# STL概论与版本简介

## STL概论

## 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

## GUN源代码开放精神

## HP实现版本

## P.J.Plauger实现版本

## Rouge Wave实现版本

## STLport实现版本

## SGI STL实现版本

### GUN C++ headers文件分布(按字母排序)

### SGI STL文件分布与简介

### 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

# 空间配置器

## 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)

## 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

### 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

### SGI特殊的空间配置器，std::alloc

#### 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

### 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\*) {}

#### \_\_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;

};

### 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));

}

};

### 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()

### 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, };

#### 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

#### 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

### 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);

};

### 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;

}

### 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);

}

### 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));

}

}

## 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

### 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"

### 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

### 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

# 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

## 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

## 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

## 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

## 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

1. 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

1. 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

1. 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

1. 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;

}

1. 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;

### 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

### 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;

};

### 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&

### 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;

};

### 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

#### 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

#### 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

#### 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()

## 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;

};

## 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

## 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

## 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

# 序列式容器

## 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

### 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)

## vector

### 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

### 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

### 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;

}

//...

};

### 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()); }

...

};

### 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**

### 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;

}

}

}

## list

### 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

### 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;

};

### 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;

}

};

### 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]);

}

### 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()); }

...

};

### 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

};

### 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]);

}

## deque

### 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

### 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

### 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));

}

### 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;

}

### 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; }

};

### 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);

}

### 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;

}

}

## stack

### 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

### 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;

}

### 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

## queue

### 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。

### 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;

}

### 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

## heap

### 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

### heap算法

#### 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：Distance((last-1)-first)

//树根洞号：Distance(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

//以下过程类似于插入排序，找到新值的洞号

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

}

#### 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

#### 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);

}

#### 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--;

}

}

#### 总结：真的反人类

### heap没有迭代器

## priority\_queue

### 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

### 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());

}

};

### priority\_queue没有迭代器

## slist

# 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

## 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)**

### 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

### 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

### 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)

### 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



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

### 双旋转

1、在内侧插入导致不平衡的情况如下



1. Hk3+1=HD+2，即HD=Hk3-1
2. HA=Hk3-1
3. HB、HC都为Hk3-1或一个为Hk3-1另一个为Hk3-2
4. **k2在插入之前的高度为HD+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HB要么与HA相同，要么比HA少1
2. 对于k2节点，满足平衡条件，因为HC要么与HD相同，要么比HD少1
3. 对于k3节点，满足平衡条件，因为Hk1高度与Hk2相同
4. **在平衡之后k3节点的高度是HD+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**

### 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

## 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

### 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

#### 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

#### 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

#### 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

#### 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

### 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

### 节点删除

1、<未完成>：如何自上而下？

### 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;

};

### 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，而此时根节点并没有右子树，即根节点为最大节点，此时header.right指向root，所以上面的循环会导致node=header，y=header.parent=root，从而使得下面这个if不成立，返回的就是header(即end())

//当node节点是最大节点，但node不是root时，通过上面的循环，y会指向header，node会指向root，因此if条件成立，返回的就是header(即end())

}

}

//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;

}

};

### 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(KeyOfValue()(v)<key(x))

遇小往右(KeyOfValue()(v)>=key(x))

//从这里可以看出来，相同的值是往右孩子支路走的(不一定，接受两个参数的insert\_unique会将相同的值往左子树放)

x = key\_compare(KeyOfValue()(v), key(x)) ? left(x) : right(x);

}

return \_\_insert(x, y, v);

//以上，x必为0，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(KeyOfValue()(v)<key(x))

遇小往右(KeyOfValue()(v)>=key(x))

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);//x必为0

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);//x必为0

//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\_;//这个x有什么含义???

link\_type y = (link\_type) y\_;

link\_type z;

//当y为head，或x!=0或v值比key(y)小时，都将新节点插入y左边

//根据下面的两个带有迭代器的insert\_unique和insert\_equal来看，x的作用仅仅就是避免重复调用比较函数(或者说可以通过传入非0的x来指定新节点作为y节点的左孩子)，在调用\_\_insert之前可能已经进行比较运算即调用函数key\_compare，那么现在将不希望再次调用，通过这种方式来减小开销

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

}

template <class Key, class Val, class KeyOfValue, class Compare, class Alloc>

typename rb\_tree<Key, Val, KeyOfValue, Compare, Alloc>::iterator

rb\_tree<Key, Val, KeyOfValue, Compare, Alloc>::insert\_unique(iterator position,

const Val& v) {

if (position.node == header->left) // begin()

if (size() > 0 && key\_compare(KeyOfValue()(v), key(position.node)))

//若v严格小于最小值，那么将新节点作为最小节点的左孩子

return \_\_insert(position.node, position.node, v);

// first argument just needs to be non-null 这个注释证实了猜测

else

//如果不是严格小于，那么将重复判断的工作转交给接受一个参数的insert\_unique函数来执行

return insert\_unique(v).first;

else if (position.node == header) // end()

if (key\_compare(key(rightmost()), KeyOfValue()(v)))

//v严格大于最大值，因此要插入到rightmost()的右孩子，此时第一个参数x必须传入0，否则会插入到左边

return \_\_insert(0, rightmost(), v);

else

//如果不是严格大于，那么将重复判断的工作转交给接受一个参数的insert\_unique函数来执行

return insert\_unique(v).first;

else {

iterator before = position;

--before;//找到前继

if (key\_compare(key(before.node), KeyOfValue()(v))

&& key\_compare(KeyOfValue()(v), key(position.node)))

//v严格小于node的实值，且严格大于前继before的实值

if (right(before.node) == 0)

//before没有右孩子，此时before一定是node左子树中的最大值，那么将新节点作为before的右孩子即可

return \_\_insert(0, before.node, v);

else

//由于before有右孩子，那么before一定是node的父辈，说明node没有左子树，因此将新节点作为node的左孩子即可

return \_\_insert(position.node, position.node, v);

// first argument just needs to be non-null 这个注释证实了猜测

else

return insert\_unique(v).first;

}

}

template <class Key, class Val, class KeyOfValue, class Compare, class Alloc>

typename rb\_tree<Key, Val, KeyOfValue, Compare, Alloc>::iterator

rb\_tree<Key, Val, KeyOfValue, Compare, Alloc>::insert\_equal(iterator position,

const Val& v) {

if (position.node == header->left) // begin()

if (size() > 0 && key\_compare(KeyOfValue()(v), key(position.node)))

//如果v严格小于最小值，那么将新节点最为最小节点左孩子

return \_\_insert(position.node, position.node, v);

// first argument just needs to be non-null 这个注释证实了猜测

else

//否则，将工作转交给接受一个参数的insert\_equal函数

return insert\_equal(v);

else if (position.node == header) // end()

if (!key\_compare(KeyOfValue()(v), key(rightmost())))

//如果v不小于最大值，那么将新节点作为最大节点右孩子，右孩子是允许于父节点值相同的

return \_\_insert(0, rightmost(), v);

else

//否则，将工作转交给接受一个参数的insert\_equal函数return insert\_equal(v);

else {

iterator before = position;

--before;//找到前继

if (!key\_compare(KeyOfValue()(v), key(before.node))

&& !key\_compare(key(position.node), KeyOfValue()(v)))

//v>=key(before.node)并且key(position.node)>=v

if (right(before.node) == 0)

//before必定是position的左子树，将新值作为before的右孩子即可，此时v可能与key(position.node)相等，而将v放到position的左子树中，因此相等的值可以任意放在左右子树中?

return \_\_insert(0, before.node, v);

else

//before为position的父辈

return \_\_insert(position.node, position.node, v);

// first argument just needs to be non-null 这个注释证实了猜测

else

return insert\_equal(v);

}

}

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;

}

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) //情况A：z最多只有一个孩子

x = y->right; //x可能是null

else

if (y->right == 0)//情况B：z没有任何孩子

x = y->left; // x is not null.

else {//情况C：z有两个孩子

//以下找到z的后继y，即右子树的最小节点

//x为y的右子节点，x可能为null

y = y->right;

while (y->left != 0)

y = y->left;

x = y->right;

}

if (y != z) {//只有情况C会跑到这里，将y节点抽出，并代替z节点，那么相当于z节点并没有被删除，被删除的节点转换成了y节点

z->left->parent = y;

y->left = z->left;

if (y != z->right) {

//z与y之间有其他节点，抽出y节点需要将x子树重新接回y的父节点

x\_parent = y->parent;

if (x) x->parent = y->parent;

y->parent->left = x;//由于y是右子树的最小节点，因此y必定是其父节点的左孩子

y->right = z->right;

z->right->parent = y;

}

else

//z与y之间没有其他节点，抽出y节点，然后将x接回z，然后再将y接替z节点，此时x又成为了y的右子节点，于是一步到位，x不需要更改

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此时指向真正被删除的节点，但是其颜色确是这两句执行以前y的颜色，因此，其实不要以上两句也没问题，只需要将y的颜色替换成z的颜色，并保留y之前的颜色即可

}

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);

}

//这里始终保持y持有的颜色是被"删除(这里的删除，不考虑节点的地址，节点的颜色以及节点的值，而是指节点在树中的位置)"节点的颜色

1. 若z最多只有一个孩子，那就是z的颜色
2. 若z有两个孩子，那么就是z右子树最小节点的颜色

//这里开始维护红黑树的性质，与算法导论基本相同，有一个小区别，STL红黑树没有哨兵节点，因此某些细节有些不一致

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;

}

### 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

### 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

#### 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

}

#### 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

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);

}

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

#### \_\_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

}

#### \_\_rb\_tree\_rebalance

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;

}

#### erase

1、源码如下(已核对)(stl\_tree.h)

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) //情况A：z最多只有一个孩子

x = y->right; //x可能是null

else

if (y->right == 0)//情况B：z没有任何孩子

x = y->left; // x is not null.

else {//情况C：z有两个孩子

//以下找到z的后继y，即右子树的最小节点

//x为y的右子节点，x可能为null

y = y->right;

while (y->left != 0)

y = y->left;

x = y->right;

}

if (y != z) {//只有情况C会跑到这里，将y节点抽出，并代替z节点，那么相当于z节点并没有被删除，被删除的节点转换成了y节点

z->left->parent = y;

y->left = z->left;

if (y != z->right) {

//z与y之间有其他节点，抽出y节点需要将x子树重新接回y的父节点

x\_parent = y->parent;

if (x) x->parent = y->parent;

y->parent->left = x;//由于y是右子树的最小节点，因此y必定是其父节点的左孩子

y->right = z->right;

z->right->parent = y;

}

else

//z与y之间没有其他节点，抽出y节点，然后将x接回z，然后再将y接替z节点，此时x又成为了y的右子节点，于是一步到位，x不需要更改

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此时指向真正被删除的节点，但是其颜色确是这两句执行以前y的颜色，因此，其实不要以上两句也没问题，只需要将y的颜色替换成z的颜色，并保留y之前的颜色即可

}

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);

}

//这里始终保持y持有的颜色是被"删除(这里的删除，不考虑节点的地址，节点的颜色以及节点的值，而是指节点在树中的位置)"节点的颜色

1. 若z最多只有一个孩子，那就是z的颜色
2. 若z有两个孩子，那么就是z右子树最小节点的颜色

//这里开始维护红黑树的性质，与算法导论基本相同，有一个小区别，STL红黑树没有哨兵节点，因此某些细节有些不一致

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;

}

## set

1、set的特性是，所有元素都会根据元素的键值自动被排序，set的元素不像map那样可以拥有实值(value)和键值(key)。set不允许两个元素有相同的键值

2、**set<T>::iterator被定义为底层RB-tree的const\_iterator，杜绝写入操作**

3、set拥有list的某些性质：当客户端对它进行元素新增操作insert和erase时，操作之前的所有迭代器，在操作完成之后都依然有效

4、STL特别提供了一组set/multiset相关算法，包括交集set\_intersection、联集set\_union、差集set\_difference、对称差集set\_symmetric\_difference

5、由于RB-tree是一种平衡二叉搜索树，自动排序效果不错，所以标准的STL set即以RB-tree为底层机制。几乎所有的set操作行为，都是转调用RB-tree的操作行为而已

5、源码如下(已核对)(stl\_set.h)

template <class Key, class Compare = less<Key>, class Alloc = alloc>

class set {

public:

// typedefs:

typedef Key key\_type;

typedef Key value\_type;

typedef Compare key\_compare;

typedef Compare value\_compare;

private:

typedef rb\_tree<key\_type, value\_type,

identity<value\_type>, key\_compare, Alloc> rep\_type;

rep\_type t; // red-black tree representing set

public:

typedef typename rep\_type::const\_pointer pointer;

typedef typename rep\_type::const\_pointer const\_pointer;

typedef typename rep\_type::const\_reference reference;

typedef typename rep\_type::const\_reference const\_reference;

typedef typename rep\_type::const\_iterator iterator;

typedef typename rep\_type::const\_iterator const\_iterator;

typedef typename rep\_type::const\_reverse\_iterator reverse\_iterator;

typedef typename rep\_type::const\_reverse\_iterator

const\_reverse\_iterator;

typedef typename rep\_type::size\_type size\_type;

typedef typename rep\_type::difference\_type difference\_type;

// allocation/deallocation

set() : t(Compare()) {}

explicit set(const Compare& comp) : t(comp) {}

template <class InputIterator>

set(InputIterator first, InputIterator last)

: t(Compare()) { t.insert\_unique(first, last); }

template <class InputIterator>

set(InputIterator first, InputIterator last, const Compare& comp)

: t(comp) { t.insert\_unique(first, last); }

set(const set<Key, Compare, Alloc>& x) : t(x.t) {}

set<Key, Compare, Alloc>& operator=(const set<Key, Compare, Alloc>& x) {

t = x.t;

return \*this;

}

// accessors:

key\_compare key\_comp() const { return t.key\_comp(); }

value\_compare value\_comp() const { return t.key\_comp(); }

iterator begin() const { return t.begin(); }

iterator end() const { return t.end(); }

reverse\_iterator rbegin() const { return t.rbegin(); }

reverse\_iterator rend() const { return t.rend(); }

bool empty() const { return t.empty(); }

size\_type size() const { return t.size(); }

size\_type max\_size() const { return t.max\_size(); }

void swap(set<Key, Compare, Alloc>& x) { t.swap(x.t); }

// insert/erase

typedef pair<iterator, bool> pair\_iterator\_bool;

pair<iterator,bool> insert(const value\_type& x) {

pair<typename rep\_type::iterator, bool> p = t.insert\_unique(x);

return pair<iterator, bool>(p.first, p.second);

}

iterator insert(iterator position, const value\_type& x) {

typedef typename rep\_type::iterator rep\_iterator;

return t.insert\_unique((rep\_iterator&)position, x);

}

template <class InputIterator>

void insert(InputIterator first, InputIterator last) {

t.insert\_unique(first, last);

}

void erase(iterator position) {

typedef typename rep\_type::iterator rep\_iterator;

t.erase((rep\_iterator&)position);

}

size\_type erase(const key\_type& x) {

return t.erase(x);

}

void erase(iterator first, iterator last) {

typedef typename rep\_type::iterator rep\_iterator;

t.erase((rep\_iterator&)first, (rep\_iterator&)last);

}

void clear() { t.clear(); }

// set operations:

iterator find(const key\_type& x) const { return t.find(x); }

size\_type count(const key\_type& x) const { return t.count(x); }

iterator lower\_bound(const key\_type& x) const {

return t.lower\_bound(x);

}

iterator upper\_bound(const key\_type& x) const {

return t.upper\_bound(x);

}

pair<iterator,iterator> equal\_range(const key\_type& x) const {

return t.equal\_range(x);

}

friend bool operator== \_\_STL\_NULL\_TMPL\_ARGS (const set&, const set&);

friend bool operator< \_\_STL\_NULL\_TMPL\_ARGS (const set&, const set&);

};

## map

1、map的特性是，所有元素都会根据元素的键值自动被排序，map的所有元素都是pair，同时拥有实值(value)和键值(key)，pair的第一个元素被视为键值，第二元素被视为实值

2、map不允许两个元素拥有相同的键值

3、pair源码如下(已核对)(stl\_pair.h)

template <class T1, class T2>

struct pair {

typedef T1 first\_type;

typedef T2 second\_type;

T1 first;

T2 second;

pair() : first(T1()), second(T2()) {}

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

template <class U1, class U2>

pair(const pair<U1, U2>& p) : first(p.first), second(p.second) {}

};

template <class T1, class T2>

inline bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y) {

return x.first == y.first && x.second == y.second;

}

template <class T1, class T2>

inline bool operator<(const pair<T1, T2>& x, const pair<T1, T2>& y) {

return x.first < y.first || (!(y.first < x.first) && x.second < y.second);

}

template <class T1, class T2>

inline pair<T1, T2> make\_pair(const T1& x, const T2& y) {

return pair<T1, T2>(x, y);

}

4、我们可以通过map的迭代器改变map元素内容么?

1. 不能修改键值，因为map元素的键值关系到map元素的排列规则，任意改变map元素键值将会严重破坏map组织
2. 可以修改实值

* 因此，map iterators既不是一种constant iterators，也不是一种mutable iterators

5、map拥有和list相同的某些性质：当客户端对它进行元素新增操作或删除操作时，操作之前的所有迭代器，在操作完成之后都依然有效(显然是不包括被删除元素的迭代器)

6、标准的STLmap即以RB-tree为底层机制，几乎所有map操作行为，都只是转调用RB-tree的操作行为而已

7、源码如下(已核对)(stl\_map.h)

template <class Key, class T, class Compare = less<Key>, class Alloc = alloc>

class map {

public:

// typedefs:

typedef Key key\_type;//键值型别

typedef T data\_type;//数据(实值)型别

typedef T mapped\_type;

typedef pair<const Key, T> value\_type;//元素型别(键值/实值)

typedef Compare key\_compare;//键值比较函数

class value\_compare

: public binary\_function<value\_type, value\_type, bool> {

friend class map<Key, T, Compare, Alloc>;

protected :

Compare comp;

value\_compare(Compare c) : comp(c) {}

public:

bool operator()(const value\_type& x, const value\_type& y) const {

return comp(x.first, y.first);

}

};

private:

typedef rb\_tree<key\_type, value\_type,

select1st<value\_type>, key\_compare, Alloc> rep\_type;

rep\_type t; // red-black tree representing map

public:

typedef typename rep\_type::pointer pointer;

typedef typename rep\_type::const\_pointer const\_pointer;

typedef typename rep\_type::reference reference;

typedef typename rep\_type::const\_reference const\_reference;

typedef typename rep\_type::iterator iterator;//由于允许修改实值，因此含有这个版本的迭代器

typedef typename rep\_type::const\_iterator const\_iterator;

typedef typename rep\_type::reverse\_iterator reverse\_iterator;

typedef typename rep\_type::const\_reverse\_iterator const\_reverse\_iterator;

typedef typename rep\_type::size\_type size\_type;

typedef typename rep\_type::difference\_type difference\_type;

// allocation/deallocation

map() : t(Compare()) {}

explicit map(const Compare& comp) : t(comp) {}

template <class InputIterator>

map(InputIterator first, InputIterator last)

: t(Compare()) { t.insert\_unique(first, last); }

template <class InputIterator>

map(InputIterator first, InputIterator last, const Compare& comp)

: t(comp) { t.insert\_unique(first, last); }

map(const map<Key, T, Compare, Alloc>& x) : t(x.t) {}

map<Key, T, Compare, Alloc>& operator=(

const map<Key, T, Compare, Alloc>& x)

{

t = x.t;

return \*this;

}

// accessors:

key\_compare key\_comp() const { return t.key\_comp(); }

value\_compare value\_comp() const { return value\_compare(t.key\_comp()); }

iterator begin() { return t.begin(); }

const\_iterator begin() const { return t.begin(); }

iterator end() { return t.end(); }

const\_iterator end() const { return t.end(); }

reverse\_iterator rbegin() { return t.rbegin(); }

const\_reverse\_iterator rbegin() const { return t.rbegin(); }

reverse\_iterator rend() { return t.rend(); }

const\_reverse\_iterator rend() const { return t.rend(); }

bool empty() const { return t.empty(); }

size\_type size() const { return t.size(); }

size\_type max\_size() const { return t.max\_size(); }

T& operator[](const key\_type& k) {

return (\*((insert(value\_type(k, T()))).first)).second;

}

void swap(map<Key, T, Compare, Alloc>& x) { t.swap(x.t); }

// insert/erase

pair<iterator,bool> insert(const value\_type& x) { return t.insert\_unique(x); }

iterator insert(iterator position, const value\_type& x) {

return t.insert\_unique(position, x);

}

template <class InputIterator>

void insert(InputIterator first, InputIterator last) {

t.insert\_unique(first, last);

}

void erase(iterator position) { t.erase(position); }

size\_type erase(const key\_type& x) { return t.erase(x); }

void erase(iterator first, iterator last) { t.erase(first, last); }

void clear() { t.clear(); }

// map operations:

iterator find(const key\_type& x) { return t.find(x); }

const\_iterator find(const key\_type& x) const { return t.find(x); }

size\_type count(const key\_type& x) const { return t.count(x); }

iterator lower\_bound(const key\_type& x) {return t.lower\_bound(x); }

const\_iterator lower\_bound(const key\_type& x) const {

return t.lower\_bound(x);

}

iterator upper\_bound(const key\_type& x) {return t.upper\_bound(x); }

const\_iterator upper\_bound(const key\_type& x) const {

return t.upper\_bound(x);

}

pair<iterator,iterator> equal\_range(const key\_type& x) {

return t.equal\_range(x);

}

pair<const\_iterator,const\_iterator> equal\_range(const key\_type& x) const {

return t.equal\_range(x);

}

friend bool operator== \_\_STL\_NULL\_TMPL\_ARGS (const map&,

const map&);

friend bool operator< \_\_STL\_NULL\_TMPL\_ARGS (const map&, const map&);

};

template <class Key, class T, class Compare, class Alloc>

inline bool operator==(const map<Key, T, Compare, Alloc>& x,

const map<Key, T, Compare, Alloc>& y) {

return x.t == y.t;

}

template <class Key, class T, class Compare, class Alloc>

inline bool operator<(const map<Key, T, Compare, Alloc>& x,

const map<Key, T, Compare, Alloc>& y) {

return x.t < y.t;

}

template <class Key, class T, class Compare, class Alloc>

inline void swap(map<Key, T, Compare, Alloc>& x,

map<Key, T, Compare, Alloc>& y) {

x.swap(y);

}

## multiset

1、multiset的特性以及用法和set完全相同，唯一的差别在于它允许键值重复，因为它的插入操作采用的是底层机制RB-tree的insert\_equal()而非insert\_unique()

2、下面源码仅列出与set不同的部分(已核对)(stl\_multiset.h)

template <class Key, class Compare = less<Key>, class Alloc = alloc>

class multiset {

...

template <class InputIterator>

multiset(InputIterator first, InputIterator last)

: t(Compare()) { t.insert\_equal(first, last); }

template <class InputIterator>

multiset(InputIterator first, InputIterator last, const Compare& comp)

: t(comp) { t.insert\_equal(first, last); }

...

iterator insert(const value\_type& x) {

return t.insert\_equal(x);

}

iterator insert(iterator position, const value\_type& x) {

typedef typename rep\_type::iterator rep\_iterator;

return t.insert\_equal((rep\_iterator&)position, x);

}

template <class InputIterator>

void insert(InputIterator first, InputIterator last) {

t.insert\_equal(first, last);

}

...

};

## multimap

1、multimap的特性以及用法与map完全相同，唯一的差别在于它允许键值重复，因为它的插入操作采用的是底层机制RB-tree的insert\_equal()而非insert\_unique()

2、下面源码仅列出与map不同的部分(已核对)(stl\_multimap.h)

template <class Key, class T, class Compare = less<Key>, class Alloc = alloc>

#else

template <class Key, class T, class Compare, class Alloc = alloc>

#endif

class multimap {

...

template <class InputIterator>

multimap(InputIterator first, InputIterator last)

: t(Compare()) { t.insert\_equal(first, last); }

template <class InputIterator>

multimap(InputIterator first, InputIterator last, const Compare& comp)

: t(comp) { t.insert\_equal(first, last); }

...

iterator insert(const value\_type& x) { return t.insert\_equal(x); }

iterator insert(iterator position, const value\_type& x) {

return t.insert\_equal(position, x);

}

template <class InputIterator>

void insert(InputIterator first, InputIterator last) {

t.insert\_equal(first, last);

}

...

};

## hashtable

1、二叉搜索树具有对数平均性，但这样的表现构造在一个假设上，输入数据具有足够的随机性???(不需要吧，红黑树不是可以自动调整么)

2、hash table(散列表)，这种结构在插入、删除、搜寻等操作上页具有"常数平均时间"的表现，这种表现是以统计为基础，不需仰赖元素的随机性

### hashtable概述

1、hash table可提供对任何有名项(named item)的存取操作和删除操作。由于操作对象是有名项，所以hashtable也被称为一种字典结构(dictionary)。这种结构的用意在于提供常数时间之基本操作，就像stack或queue那样

2、假设所有元素都是16-bits且不带正负号，范围0-65535。那么简单的运用一个array就可以满足要求

* 首先配置一个array A，拥有65535个元素，索引号码0-65535，初值全部为0
* 每个元素代表相应元素的出现次数
* 如果插入元素i，就执行A[i]++
* 如果删除元素i，就执行A[i]--
* 如果搜寻元素i，就检查A[i]是否为0
* 以上每一个操作都是常数时间，但是额外负担是array的空间和初始化时间

3、以上解法存在两个现实问题

1. 如果元素是32bit而不是16bit，那么准备的array就必须是232=4GB，这就大得不切实际了
2. 如果元素形态是字符串(或其他)而非整数(非负整数有一个天然优势，可以直接关联到数组索引，这本身就是一种映射)，将无法被拿来作为array的索引

4、如何避免一个大的荒谬的array呢，办法之一就是使用某种映射函数，将大数映射为小数

* 负责将某一个元素映射为一个"大小可接受的索引"，这样的函数称为hash function(散列函数)
* 例如最简单的可以是func(X)=X%TableSize

5、使用hash function会带来一个问题：可能有不同的元素被映射到相同的位置(亦即有相同的索引)。这样无法避免，因为元素个数大于array容量，这便是所谓的"碰撞"问题，解决碰撞的方法有以下几种：

1. 线性探测(linear probing)
2. 二次探测(quadratic probing)
3. 开链(separate chaining)

#### 线性探测

1、首先介绍一个名词：负载系数(loading factor)，意指元素个数除以整表格大小，负载系数永远在0~1之间---除非采用开链策略

2、当hash function计算出某个元素的插入位置，而该位置上的空间已不再可用时，最简单的办法就是循序往下一一寻找(如果到达尾端，就绕到头部继续寻找)，直到找到一个可用空间为止。只要表格足够大，总是能够找到一个安身立命的空间

3、进行元素搜索操作时，道理也想通，如果hash function计算出来的位置上的元素值与我们的搜寻目标不符，就循序往下一一寻找，直到找吻合者，或直到遇上空格元素

4、元素的删除必须采用惰性删除(lazy deletion)，也就是只标记删除记号，实际删除操作则待表格重新整理(rehashing)时再进行---因为hash table中的每一个元素不仅表述它自己，也关系到其他元素的排列

5、欲分析线性探测的表现，需要两个假设

1. 表格足够大
2. 每个元素都独立

* 在此假设下，最坏的情况是线性巡访整个表格，平均情况则是巡访一半表格，这已经和我们所期望的常数时间天差地远了，而实际情况更糟，因为第二个假设太天真

#### 二次探测

1、二次探测主要用来解决主集团(primary clustering)的问题，其命名由来是因为解决碰撞问题的方程式F(i)=i2是个二次方程式。更明确地说，如果hash function计算出新元素的位置为H，而该位置实际上已被使用，那么我们就依序尝试H+12,H+22,...,H+i2，而不是像线性探测那样依序尝试H+1,H+2,...,H+i

2、二次探测带来的问题

1. 线性探测每次探测的都必然是一个不同的位置，二次探测法能否保证如此?，二次探测法是否能够保证表格之中如果没有X，那么我们插入X一定能够成功
2. 线性探测法的运算过程极其简单，二次探测法显然复杂得多，这是否会在效率上带来太多负面影响
3. 不论线性探测或者二次探测，当负载系数过高时，表格能否动态成长

3、假设表格大小为质数(prime)，而且永远保持负载系数在0.5以下(也就是超过0.5就重新配置并重新整理表格，那么就可以确定每插入一个新元素所需要的探测次数不多于2

4、实现复杂度的问题，一般总是这样考虑：赚的比花的多，才值得去做，我们增加了探测次数，所获得的利益好歹总得多过二次函数计算所多花的时间

Hi=H0+i2(mod M)

Hi-1=H0+(i-1)2(mod M)

Hi-Hi-1=i2-(i-1)2(mod M)

Hi-Hi-1+2i-1(mod M)

* 于是，我们就能通过前一个H来计算下一个H，就不需要执行二次方所需要的乘法了，虽然还需要一个乘法，但是可以位移来快速完成，mod运算也并非真正需要

5、array的成长，首先必须找出下一个新的而且够大的(大约两倍)质数，然后必须重考虑表格重建(rehashing)的成本，因为我们不能原封不动地拷贝，而是要将所有元素重新插入新表格中

6、二次探测可以消除主集团(primary clustering)，却可能造成次集团(secondary clustering)：两个元素经hash function计算出来的位置若相同，则插入时所探测的位置也相同，形成某种浪费，消除次集团的办法也有，例如复试散列

#### 开链

1、另一种与二次探测法分庭抗礼，所谓的开链(seqarate chaining)法，这种做法是在每个表格中维护一个list，hash function为我们分配某一个list，然后我们在那个list上执行元素插入、搜寻、删除等操作，虽然针对list而进行的搜寻只能是一种线性操作，但如果list足够短，速度还是够快的

## hashtable的桶子(buckets)与节点(nodes)

1、SGI采用的就是开链法

# 内存

## 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、

## 成员函数

### 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);

### 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

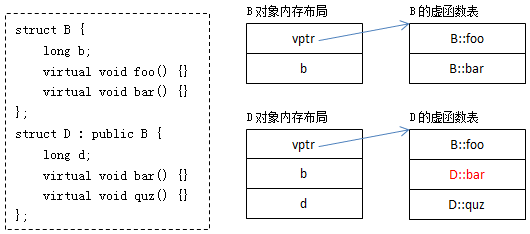
2、静态成员函数的地址，同样在编译期就已经确定了

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\*

## 虚函数的底层实现机制

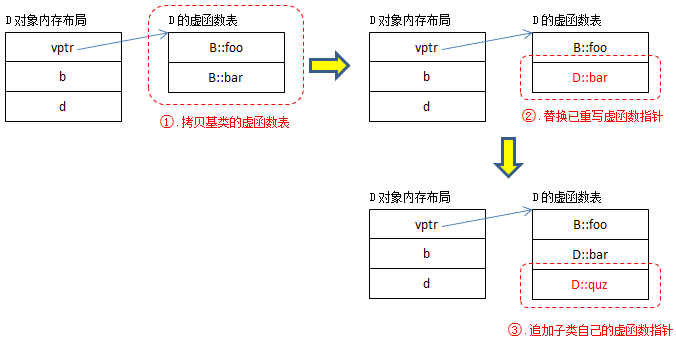
### 概述

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



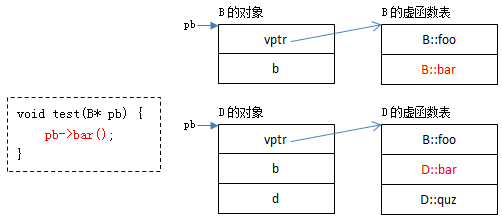
### 虚函数表构造过程

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

### 虚函数调用过程



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