1 简介

在本次测试中，我创建了一个自定义链表类，涵盖了整数、浮点数和字符类型的链表。测试过程中，我向链表中插入了多种元素，并进行了删除、访问、插入、清空和异常处理等操作，以验证链表的功能和稳定性。通过调用 front() 和 back() 方法，检查链表的首尾元素，并验证其正确性。使用迭代器遍历链表中的元素，确保可以按预期访问所有元素。删除操作：对链表进行 popfront 和 popback 操作，以测试元素的删除功能。尤其是对字符链表的操作，删除后验证链表大小和元素状态。

异常处理：在空链表上调用 front() 和 back() 方法，捕捉异常，确保程序能正确处理非法操作。

拷贝和移动构造：

对链表进行拷贝构造和移动构造，测试在边界情况下的表现，例如从空链表进行拷贝或移动，检查目标链表的状态。确认拷贝构造后的链表大小和内容与源链表一致，移动构造后源链表应为空。范围删除：在字符链表上进行范围删除，验证清空列表的操作是否成功，并检查链表状态是否符合预期。

嵌套链表：创建一个外层链表 List<List<int»，并向其中插入内层链表，测试嵌套结构的操作，包括插入、访问和删除等。

2 测试程序介绍

下面将分段介绍我的测试程序。

```
void testList() {
    List<int> intList;
    ...
}
```

这段代码测试了不同类型链表的方法，包括 int、float 和 char 类型。

2.1 测试 int 类型列表

首先，测试 int 类型列表。

```
List<int> intList;
std::cout << "Initial size (should be 0): " << intList.size() << std::endl;
std::cout << "Is empty (should be true): " << intList.empty() << std::endl;
```

这段代码测试了空列表的初始状态。

```
intList.push_back(1);
intList.push_back(2);
intList.push_front(3);
std::cout << "After adding elements (size should be 3): " << intList.size() << std::endl;
intList.print();
```

这段代码测试了添加元素后列表的大小和内容。

```
intList.pop_front();
intList.pop_back();
std::cout << "After pop_front (size should be 5): " << intList.size() << std::endl;
intList.print();
```

这段代码测试了前后弹出操作。

```
intList.clear();
std::cout << "After clear (size should be 0): " << intList.size() << std::endl;
```

这段代码测试了清空列表的功能。

2.2 测试 float 类型列表

接下来，测试 float 类型列表。

```
List<float> floatList = {1.1f, 2.2f, 3.3f};
std::cout << "Float list size (should be 3): " << floatList.size() << std::endl;
```

这段代码测试了初始化列表构造。

```
floatList.push_back(4.4f);
floatList.push_front(0.0f);
std::cout << "After adding float elements (size should be 5): " << floatList.size() << std::endl;
```

这段代码测试了对 float 列表的添加操作。

2.3 测试 char 类型列表

然后，测试 char 类型列表。

```
List<char> charList;
std::string letters = "wanghengning";
for (char ch : letters) {
    charList.push_back(ch);
}
```

这段代码测试了向字符列表添加元素。

2.4 测试嵌套链表

最后，测试嵌套链表。

```
List<List<int>> outerList;
List<int> innerList1;
innerList1.push_back(1);
outerList.push_back(innerList1);
```

这段代码测试了外层链表存放内层链表的功能。

3 测试结果

都在预期之内，全部正常输出，函数没有遗漏，功能合理得当。

4 bug 报告

我发现了几个 bug，如下：

1. 首先，您可以看到目录里有一个东西叫 bug.cpp，里面有六个代码块，是我猜测会出问题的操作，但是我试过了，没出大问题。

2. 然后，在运行 bug 之后输入 6(回车)，会出现一个随机数，虽然系统没有崩溃但是这个随机数是不对的。说明访问到了不该访问的东西。可能对某些文件安全有威胁。

据我分析，它出现的原因是：先获取一个位置，然后释放了这个位置，再次访问这个位置。解决方案是：在释放这个位置之后，将这个位置置为 NULL，或者在检测的时候先确定是否是空指针。进一步的，其实可以注意到 list.h 中很多地方没有进行空指针检测。相比之下上周的那个搞得比较好。

1. 另外一个 bug 是 erase 的鲁棒性不够强，您可以在 bug.cpp 里使用 7 查看

2. 试图在一个链表上调用 erase(i,j)，其中 i，j 超出了 erase 的范围，如 j=++mylist.end(), 这会导致程序崩溃，相对来说可能跳过操作，或者像 python 一样尽可能从头删到尾更有鲁棒性

据我分析，它出现的原因是：from 和 to 指向的是相邻的节点（例如在一个只有 3 个节点的列表中删除 begin++ 到 end++），这会导致访问越界或对已释放内存的访问。可以考虑在 erase 方法中传递 to 的地址，以确保在删除节点后不会尝试访问无效的迭代器。

1. 第三个 bug 和第一个 bug 有点像。您可以在 bug.cpp 里使用 8 查看

2. 先创建 list，填充，clear，然后 print，会显示一段乱码。猜测是读取到了不该读的内存

据我分析，它出现的原因是：print 方法未检查链表是否为空，直接访问了已释放的节点。解决方案是：在 print 方法中添加空指针检查，确保在链表为空时不进行打印操作。这样可以避免访问无效内存并提高代码的鲁棒性。

1. 还有一点，试图创建 List<List<int»，也就是” 链表链表” 的时候，操作极其麻烦，访问成员链表操作复杂且复用性差

2. 而且这个嵌套的东西，直觉上就感觉有很多 bug……在今后的学习中希望能得到更好的解决方案，也恳请助教和老师多多提意见