作业报告

自动化（控制）韩箫；3230100653

2024.10.20

1 测试程序的设计思路

1. 使用初始化列表构造函数和拷贝构造函数，来创建多个 List 对象，并打印每个 List 对象的内容，直接观察验证构造函数的正确性。
2. 使用赋值运算符，将一个 List 对象赋值给另一个 List 对象。打印赋值后的 List 对象的内容，验证重载后的拷贝赋值函数。并进行连续赋值，验证拷贝复制的返回情况是否正确。再进行自身赋值，检验正确性。
3. 调用移动构造函数，使用 std::move 将一个 List 对象移动到另一个 List 对象，输出内容来验证移动构造函数的正确性。同理进行移动赋值的测试。
4. 调用 push 函数，实则也对内嵌的 insert 函数进行了调用和检验。
5. 使用 pop 方法从 List 对象的前后删除元素，同理也同时检验了 remove 函数。
6. 使用 front 和 back 方法访问已知对象的第一个和最后一个元素，直接观察判断检验。
7. 使用 clear 方法清空 List 对象，并打印空列表进行检验。
8. 使用 isEmpty 方法检查预先已知的一个空链表和一个非空链表，观察结果是否符合预期。
9. 使用 getsize 方法获取一个已知表的大小。
10. 最后调用析构函数进行清理，并为防止重复清理增加判断和报错。

2 测试结果

测试结果如图 1 所示，测试结果正常，符合直观预期。
我用 valgrind 进行测试，发现没有发生内存泄露。测试报告如图 2 所示

3 bug 报告

我发现了一个 bug，触发条件如下：

1. 如果不在 clear 以及 remove 函数中增加对 head 和 tail 情况的判断和相应报错，在析构函数中会重复释放内存，导致 Segmentation fault，如图 3所示。
2. 通过我在 begin 函数中所加的报错语句，可见此时有 head 指针变为了空指针。而教材中目前授课进度所涉及的部分还未包含错误检验。所以若完全遵照教材代码, 就会出现该问题。
3. 如果在 clear 函数中增加检验和反馈的语句，如图 4 所示，会发现在 remove 函数中的 head 和 tail 已经被清理，因此导致重复释放内存。

```
summer_hare@localhost:/mnt/d/浙大本科学习/大二上/数据结构与算法分析/My Project/List$ ./test
1       2       3       4       5
1       2       3       4       5
6       7       8

6       7       8
6       7       8
1       2       3       4       5
9       10
9       10
11      12      13      14      115
12      13      14
Front of h: 12
Back of h: 14
List h after clear

Is h empty? Yes
6       7       8
Is a empty? No
Size of a is 3
it's been cleaned before
```

图 1: 测试结果

```
==102740== HEAP SUMMARY:
==102740==     in use at exit: 0 bytes in 0 blocks
==102740==   total heap usage: 227 allocs, 227 frees, 21,159 bytes allocated
==102740==
==102740== All heap blocks were freed -- no leaks are possible
==102740==
==102740== For lists of detected and suppressed errors, rerun with: -s
==102740== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

图 2: valgrind 测试报告

3.1 原因分析

1. 如前所述，在 begin 函数中增加了报错语句，可以看到 head 指针变为了空指针，但在 remove 函数中没有对这种情况进行相应的处理。

2. 经过后来的调试，发现程序在输出 a.getsize() 之后应该已经完成了所有的操作，但仍然出现了 segmentation fault。这可能是由于程序在退出时调用了某些析构函数，而这些析构函数中可能存在对无效指针的访问。即在主程序所有的显式的函数都完成调用之后，仍然会发生 begin 的报错，由此推断应当是析构函数的问题。因为只有析构函数在所有显式的函数调用完成后还会被调用。

3. 确定析构函数后再倒推，可能是移动后的对象状态：在调用 std::move 后，原对象的状态是未定义的。

4. 在调用 std::move(b) 后，b 的状态确实是未定义的，因此不应该再使用该链表。同时为了防止错误，应当在代码中添加检查，在访问之前确认其状态。

3.2 解决方法

所以最终我在 remove 以及 clear 中加上了相应的判断和检验语句，如图 4，问题得到解决。

4 代码细节改动或改进

因为是自己写了一遍完整的代码，所以有一些地方和教材会有一定出入和改动，再此罗列说明。

```
1       2       3       4       5
1       2       3       4       5
6       7       8

6       7       8
6       7       8
1       2       3       4       5
9       10
9       10
8       9       10      11
9       10
Front of b: Error: head or head->next is nullptr in begin()
Segmentation fault (core dumped)
```

图 3: Segmentation fault

```
242     void clear()
243     {
244         if(head == nullptr){
245             std::cerr << "it's been cleaned before" << std::endl;
246             return;
247         }
248         else
249         //remove(begin(),end());
250         remove_several(begin(),end());//我调用了remove一段，应该和逐个pop_front等价？
251     }
252
```

图 4: clear 检验

1. 初始化函数 initialization 中，如果不对 iterator 的 current 指针进行相应的初始化，可能会为空，或者原有内容清空后变为未定义，是否会引发一些错误？因此增加了初始化 current 为 tail 的语句，如图 5，之所以将 current 初始化为 tail 而非 head，主要是因为考虑 insert 函数是在 current 所指节点之前插入新节点。

2. 另外，经过编译发现，如果按教材代码所写，不明确指出所用的 current 是在哪个迭代器之下，编译器也会报错，如图 6 所示。所以在自增等几个迭代器函数中都明确写了 this->current，证明实际操作与教材仍存在差异性。

3. 要求思考 remove 一串节点的情况，内外循环的逻辑相似，可以进行优化，因此我在 LIst.h 的 line214-234 写了另一个版本的 remove-several 函数，和 remove 单个节点的逻辑略有不同，应该能减少一些指针操作？

4. 在 clear 函数中，教材代码给的是 while 循环下的 pop-front 函数，我直接使用了 remove-several 函数，经过测试检验效果应该是等价的。

5. 关于 push 和 pop 函数，本身内在逻辑是调用 insert 和 remove 函数；但 insert 和 remove 函数本身已经定义了左值和右值两种版本，所以 push 和 pop 函数左值和右值版本，好像也没有必要再调用 move 函数对参数进行转换，结果来看是等价的。

6. 移动赋值函数似乎没有被拷贝赋值所替代，不写的话还是会报错，和课上讲得有出入？所以还是写了一下，另外教材代码中的赋值都没有对自身赋值进行判断，所以我都加了一句判断。

```
void initialization()
{
    head = new Node;
    tail = new Node;
    head->next = tail;
    tail->pre = head;
    const_iterator(tail);//如果不加这句，空链表直接insert会发生问题？
    size = 0;
}
```

图 5: 初始化 current

```
List.h: In member function 'List<T>::iterator& List<T>::iterator::operator++()':
List.h:99:13: error: 'current' was not declared in this scope
   99 |             current = current->next;//current 为 nullptr 或 tail 情况?
      |             ^~~~~~~
```

图 6: current 归属

4.1 初学者水平有限，不当修改之处和不完善的想法，恳请指正包涵

5 附记部分心得和细节总结

因为感觉对数据结构和 C++ 语言本身不是很熟悉，所以还是自己写了一遍完整的程序，以期有更深入的记忆和感觉。在这个过程中，确实一点一点打代码，会比直接看教材代码，对于细节会有更多的关注和思考。通过和 gpt 的讨论，对一些知识细节有了更深入的理解。让 gpt 基于对话问答生成了一份知识点总结，下附于此。

- **迭代器的 operator++() 中 *this 返回值是什么？**

  在迭代器的 operator++() 方法中，return *this 返回的是调用 operator++() 这个函数的对象本身，而不仅仅是 current 成员。这是因为 this 是一个指向当前对象的指针，*this 就是这个对象本身。当你执行 current = current->next; 后，你改变了对象的 current 成员，但这个对象可能还有其他的成员。

- **迭代器的 * 操作符为什么不直接返回节点？**

  迭代器的 * 操作符通常用来获取当前迭代器指向的元素的值，而不是获取迭代器本身或者底层的节点。这是因为迭代器的设计目的是为了让用户能够方便地访问和遍历容器中的元素，而不需要关心底层的实现细节。

- **函数参数的左值和右值输入？**

  如果函数的参数是一个普通类型（非引用类型），那么无论你传入的是左值还是右值，都可以正常调用。但是，如果函数的参数是一个引用类型，那么情况就会有所不同。如果参数是一个非常量左值引用，那么你只能传入左值。如果参数是一个常量左值引用，那么你可以传入左值或右值。如果参数是一个右值引用，那么你只能传入右值。

- **for 循环中的迭代器是不是会被清理无法自增**

  在 for 循环中，循环体会先执行，然后才会执行循环变量的更新操作（即 iter++）。在你的代码中，如果 remove 函数会删除 iter 当前指向的节点并释放它的内存，那么在执行 iter++ 时，iter 可能就会指向一个已经被释放的内存，这将导致未定义的行为。

- **如何正确引用 std::initializer_list<T>？**

  在 C++ 中，std::initializer_list<T> 的对象通常是只读的，不能被修改，因此通常我们会使用左值引用或者常量左值引用来引用它，而不是右值引用。

- **std::swap 和 std::move 函数在哪个头文件中定义？**

  std::swap 函数是在 <algorithm> 头文件中定义的。std::move 函数在 <utility> 中定义。

以上，感谢!