### 记录时间2019.12.3-2019.12.8

**开发**： **BUG/问题**： **分析/解决**： **辨析研究**： **重点标记**： **阅读书籍**：

2019.12.3

**阅读书籍**：C++设计新思维

policy based design

2019.12.4

**开发1**：完成list容器，进行一次code review

1. 规范了git提交信息格式：

<类型>(影响模块) 具体描述信息 (#issue？)

**类型**： feature、style、fix、refactor、test、docs、accomplish、update

**影响模块**：例如list.h这样写

**具体描述信息**：这次提交做了什么事情

**#issue？**：对应具体的issue号

具体例子如

<feature>(deque.h)创建deque容器

1. 规范一套自己的 gitflow

新开一个功能模块，就开一条分支，当完成一定量的设计时，再开一条test分支，用来检查之前的工作，大概没问题以后进行合并。

master

↘deque-------------- ---------

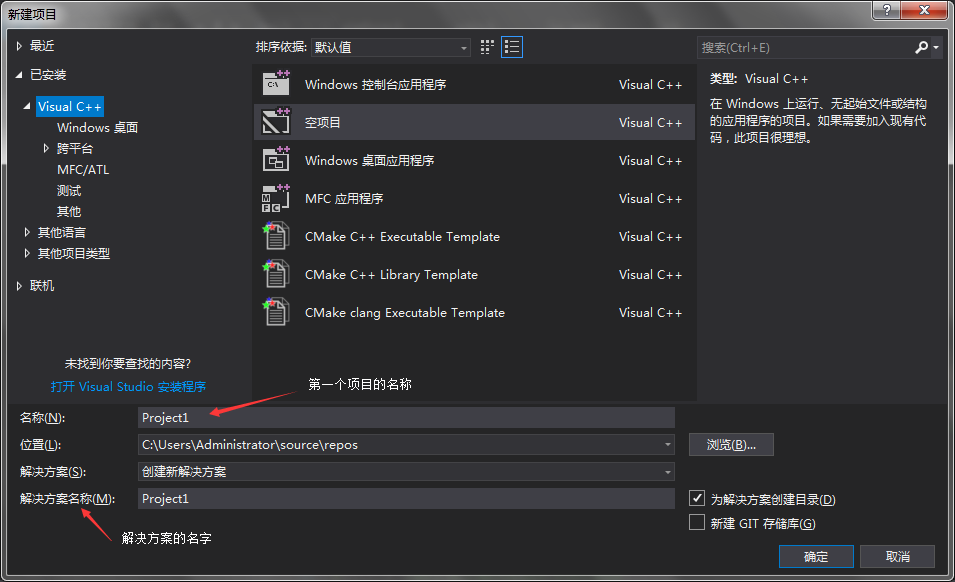
↘test--------↗

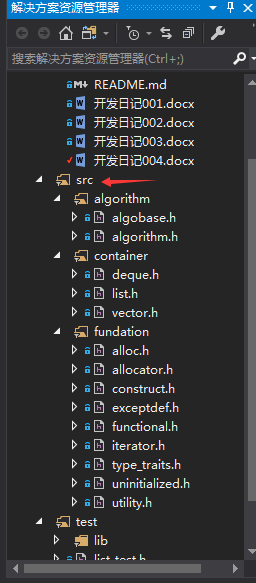
2019.12.5

**辨析研究1**：vs2017文件组织层次

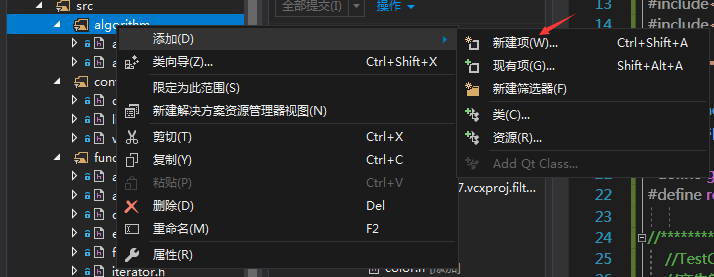
博客讲解：<https://blog.csdn.net/luoyu510183/article/details/83692426>

这个博主的一系列讲解我个人觉得很实用也很仔细，我整个文件组织形式就是参照了他说的格式。

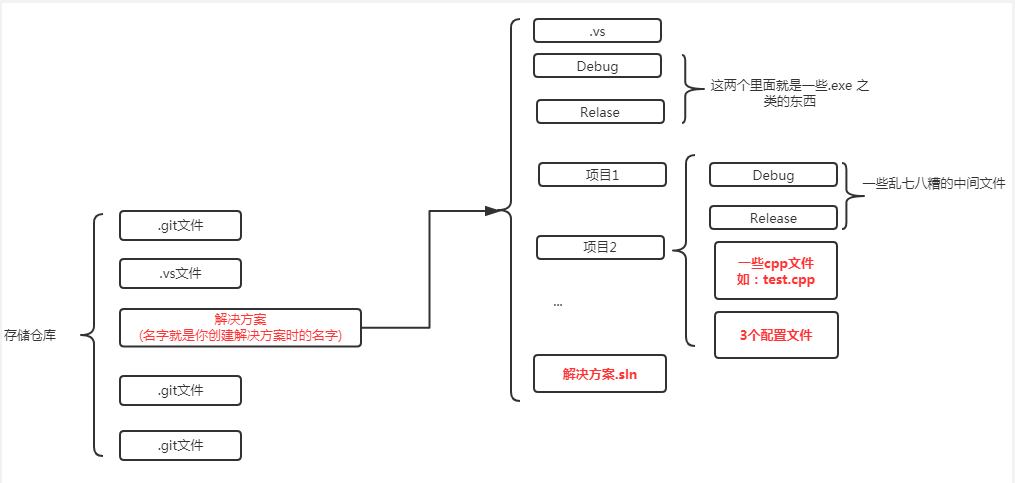




这里资源管理器中的文件夹其实并不是真实存在的，只是一个筛选器的作用，帮助我们组织视图而已。



当我们在这些“文件夹”下新建文件时，所有的文档其实都在一个目录下面，如下图所示的红色字体：一些cpp文件 如：test.cpp

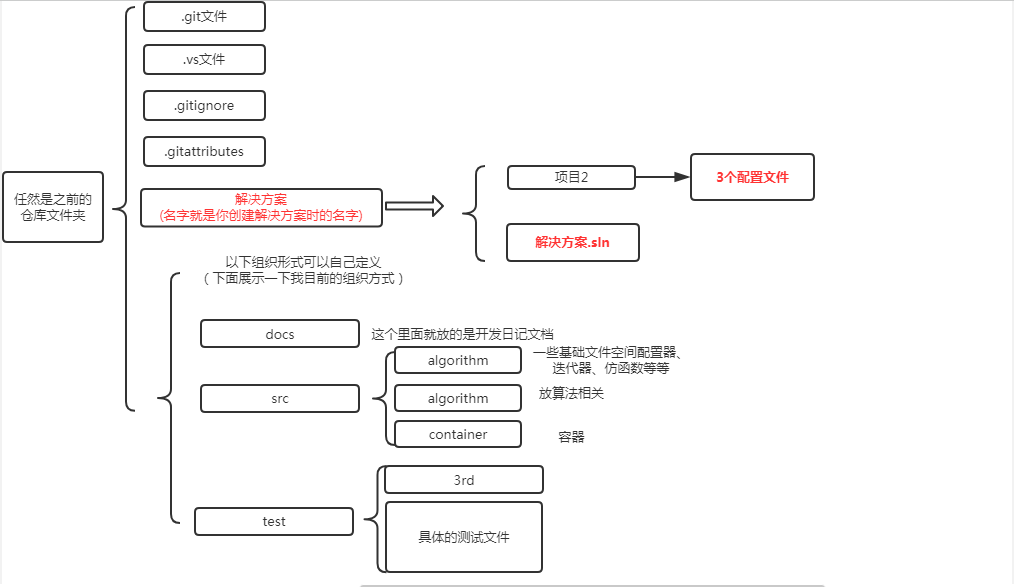


通常情况下vs下的文件组织形式是这样的，当我们要把一个项目上传时，里面有很多东西是需要的。下面展示一个重新组织的结构。

这里有一点需要说明的就是，当一些文件移动以后，在vs2017中解决方案资源管理器中的文件就失效了，需要删除，然后点击添加现有文件重新导入。

资源管理器中的视图按照我们真实的文件组织方式来构建筛选器，这样统一便于管理。

新建一个文档时，不要在“解决方案资源管理器”中新建，不然默认会创建在下图那个项目2里面去.. 应该找到你想放的文件夹，在里面新建，然后再在vs中添加



这样我们上传项目时，就可以大大减小空间了

**辨析研究2**：Google test测试框架

宏定义中一些符号的含义：

#define Fun(arg) my##arg

##是连接的意思，Fun(abc) 展开是 myabc

#define strcpy\_(dst,src) strcpy(dst, #src)

#是把后面的参数转换成一个字符串，strcpy\_(buff,abc) 相当于strcpy(buff,”abc”)

#define TESTCASE\_NAME(testcase\_name) \

testcase\_name##\_TEST

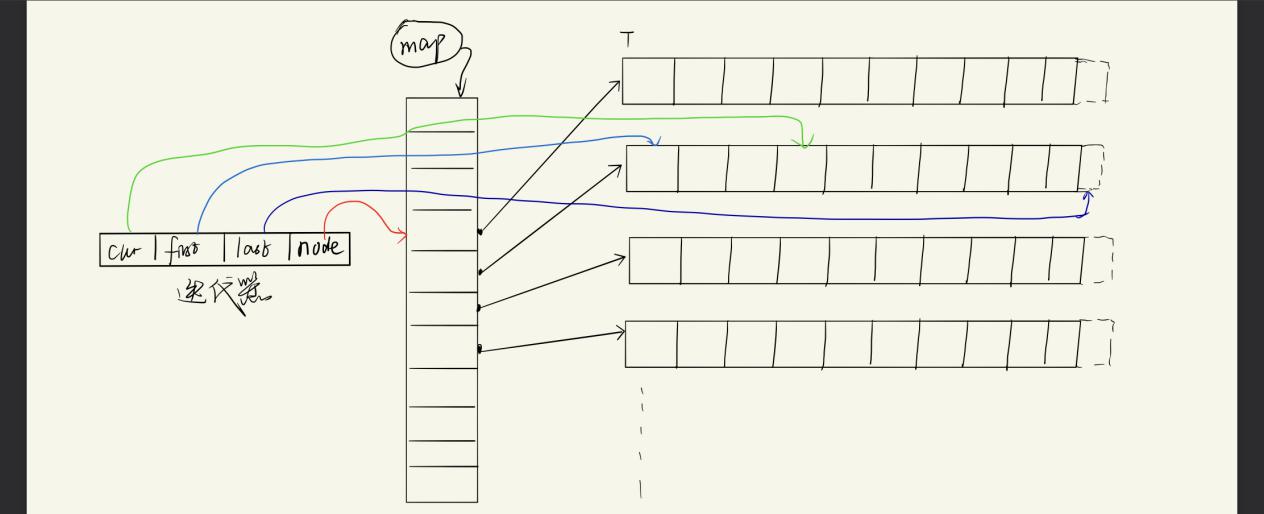
这里\表示下一行和上一行是属于一行的

参考博客讲解：<https://www.cnblogs.com/coderzh/archive/2009/04/12/1434155.html>

先使用起来，后续熟悉了这个框架再添加额外的功能

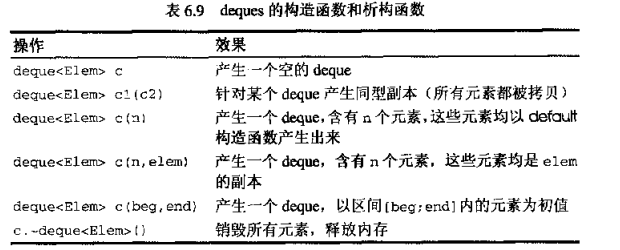
**开发1**：熟悉deque内部数据结构，每个成员函数的作用

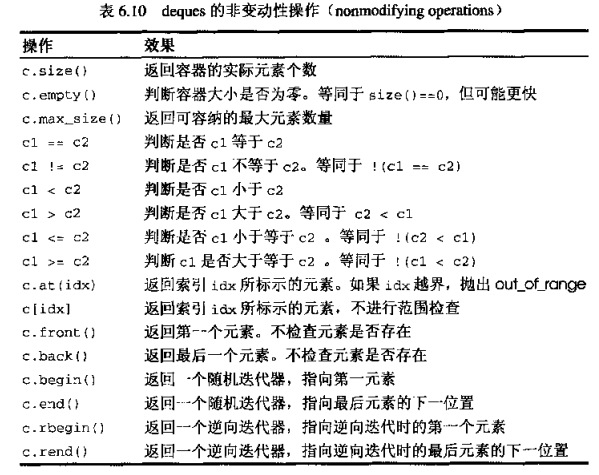
参考STL源码剖析，钻研一下内部构造

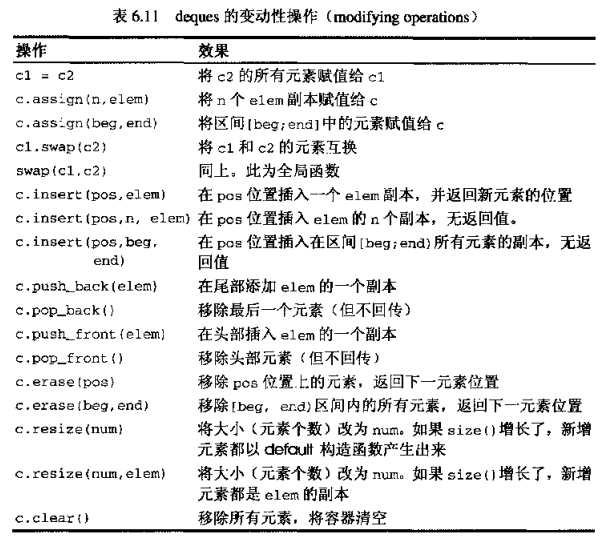


其实deque和vector可以比作是一维数组和二维数组。map是一个指针，指向一块连续的内存区，这个内存区存的是指向真正的存储单元的指针。

成员函数参考标准库这本书，大体上和vector差不多







**开发2**：迭代器的设计

根据deque容器的特点，迭代器是个random\_access\_iterator。迭代器的基本操作：

位移操作符：++、--、（根据需求可能会有+、+=、-、-=）

取值操作符：\*、->

比较操作符：==、!=、（看实际需求）

尝试转换一下开发的模式，先试着分析每个成员函数的内部实现的大概细节，然后试着归纳应该提供一些怎样的辅助函数。

2019.12.6

**开发1**：运用软件工程的知识来开发容器！

之前的vector、list设计的时候都显得好混乱不堪，居然把软件工程的相关方法给忘了！不应该、不应该，deque试着运用软件工程学相关的知识点来开发。

一下午的实验，一天啥也没干。

2019.12.7

**开发1**：需求分析

**①相对于vector，deque可以在首尾进行元素增删**

②容器内部元素可以是任意类型的(所有元素统一)

③用户不需要知道容器内部的数据结构，只需要操作统一的一种抽象，迭代器。由迭代器来真正操作容器内部的元素

**开发2**：总体设计

->**分析1**：基于①的需求，尝试分析deque的数据结构：

我们都知道数据结构双端队列是可以进行首尾增删的，而deque是基于这样一种首尾输入输出不受限制的数据结构，deque类封装了一些方法，在底层数据结构的基础上提供了一系列的功能。

假设deque像vector一样设计

留给首部增加元素 < 真正存储元素的内存 > 留个尾部增加元素

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

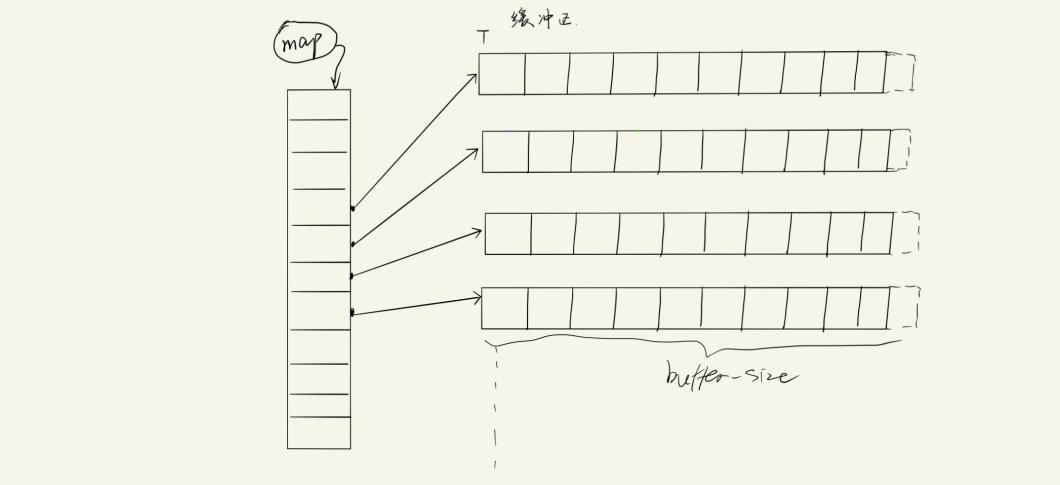
用四个数据来表示一个容器

\_Start\_of\_storage \_ \_First \_Last \_End\_of\_storage

这样设计和vector一样存在扩容问题，不管是首部空间还是尾部空间用尽，都需要新增空间，但是为了维护连续性！不得不重新申请一块更大的内存，而不是说我申请一块新内存拼接上去。当元素数量非常多的时候，而且has\_trivial\_copy\_constructor 为false，这样复制旧元素的代价是非常大的。

所以，如果与vector保持一样的设计，那deque容器完全没必要出现了，我们修改vector容器就行了。一般情况下使用vector，需要在首部插入使用insert就行了。新的deque应该要对vector的缺点进行改进，以适应需要频繁在首尾插入的场景需求。

如上分析，需要解决的就是扩容的问题，而扩容主要的问题在于，如何维护一种逻辑上的内存空间连续的现象。



这时候看这张图就很有意思了，这样的结构避免了扩容时需要重新移动所有元素的问题！只需要新申请一块内存然后挂载再map下面，有点内存池的味道。

这样设计，主要问题就在于迭代器的移动。 因为vector的迭代器就是普通指针，普通指针的能力也刚好能够用在vector的结构上。那么deque的迭代器显然就不能是普通指针了，需要针对这个结构进行设计。

->**分析2**：基于底层数据结构进行迭代器的设计

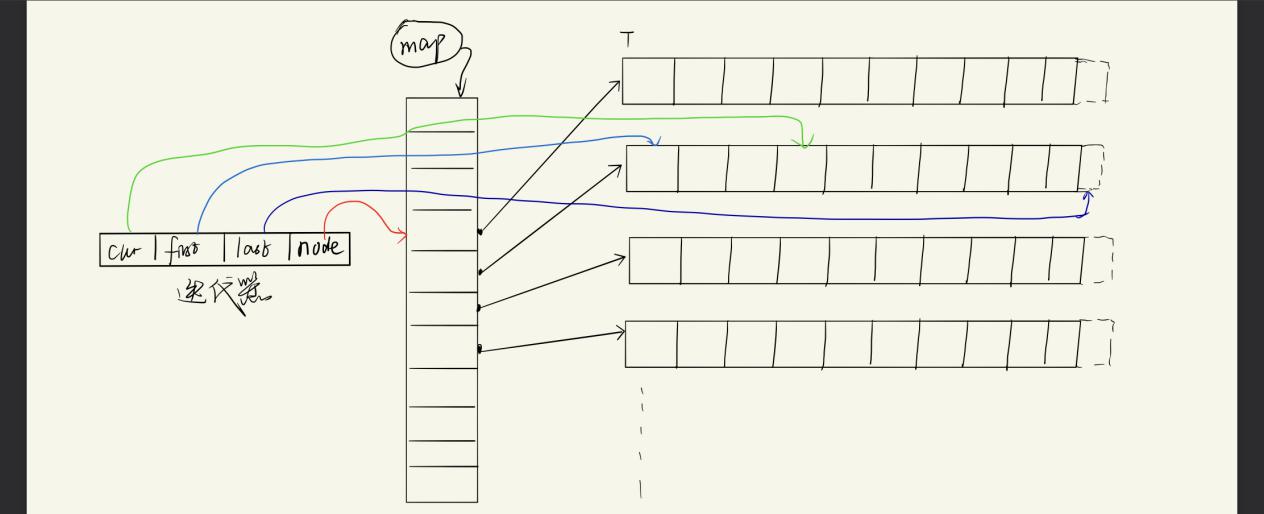
问题核心就是如何存取和行进：

1. 如何定位到某一个缓冲区
2. 如何取出缓冲区中某一个元素
3. 如何向缓冲区中添加元素

3.1 如果缓冲区满了，如何移动到下一个缓冲区

3.1.1 如何判断缓冲区满了

3.2 如果非要往已经满的缓冲区插入，那么元素如何移动



定位缓冲区： node = map+n

如何取出元素： \*cur （重载了\*操作符就行）

如何向缓冲区添加元素： 说到底还是allocate(ptr,val)

判断满了没：cur == last（后插） cur==first（前插）

移动到下一个缓冲区： node = map+n+1；从上一个位置到下一个位置就行，然后重新赋值first、last、cur等

**开发3**：详细设计

->**分析1**：迭代器构造函数

什么情况下会调用迭代器的构造函数呢！

拷贝构造函数是调用最频繁的，因为当迭代器作为成员函数的参数时，因为是传值的方式，所以会调用拷贝构造函数。

->**分析2**：迭代器的各项操作：

参考C++标准程序库一书，上面列举了random access iterator具备的操作

|  |  |  |
| --- | --- | --- |
| 表达式 | 效果 | 备注 |
| ☑\*iter | 存取实际元素 | 集合了  input iterator  output iterator  的功能 |
| ☑iter->member | 存取实际元素的成员 |
| ☑++iter | 向前进（传回新位置） |
| ☑iter++ | 向前进（传回旧位置） |
| ☑iter1 == iter2 | 判断两个迭代器是否相等 |
| ☑iter1 != iter2 | 判断两个迭代器是否不等 |
| ☑TYPE() | 产生迭代器（default构造函数） |
| ☑TYPE(iter) | 复制迭代器（copy构造函数） |
| ☑iter1 = iter2 | 赋值 |
| ☑--iter | 后退（传回新位置） | 双向迭代器增加的功能 |
| ☑iter-- | 后退（传回旧位置） |
| ☑iter[n] | 存取索引位置为n的元素 | 在双向迭代器的基础上，随机存取迭代器新增的功能 |
| ☑iter+=n | 向前跳n个元素（如果n为负，则向后跳） |
| ☑iter-=n | 向后跳n个元素（如果n为正，则向前跳） |
| ☑iter+n 和 n+iter | 传回iter之后的第n个元素 |
| ☑iter-n | 传回iter之前的第n个元素 |
| ☑iter1 - iter2 | 传回iter1和iter2之间的距离 |
| ☑iter1 < iter2 | 判断iter1是否在iter2之前 |
| ☑iter1 > iter2 | 判断iter1是否在iter2之后 |
| ☑iter1 <= iter2 | 判断iter1是否不在iter2之后 |
| ☑iter1 >= iter2 | 判断iter1是否不在iter2之前 |

->**辨析研究1**：关于this指针的相关问题

重载运算符时，大量出现return \*this，但是概念很模糊，今天总结辨析一下。

参考博客： https://www.cnblogs.com/zhengfa-af/p/8082959.html

this指针的用处：

一个对象的this指针并不是对象本身的一部分，不会影响sizeof(对象)的结果。this作用域在类的非静态成员函数中访问类的非静态成员时，编译器会自动将对象本身的地址作为一个隐含参数传递给函数。

this指针的使用：

一是，在类的非静态成员函数中返回类对象本身时，直接使用return\*this

二是，当参数与成员变量名相同时，this->xxx = xxx，而不是写成xxx=xxx

this指针程序示例：

this指针存在于类的成员函数中，指向被调用函数所属于的类实例的地址

this指针的特点：

①this指针只能在成员函数中使用，全局函数、静态函数都不能使用this

实际上，成员函数的第一个参数为 T\* const this。某个成员函数 int fun(int p){}，在编译器看来应该是：int func(A\* const this, int p)

②this在成员函数的开始前构造，在成员函数的结束后清除

这个生命周期同任何一个函数的参数都是一样的，没有任何区别。当调用一个类的成员函数时，编译器将类的指针作为函数的this参数传递进去。如：

A a; a.func(10);

编译器会编译成： A::func(&a, 10);

似乎和static函数没差别，但是编译器会对this指针做一些优化。（VC通常是通过ecx寄存器传递this参数的）

2019.12.8

**辨析研究1**：右值引用与移动语义（内容挺多的..）

参考书籍《深入理解C++11》

移动语义，说白了就是把别人的资源移为己用。问题的引发看如下例子：

Class A

{

....

}

A GetTemp() { return A();}

A a = GetTemp();

A a= GetTemp()这一句话可以分解成很多步骤，我们从右往左看：

①调用GetTemp函数，函数内部调用默认构造函数，创建了一个匿名的对象

②GetTemp函数返回的是一个对象，所以会引发一次拷贝构造，创建一个临时对象Temp

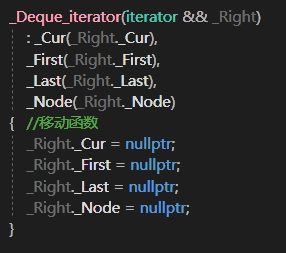
③赋值语句，把临时对象Temp赋值给a变量，这又引发了一次拷贝构造，以临时变量Temp为值进行构造。

我们知道临时对象Temp的生命周期是很短暂的，当函数GetTemp结束时，临时对象Temp也就被析构了。这导致我们无法在后续继续使用这个对象，为了能够继续使用该对象，就只有重新再构造一个对象了。

但是这有一个问题，如果对象占的资源非常多，那么拷贝构造这一过程对性能的影响就非常大了。而且可以看到，第二次调用拷贝构造是出于无奈的选择，因为临时对象会被析构。那么这个时候，就可以引入右值引用了~

使用右值引用时，我们可以把Temp的寿命延长任意你需要的时间，在此基础之上，我们就可以避免一次拷贝构造！

因为使用了右值引用，临时对象得以存活，这个时候，我们其实可以直接夺取临时对象Temp的资源！



->**问题1**：返回一个临时对象的右值引用，这个临时对象什么时候被析构呀?

需要看函数中的对象类型，如果是new出来的对象，则不会被析构。建议传递引用。

**开发1**：完善迭代器的内容