

1 测试程序的设计思路

首先, 我创建了一个空的 List 对象, 以便后续进行各种操作的测试。

2, 我通过 pushfront 和 pushback 的方法在链表的头部和尾部插入元素, 并且打印链表中的内容来验证插入是否成功。我的思路是插入四个元素, 分别用 pushfront 和 pushback 操作, 观察四个元素的排列顺序即可证明。

根据 “List elements after pushfront and pushback::5 10 20 25” 的输出, 发现插入成功了。

3, 我通过 size 和 empty 方法来验证链表的大小和是否为空。因为上文我插入了四个元素, 那么此时 size 的结果也应该是 4, 并且显然链表不是空链表, 那么 empty 的结果应该是 false。

根据 “size: 4 Is empty: No” 的输出, 发现 size 和 empty 方法的结果是正确的。

4, 我通过 front 和 back 方法来验证链表的头部和尾部元素是否正确。因为我插入了四个元素, 那么头部元素应该是第一个插入的元素, 尾部元素应该是最后一个插入的元素。

根据 “Front: 5 Back: 25” 的输出, 发现 front 和 back 方法的结果是正确的。

(补充) 我测试迭代器, 将 List1 中的元素插入到迭代器中, 观察迭代器的元素内容。

根据 “List elements: 5 10 20 25 ” 的输出, 发现迭代器结果正确。

5, 我通过 popfront 和 popback 方法来验证链表的头部和尾部元素是否被正确删除。因为我插入了四个元素, 那么我 popfront 和 popback 之后, 链表中应该只剩下两个元素。并且这两个元素应该是第二个和第三个插入的元素, 即 10 和 20。

根据 “After popfront and popback, size: 2 List elements after popfront and popback: 10 20 ” 的输出, 发现 popfront 和 popback 方法的结果是正确的。

6, 我通过 insert 的方法在链表插入元素, 并且打印链表中的内容来验证插入是否成功。我的思路是在链表表头插入一个元素 15, 然后打印链表中的内容, 观察此时链表的 size 并且观察 15 是否在表头。

根据 “After insert, size: 3 List elements after insert: 15 10 20” 的输出, 发现 insert 方法的结果是正确的。

7, 我通过 erase 的方法在链表删除元素, 并且打印链表中的内容来验证删除是否成功。我的思路是删除链表中头元素, 即刚才用 insert 插入的 15, 然后打印链表中的内容, 观察此时链表的 size 并且观察 15 是否在链表中。

根据 “After erase, size: 2 List elements after erase: 10 20” 的输出, 发现 erase 方法的结果是正确的。

8, 我通过 clear 的方法清空链表, 并且打印链表中的内容来验证清空是否成功。我的思路是清空链表, 然后打印链表中的内容, 观察此时链表的 size 是否为 0。

根据 “After clear, size: 0 ” 的输出, 发现 clear 方法的结果是正确的。

9, 我测试初始化列表构造函数。使用初始化列表构造函数创建一个新的链表 list2, 并初始化其内容为 1, 2, 3, 4, 5, 然后打印链表中的内容, 通过观察链表的 size 和内容, 来验证初始化是否成功。

根据 “List2 size: 5 List2 elements: 1 2 3 4 5” 的输出, 发现初始化列表构造函数的结果是正确的。

10, 我测试拷贝构造函数。使用拷贝构造函数创建一个新的链表 list3, 并将 list2 拷贝给 list3, 然后打印链表中的内容, 通过观察链表的 size 和内容, 来验证拷贝是否成功。

根据 “List3 size (copy of list2): 5 List3 elements (copy of list2): 1 2 3 4 5 ” 的输出, 发现拷贝构造函数的结果是正确的。

11, 我测试赋值运算符。我构建了一个新链表 List4, 然后将 List3 赋值给 List4, 然后打印链表中的内容, 通过观察链表的 size 和内容, 来验证赋值是否成功。

根据 “List4 size (assigned from list3): 5 List4 elements (assigned from list3): 1 2 3 4 5 ” 的输出, 发现赋值运算符的结果是正确的。

12, 我测试移动构造函数。我构建了一个新链表 List5, 然后将 List4 移动给 List5, 然后打印链表中的内容, 通过观察链表 5 的 size 和内容, 以及链表 4 的 size 是否为 0, 来验证移动是否成功。

根据 “List5 size (moved from list4): 5 List4 size after move: 0 List5 elements (moved from list4): 1 2 3 4 5 ” 的输出, 发现移动构造函数的结果是正确的。

13, 我测试前置自增和后置自增运算符。我取链表 2 中的头元素 1, 对它分别进行 `++iter`, `itre++`, 再 `iter++` 的操作, 分别观察三次的输出情况。

根据 “Initial value: 1 Value after `++iter`: 2 Value after `itre++`: 2 Value after `iter++`: 3” 的输出, 发现前置自增和后置自增运算符的结果是正确的。

15, 我测试前置自减和后置自减运算符。我取前文运算后得到的结果 3, 对它分别进行 `--iter`, `itre--`, 再 `iter--` 的操作, 分别观察三次的输出情况。

根据 “Initial value: 3 Value after `--iter`: 2 Value after `itre--`: 2 Value after `iter--`: 1” 的输出, 发现前置自减和后置自减运算符的结果是正确的。

16, 对于迭代器的测试, 我创建一个新链表 List6, 是 5,10,15,20,25。让迭代器 `itr1` 先指向第一个元素 5, 然后将第一个元素修改成-5, 输出两次结果后将迭代器指向下一个元素 10 并输出。然后我新建一个迭代器 `itr2`, 让 `itr2` 指向 List6 末端超出最后的一个元素的位置, 此时应该输出结果是 0, 然后再令 `itr2` 指向前一个元素 25, 输出结果应该是 25。然后我让 `itr1` 和 `itr2` 分别前进后退一个位置, 应该输出 10 和 20, 发现结果正确。最后分别测试与自身的相等, 他元素相等, 自身不等, 他元素不等, 判断 1 和 0 的 true or false。

2 测试的结果

测试结果一切正常。详细的结果见上文的测试程序的设计思路部分。

我用 `valgrind` 进行测试, 发现没有发生内存泄露。

3 (可选) bug 报告

我发现了一个 bug, 触发条件如下:

1. 首先……
2. 然后……
3. 此时发现……

据我分析, 它出现的原因是: