
        }
    }
    root->color = __rb_tree_black;
}

inline void
__rb_tree_rotate_left(__rb_tree_node_base* x, __rb_tree_node_base*& root)
{
    __rb_tree_node_base* y = x->right;
    x->right = y->left;
    if (y->left !=0)
        y->left->parent = x;
    y->parent = x->parent;

    if (x == root)
        root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->left = x;
    x->parent = y;
```

```
}

inline void
__rb_tree_rotate_right(__rb_tree_node_base* x, __rb_tree_node_base*&
root)
{
    __rb_tree_node_base* y = x->left;
    x->left = y->right;
    if (y->right != 0)
        y->right->parent = x;
    y->parent = x->parent;

    if (x == root)
        root = y;
    else if (x == x->parent->right)
        x->parent->right = y;
    else
        x->parent->left = y;
    y->right = x;
    x->parent = y;
}
```

5.2.7. RB-tree 的构造与内存管理

1、下面是 RB-tree 所定义的专属空间配置器 rb-tree_node_allocator，每次恰恰配置一个节点，使用 simple_alloc<>定义于第二章

2、源码如下(已核对)(stl_tree.h)

```
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;
    ...

    void init() {
        header = get_node();//产生一个节点空间，令 header 指向它
        color(header) = __rb_tree_red; //令 header 为红色，用来区分 header
        和 root，在 iterator.operator--之中
        root() = 0;
        leftmost() = header;//令 header 的左子节点为自己
        rightmost() = header;//令 header 的右子节点为自己
    }
```

};

3、RB-tree 的构造方式有两种，一种以现有的 RB-tree 复制一个新的 RB-tree，另一种是产生一棵空空如也的树

4、树状结构的各种操作，最需注意的就是边界情况的发生，也就是走到根节点要有特殊的处理，为了简化处理，SGI STL 特别为根节点再设计一个父节点，名为 header

1) 左孩子指向 begin()
2) 右孩子指向 end()
3) 父节点指向 root

➢ 初始化时三个字段都指向自己

5.2.8. RB-tree 的元素操作

1、RB-tree 一开始即要求用户必须明确设定所谓的 KeyOfValue 仿函数

2、key_compare(x1,x2)

1) 当 x1<x2 返回 true
2) 当 x1>=x2 返回 false

5.2.8.1. insert_equal()

1、源码如下(已核对)(stl_tree.h)

```
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;
    link_type x = root();//从根节点开始
    while (x != 0) {//从根节点开始，往下寻找适当的插入点
        y = x;
        //遇大往左，遇小往右
        x = key_compare(KeyOfValue()(v), key(x)) ? left(x) : right(x);
    }
    return __insert(x, y, v);
    //以上，x 为新值插入点，y 为插入点之父节点，v 为新值
}
```

5.2.8.2. insert_unique()

1、源码如下(已核对)(stl_tree.h)

```
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();//从根节点开始
```

```
        bool comp = true;
        while (x != 0) {//从根节点开始，往下寻找适当的插入点
            y = x;
            comp = key_compare(KeyOfValue()(v), key(x));
            //为什么不在这里进行相等性判断???这样效率会更高的
            //遇大往左，遇小往右
            x = comp ? left(x) : right(x);
        }
        //明确 key_compare 函数当 x1>x2 时返回 true，当 x1<=x2 时返回 false
        //离开循环后，y 所指即插入点之父节点(此时它必为叶节点)
        iterator j = iterator(y);
        if (comp)//如果离开循环时，comp 为真，即表示遇大(v 严格小于父节点的关键字)，将插入于左侧
            if (j == begin())//如果插入节点之父节点为最左节点
                return pair<iterator,bool>(__insert(x, y, v), true);
            else//否则(插入节点之父节点不为最左节点)
                --j;//找到前继节点
        //这个判断处理的是
          3)  comp 返回 false，即 v>=j.key
          4)  comp 返回 true，向上找到第一个满足性质(该节点是其父节点的右孩子，因为只有向右走的过程会发生相等)的节点，而上一步的--j 操作会找到这个节点
        if (key_compare(key(j.node), KeyOfValue()(v)))
            //新键值不与既有节点之键值重复，于是执行安插操作
            return pair<iterator,bool>(__insert(x, y, v), true);
        //新值一定与树中键值重复，那么不该插入新值，并返回与新值相等值得迭代器
        return pair<iterator,bool>(j, false);
    }
```

2、这个过程其实可以在寻找叶节点的时候就比较是否重复，不用在后面大费周章

5.2.8.3. __insert()

1、源码如下(已核对)(stl_tree.h)

```
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) {
    link_type x = (link_type) x_;
    link_type y = (link_type) y_;
    link_type z;

    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()=x
            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;//设定新节点的父节点
        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);
        ++node_count;//节点数累加
        return iterator(z);//返回一个迭代器，指向新增节点
    }
```

5.2.8.4. 调整 RB-tree(旋转及改变颜色)

1、任何插入操作，与节点插入完毕后，都要做一次调整操作，将树的状态调整到符合 RB-tree 的要求

2、源码如下(已核对)(stl_tree.h)这几个函数与算法导论红黑树完全一致

```
    inline void
    __rb_tree_rebalance(__rb_tree_node_base* x, __rb_tree_node_base*& root)
    {
        x->color = __rb_tree_red;
        while (x != root && x->parent->color == __rb_tree_red) {
            if (x->parent == x->parent->parent->left) {
                __rb_tree_node_base* y = x->parent->parent->right;
                if (y && y->color == __rb_tree_red) {
                    x->parent->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->right) {
                        x = x->parent;
                        __rb_tree_rotate_left(x, root);
```

```
                }
                x->parent->color = __rb_tree_black;
                x->parent->parent->color = __rb_tree_red;
                __rb_tree_rotate_right(x->parent->parent, root);
            }
        }
        else {
            __rb_tree_node_base* y = x->parent->parent->left;
            if (y && y->color == __rb_tree_red) {
                x->parent->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) {
                    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);
            }
        }
    }
    root->color = __rb_tree_black;
}

inline void
__rb_tree_rotate_left(__rb_tree_node_base* x, __rb_tree_node_base*& root)
{
    __rb_tree_node_base* y = x->right;
    x->right = y->left;
    if (y->left !=0)
        y->left->parent = x;
    y->parent = x->parent;

    if (x == root)
        root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->left = x;
```

```
        x->parent = y;
}

inline void
__rb_tree_rotate_right(__rb_tree_node_base* x, __rb_tree_node_base*&
root)
{
        __rb_tree_node_base* y = x->left;
        x->left = y->right;
        if (y->right != 0)
                y->right->parent = x;
        y->parent = x->parent;

        if (x == root)
                root = y;
        else if (x == x->parent->right)
                x->parent->right = y;
        else
                x->parent->left = y;
        y->right = x;
        x->parent = y;
}
```

5.2.8.5. 删除元素

1、源码如下

```
template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
inline void
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::erase(iterator position) {
        link_type y = (link_type) __rb_tree_rebalance_for_erase(
                position.node,
                header->parent,
                header->left,
                header->right);
        destroy_node(y);
        --node_count;
}

inline __rb_tree_node_base*
__rb_tree_rebalance_for_erase(__rb_tree_node_base* z,
                                        __rb_tree_node_base*& root,
                                        __rb_tree_node_base*& leftmost,
                                        __rb_tree_node_base*& rightmost)
{
        __rb_tree_node_base* y = z;
```

```
__rb_tree_node_base* x = 0;
__rb_tree_node_base* x_parent = 0;
if (y->left == 0) // z has at most one non-null child. y == z.
        x = y->right; // x might be null.
else
        if (y->right == 0)// z has exactly one non-null child.    y == z.
                x = y->left; // x is not null.
        else {// z has two non-null children.    Set y to
                y = y->right; //z's successor.    x might be null.
                while (y->left != 0)
                        y = y->left;
                x = y->right;
        }
if (y != z) {// relink y in place of z.    y is z's successor
        z->left->parent = y;
        y->left = z->left;
        if (y != z->right) {
                x_parent = y->parent;
                if (x) x->parent = y->parent;
                y->parent->left = x;// y must be a left child
                y->right = z->right;
                z->right->parent = y;
        }
        else
                x_parent = y;
        if (root == z)
                root = y;
        else if (z->parent->left == z)
                z->parent->left = y;
        else
                z->parent->right = y;
        y->parent = z->parent;
        __STD::swap(y->color, z->color);
        y = z;
        // y now points to node to be actually deleted
}
else {// y == z
        x_parent = y->parent;
        if (x) x->parent = y->parent;
        if (root == z)
                root = x;
        else
                if (z->parent->left == z)
                        z->parent->left = x;
```

```
            else
                z->parent->right = x;
        if (leftmost == z)
            if (z->right == 0)// z->left must be null also
                leftmost = z->parent;
        // makes leftmost == header if z == root
            else
                leftmost = __rb_tree_node_base::minimum(x);
        if (rightmost == z)
            if (z->left == 0) // z->right must be null also
                rightmost = z->parent;
        // makes rightmost == header if z == root
            else// x == z->left
                rightmost = __rb_tree_node_base::maximum(x);
}
//这里开始维护红黑树的性质，与算法导论基本相同
if (y->color != __rb_tree_red) {
        while (x != root && (x == 0 || x->color == __rb_tree_black))
            if (x == x_parent->left) {
                __rb_tree_node_base* w = x_parent->right;
                if (w->color == __rb_tree_red) {
                    w->color = __rb_tree_black;
                    x_parent->color = __rb_tree_red;
                    __rb_tree_rotate_left(x_parent, root);
                    w = x_parent->right;
                }
                if ((w->left == 0 || w->left->color == __rb_tree_black) &&
                   (w->right == 0 || w->right->color == __rb_tree_black)) {
                    w->color = __rb_tree_red;
                    x = x_parent;
                    x_parent = x_parent->parent;
                } else {
                    if (w->right == 0 || w->right->color == __rb_tree_black)
                    {
                        if (w->left) w->left->color = __rb_tree_black;
                        w->color = __rb_tree_red;
                        __rb_tree_rotate_right(w, root);
                        w = x_parent->right;
                    }
                    w->color = x_parent->color;
                    x_parent->color = __rb_tree_black;
                    if (w->right) w->right->color = __rb_tree_black;
                    __rb_tree_rotate_left(x_parent, root);
                    break;
```

```
                }
            } else {// same as above, with right <-> left.
                __rb_tree_node_base* w = x_parent->left;
                if (w->color == __rb_tree_red) {
                    w->color = __rb_tree_black;
                    x_parent->color = __rb_tree_red;
                    __rb_tree_rotate_right(x_parent, root);
                    w = x_parent->left;
                }
                if ((w->right == 0 || w->right->color == __rb_tree_black) &&
                    (w->left == 0 || w->left->color == __rb_tree_black)) {
                    w->color = __rb_tree_red;
                    x = x_parent;
                    x_parent = x_parent->parent;
                } else {
                    if (w->left == 0 || w->left->color == __rb_tree_black) {
                        if (w->right) w->right->color = __rb_tree_black;
                        w->color = __rb_tree_red;
                        __rb_tree_rotate_left(w, root);
                        w = x_parent->left;
                    }
                    w->color = x_parent->color;
                    x_parent->color = __rb_tree_black;
                    if (w->left) w->left->color = __rb_tree_black;
                    __rb_tree_rotate_right(x_parent, root);
                    break;
                }
            }
        if (x) x->color = __rb_tree_black;
    }
    return y;
}
```

Chapter 6. 内存

6.1. 内存内容

1、在 C++中，并不会在对象所占的区域记录"内存存放内容的类型"，**内存的类型完全取决于你如何使用它**(如何看待它，即获取它的一个视图)

- 一种指针类型在编译期可以转向任意一种指针类型(好像成员函数的指针不可以)，因为这仅仅是变换了这个内存的一种视图，而在这种视图下，所执行的操作是否正确，得到运行时才能知道，编译期并不会给予保证

2、除了对象字段所占的内存空间外，会不会有额外的系统开销(overhead)或者系统开销中存放了何种数据，完全取决于编译器的实现

- 我们可以通过重写 operator new 和 operator delete 来自定义任何的负载(overhead)，这两个函数负责的是内存的创建与销毁

```
class A {
        char c[4];
        public:
        void * operator new(size_t size);
        void operator delete(void *);
};

void * A::operator new(size_t size) {
        void *raw = (void*)malloc(size+2);//多分配 2 个字节，前两个字节作为overhead
        char* pc = (char*)raw;
        pc[0] = 'a';
        pc[1] = 'b';
        return (void*)(pc+2);
}

void A::operator delete(void *p) {
        char * pc = (char*)p;
        cout << *(pc - 2) << endl;
        cout << *(pc - 1) << endl;
        free((void*)(pc - 2));
}
```

3、

6.2. 成员函数

6.2.1. 非静态成员函数

1、成员函数可以被看作是类作用域的全局函数，不在对象分配的空间里，只有虚函数才会在类对象里有一个指针，存放虚函数的地址等相关信息。

2、成员函数的地址，编译期就已确定，并静态绑定或动态的绑定在对应的对象上。在该类的第一个实例生成之前，成员函数的内存地址就已经确定了

3、对象调用成员函数时，编译器可以确定这些函数的地址，并通过传入 this 指针和其他参数，完成函数的调用，所以类中就没有必要存储成员函数的信息
4、非静态成员函数无法强制转型为(void*)类型，因为该函数与一般的函数不同，有一个接受类对象指针的入口

```
class A{
public:
    void func(int i){}
};

//声明 p 为 A 的成员函数指针
void (A::*p) (int);
p=&A::func;
```

5、有一种猥琐的方法可以将成员函数指针类型转换为 void*

```
union u{
    void * addr;
    void (A::*p) (int);
};

u _u;
_u.p=&A::func;
//然后_u.addr 就是 void*了

//如何通过 void*再转回去呢
void * temp=_u.addr;
void (A::*temp2) (int)=((u*)&temp)->p;
A a;
(a.*temp2)(1);
```

6.2.2. 静态成员函数

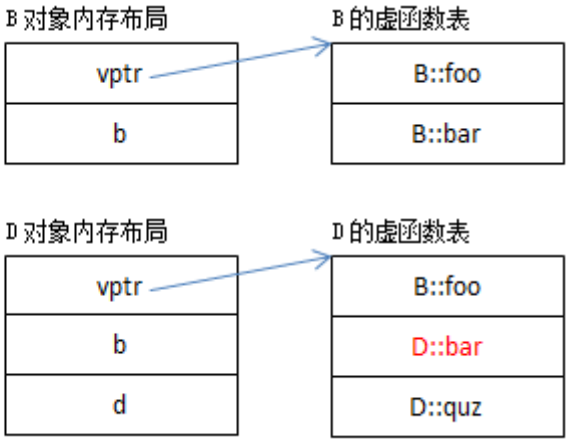
1、静态成员函数与全局函数基本相同，只是多了一个类作用域，同时可以访问该类作用域内的静态成员变量
2、静态成员函数的地址，同样在编译期就已经确定了
3、静态成员函数的地址与全局函数的地址类似，不会出现成员函数指针那样无法转换成 void*

6.3. 虚函数的底层实现机制

6.3.1. 概述

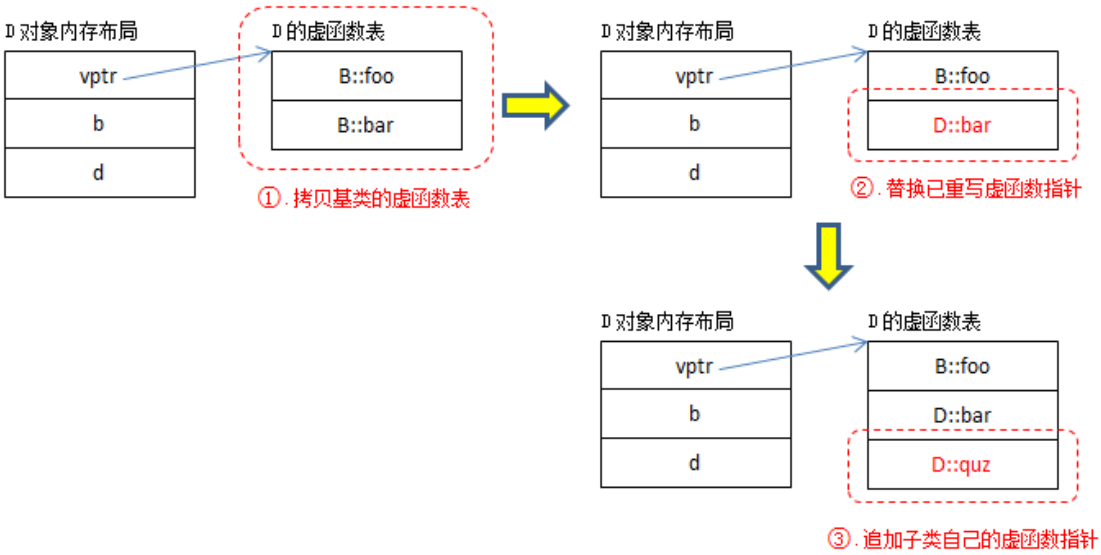
1、简单地说，每一个含有虚函数（无论是其本身的，还是继承而来的）的类都至少有一个与之对应的虚函数表，其中存放着该类所有的虚函数对应的函数指针

```
struct B {
    long b;
    virtual void foo() {}
    virtual void bar() {}
};
struct D : public B {
    long d;
    virtual void bar() {}
    virtual void quz() {}
};
```



6.3.2. 虚函数表构造过程

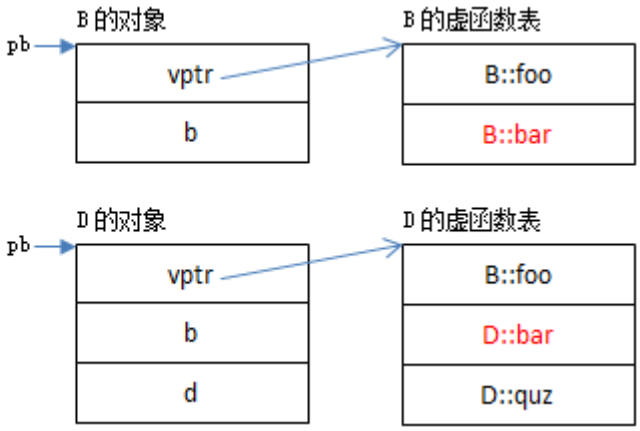
1、从编译器的角度来说，B 的虚函数表很好构造，D 的虚函数表构造过程相对复杂。下面给出了构造 D 的虚函数表的一种方式



2、过程是由编译器完成的，因此也可以说：虚函数替换过程发生在编译时

6.3.3. 虚函数调用过程

```
void test(B* pb) {
    pb->bar();
}
```



1、编译器只知道 pb 是 B*类型的指针，并不知道它指向的具体对象类型：pb 可能指向的是 B 的对象，也可能指向的是 D 的对象
2、但对于“pb->bar()”，编译时能够确定的是：此处 operator->的另一个参数是 B::bar（因为 pb 是 B*类型的，编译器认为 bar 是 B::bar），而 B::bar 和 D::bar 在各自虚函数表中的偏移位置是相等的
3、无论 pb 指向哪种类型的对象，只要能够确定被调函数在虚函数中的偏移值，待运行时，能够确定具体类型，并能找到相应 vptr 了，就能找出真正应该调用的函数

缩写

1. SGI：Silicon Graphics
2. STL：Standard Template Library

数据类型

1. size_t：size_t 类型定义在 cstddef 头文件中，该文件是 C 标准库的头文件 stddef.h 的 C++版。它是一个与机器相关的 unsigned 类型，其大小足以保证存储内存中对象的大小
2. ptrdiff_t：ptrdiff_t 是 C/C++标准库中定义的一个与机器相关的数据类型。ptrdiff_t 类型变量通常用来保存两个指针减法操作的结果