### 记录时间：2019.11.11~2019.11.17（一周）

2019.11.11

**问题1：无参构造和push\_back之间的问题**

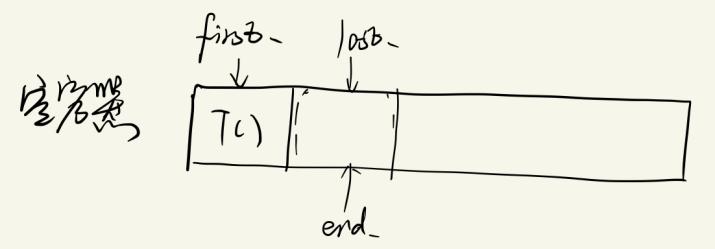
template<typename T, typename Alloc>

vector<T, Alloc>::vector() {

allocate\_and\_fill\_n(1, value\_type());

}

first\_ 和last\_ 之间有一个元素的距离诶！



但是push\_back是针对容器末尾插入的，也就是在last\_位置插入的。所以无参构造函数还是需要改成

template<typename T, typename Alloc>

vector<T, Alloc>::vector() {

first\_ = data\_allocator::allocate();

last\_ = end\_ = first\_;

}

**问题2：push\_back 进行大量数据插入时，次数一多就出现写入异常**

扩容问题：

vector扩容逻辑似乎被我忽略了，因为push\_back每次都是只插入一个元素，按照我之前的写法，扩容策略是 old\_size+ max（need\_size，old\_size）。也就是说每次扩容后只能刚好容纳下旧数据和新数据，下次插入时又满了....可想而知，效率得多低。

引起扩容的情况有如下：

①reverse()、resize()函数

②push\_back() 当容器存满时

③insert() 容器剩余容量无法满足新数据大小

①reverse、resize

reverse() 具体容量大小由调用者确定，只有当新容量大于capacity()时，才会扩容到要求大小，否则不作为。

resize(n) 具体容量也由调用者确定，n< size()时，析构掉超出的部分；size()<n<capacity()时，从[last\_, last+n)进行T() 数据填充即可；n>capacity()时，重新申请内存大小为n的空间

②push\_back

因为push\_back()函数每次都只插入一个元素，所以当插满时，扩容后的大小必须得能保证进行多次插入，否则频繁的内存申请、移动、释放会影响效率

所以push\_back插满后，扩容策略可以是 new\_size = 2\*capacity();但是要注意capacity()为0的情况

③insert

insert的情况其实包含了push\_back，只是insert插入数量、位置都不确定。可能第一次插入一个元素，但是第二次调用就插入10000个。

针对这种情况，扩容策略可以是new\_size = capacity()+ 2\*need\_size; 注意capacity()为0的情况

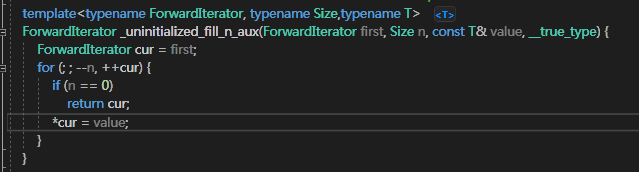
插入次数多导致写入异常问题：

可能有的情况:

①频繁插入导致内存申请、释放等环节出现纰漏，导致内存资源不足？

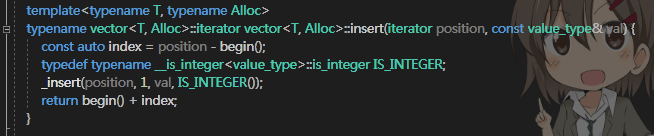
②...

调试运行时，提示\*cur = value 写入异常，那么肯定就是++cur 移动到了不能访问的内存，也就是说传入的以first为开头的内存空间可能不足。

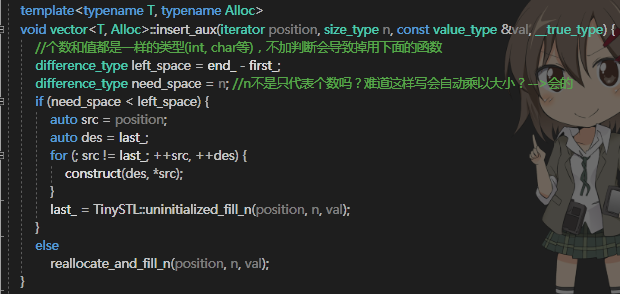


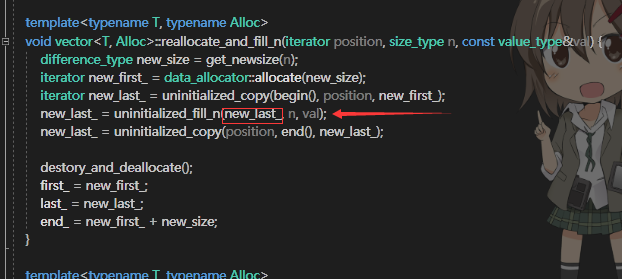
分析：

我对容器进行了10000次push\_back插入操作，但是到了532次就出现异常了。我的push\_back是调用insert的，只传了一个end() 和值进去



最终调用的是如下，也就是说是因为need\_space>left\_space导致调用reallocate\_and\_fill\_n引起扩容操作导致错误的





有很大的概率是 uninitialized\_copy 出了错误

仔细调试了一翻，问题不在uninitialized\_copy，而是vector.h中相关辅助函数有问题

2019.11.12

**问题1：push\_back插入自定义类型数据出现错误**

昨天重写了一些辅助函数之后，可以对POD类型的数据进行插入，但是对象自定义类型的数据插入有问题。

template<typename T, typename Alloc>

template<typename InputIterator>

① void vector<T, Alloc>::insert(iterator position, InputIterator first, InputIterator last) {

typedef typename \_\_is\_integer<InputIterator>::is\_integer IS\_INTEGER;

insert\_aux(position, first, last, IS\_INTEGER());

}

template<typename T, typename Alloc>

② void vector<T,Alloc>::insert(iterator position, size\_type n, const value\_type& val){

typedef typename \_\_is\_integer<size\_type>::is\_integer IS\_INTEGER;

insert\_aux(position, n, val, IS\_INTEGER());

}

第二个函数一开始是我是写的 \_\_is\_integer<value\_type>::is\_integer IS\_INTEGER; 这样的话其实IS\_INTEGER()表示的是\_\_false\_type

实际上push\_back(类对象)时，应该调用的是下面这个函数...

void vector<T, Alloc>::insert\_aux(iterator position, size\_type n, const value\_type &val, \_\_true\_type)

说一下我自己关羽①②函数的理解吧，在我调试的过程中，不管你输入的参数类型是什么，都会匹配到①，然后从①里面进行判断。

**开发：设计一个测试框架**

需要一个测试框架，否则后续出现vector能跑，但是list不能跑就糟糕了。用累计测试法，每次增加新的容器都要连带已经开发的部分一起测试。

。。。

了解到了gtest测试框架，准备学习一下并且使用

2019.11.13

准备结束vector的开发，还剩下几个函数以后再说。准备开启list容器的编码。

回顾总结分析vector容器开发的经验：

①不要着急动手写代码，务必先设计容器框架和函数依赖

②先了解list有哪些函数，具体怎么用

2019.11.14~15

14号对list容器大概有个了解了，具体涉及到哪些函数，有什么作用等

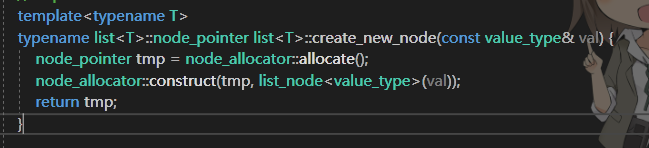
15号：

list的容器的迭代器，是list\_iterator的一个实例化对象，这个对象包含了list\_node这个对象成员。List.p代表的就是一个节点。

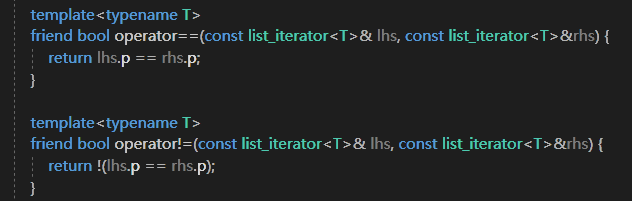


当你定义了析构函数却没写析构函数时，其实已经提示的很明显了。 ~list

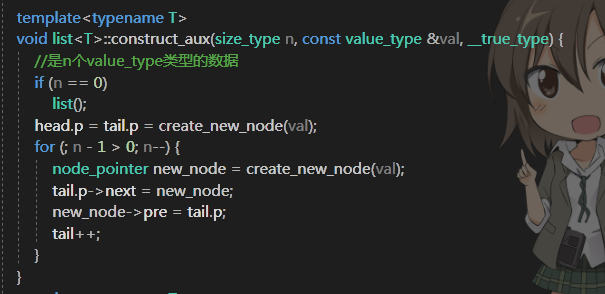
如果初始化没有赋值，那么就是用的默认值，后续可以通过访问容器中元素来进行修改。如vector<test> v; v[x] = xx;

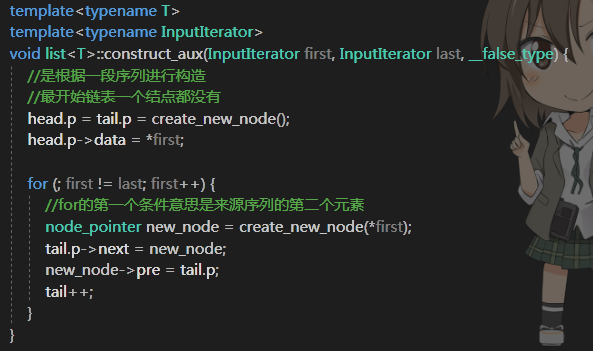


因为链表的插入是一个一个结点插入的，每次插入时才构建结点。申请内存时，也就只需要申请一个结点的内存大小。关于这个结点的内存大小的计算在我们定义node\_allocator时就说明了。 typedef allocator<list\_node<T>> node\_allocator，所以node\_allocator 申请内存的大小单位就是list\_node<T>对象的大小（4+4+sizef(T),两个指针嘛~）



如果上述重载函数不写出friend，则使用lhs. 没有相对应的成员可以用，如果只有一个参数的话就可以。这其实涉及到相关重载的知识点。





构造函数我是这样想的，我用create\_new\_node把结点创建好，只需要进行一些连接操作就好了。但是有个问题就是，想用tail.p->next = new\_node;这种类似于链表插入新结点的操作时，必须是有一个结点在的！但是一开始链表哪里有结点。这就需要额外的操作来处理这个问题。

解决方法就是，每一个操作单元负责 为前一个结点复制同时为下一结点申请空间。

妈的，这样设计又有问题，比方说我要创建一个含有2个结点的链表，我给最后一个结点赋值的时候还会再创建一个结点。。。会多出一个结点来。

最终解决方法是把这些问题都放在push\_back里面处理。

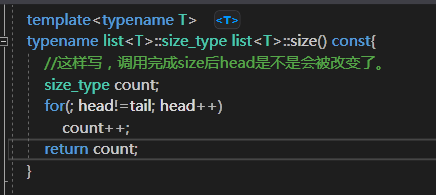
问题tail不是指向最后一个结点的下一个位置，搞的head !=first 时老是要单独拿出来判断一下。。。所以说上面那个多出一个结点的操作还是很有必要的！

因此还采用 **每一个操作单元负责 为前一个结点复制同时为下一结点申请空间**。这么一个解决方案。

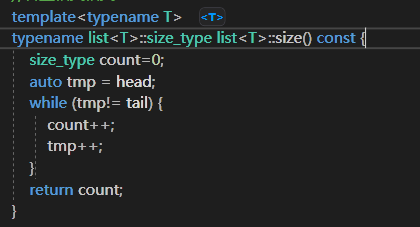
2019.11.16

1.





这个问题出现好几次了，不能对head进行++操作吗？因为size()用const修饰了，所以无法修改成员变量，一旦修改就会报错，把const去掉就可改。



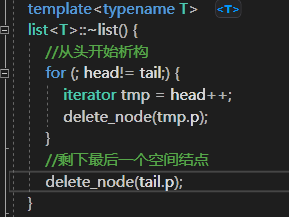
2.delete\_node功能超出本身工作范围

delete\_node本意应该是删除一个结点的，不应该让它承担删除整个链表的功能，这样局限性太大，无法被其他地方使用。

重新设计delete\_node，删除传入的位置那个结点。作为一个辅助函数来讲，主要是为public成员函数使用，一般都是用来删除头部，尾部等操作，如果说是删除链表中间某一结点，还是需要遍历链表。

**又用错了函数，allocator里面提供的函数，带n的都是指元素个数！不是大小！**

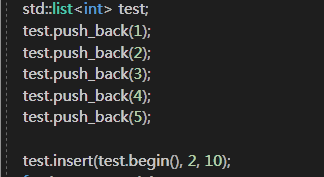
新认知tail其实一直指向结尾的下一个位置，是一个空结点。

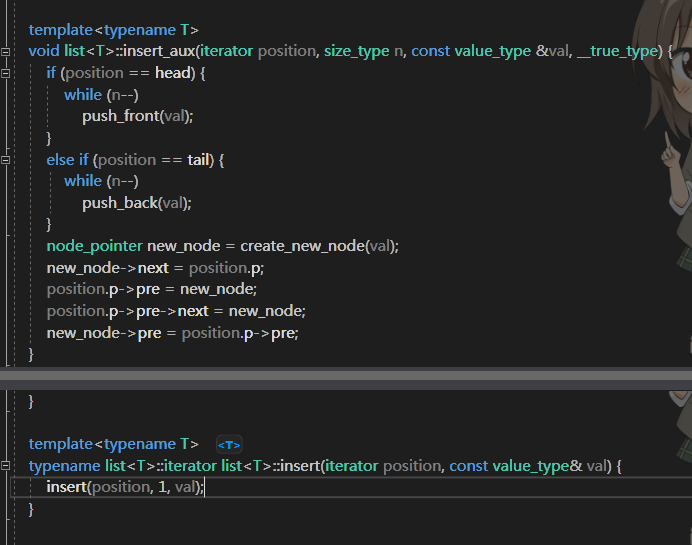


每删除一个结点，head必须更新指向下一个结点，如下写法，第一次师傅了tmp.p后，head这个迭代器就失效了... 因为它指向并没有移动.



1. clear函数，会清空所有的真实结点，但会保留伪结点
2. insert函数，插入位置是在position之前

结果是10 10 1 2 3 4 5



设计不够优秀，\_\_true\_type 和\_\_false\_type写法和功能上高度重复了，而且需要返回的没有返回值。

我的想法是position不动，就一直在position前一个位置插入就行。

position

↓

□□（）□ 每次都往（）里插入即可

2019.11.17

因为使用到了仿函数，所以决定先整体系统的学习一下防函数的概念