

**Lab Report**

**实验报告**

|  |  |
| --- | --- |
| **Course**: | Class Libraries and Data Structures |
| **Semester**: | 1st semester of the academic year **2024-2025** |
| **Major**: | Software Engineering |
| **Class**: | 2023 |
| **Student Name**: | 吴仲雄 |
| **Student ID:** | 222023321062007 |
| **Teacher:** | ZHAO, Hengjun (赵恒军) |

**School of Computer and Information Science**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | | Complete Implementation of the List Container in STL  STL中list容器的完整实现 | | | |
| Date | | Nov，2024 | Type | | ☑Confirmatory （验证确认型）  ☑Design（设计型）  🗆Comprehensive（综合型） |
| 1. **Objective & Requirements（实验目的）**    1. Further improve the ability of developing sequential containers and using sequential containers in applications.   进一步提升顺序容器的设计开发能力，以及应用顺序容器解决实际问题的能力   * 1. Improve the development skills of iterators.   提升对迭代器的理解和对迭代器设计实现能力的掌握   * 1. Grasp various manipulations of linked list structure. Grasp the design of storage management and operations.   熟练掌握链表数据结构的各种操作，掌握底层存储的管理。   * 1. Gain a complete understanding of principles of C++ containers and the Standard Template Library.   获得对C++容器原理和C++标准模板库STL底层原理的全面理解。 | | | | | |
| 1. **Experimental environment (**platform and software**)（实验环境）**   Windows 7 (or higher versions) + Visual Studio 2010 (or higher versions) | | | | | |
| 1. Experimental content and design (Main Content, Procedure, Codes and Results)（此部分应包含每一个实验内容的详细设计，含实验思路、详细实验步骤、核心代码说明等）   Task 1  You are required to implement your own generic list container that has the same funcionalities as in the Standard Template Library (STL), for which you can reference：<https://cplusplus.com/reference/list/list/>  or  <https://en.cppreference.com/w/cpp/container/list>  The detailed requirements are as follows:  本任务要求参考STL中list容器的功能（可参考<https://cplusplus.com/reference/list/list/>或<https://en.cppreference.com/w/cpp/container/list>  ）给出一个通用链式容器较为完整的实现。具体要求如下：   * You should use doubly linked list storage with head pointer and tail pointer.     要求使用如图所示的带有头、尾指针的双向链式存储结构   * You should use new() and delete() operations to allocate and release node storage   要求使用new()和delete()方法动态创建和释放存储空间   * Your design of the list class and associated iterator class must provide the following 22 methods   所设计的容器类和迭代器类必须按要求实现并测试后文列表中的所有方法（22个），测试用例可参考<https://cplusplus.com/reference/list/list/>  或  <https://en.cppreference.com/w/cpp/container/list>  例如对splice功能的测试可参考：  <https://cplusplus.com/reference/list/list/splice/>      =============================================================================  1.  //Postcondition: this list is empty  list ();  Note: This default constructor is usually invoked implicitly, for example,  list<Employee> employees;  makes employees an empty list, whose items will be of type Employee.  2.  //Postcondition: this list has been destroyed  ~list ();  3.  //copy constructor  //Postcondition: this list has been constructed and initialized to a copy of x  //The worstTime(n) is O(n), where n is the size of x.  list (const list<T>& x);  Note: this kind of constructor is referred to as a copy constructor  4.  //Postcondition: x has been inserted at the front of this list.  void push\_front (const T& x);  5.  //Postcondition: x has been inserted at the back of this list.  void push\_back(const T& x);  6.  //Postcondition: x has been inserted in this list **in front of** the item that position was  //positioned at before this call. An iterator positioned at x has been returned.  iterator insert (iterator position, const T& x);  Note: The worstTime(n) is constant.  7.  //Precondition: this list is not empty  //Postcondition: the item that was at the front of this list before this call was made  //has been deleted from this list.  void pop\_front();  8.  //Precondition: this list is not empty  //Postcondition: the item that was at the back of this list before this call was made  //has been deleted from this list.  void pop\_back();  9.  //Precondition: position is positioned at an item in this list  //Postcondition: the item that position was positioned at before this call was made  //has been deleted from this list  //return value: An iterator pointing to the element that **followed** the element  //erased by the function call.  iterator erase (iterator position);  Note: The worstTime(n) is constant.  10.  //Precondition: first is positioned at some item in this list, and last is positioned one past  //some item in this list.  //Postcondition: all the items that, before this call was made, were in the range from first  //(inclusive) to last (exclusive) have been deleted from this list  //that is, remove the elements in the range [first, last).  //return: Iterator following the last removed element.  iterator erase (iterator first, iterator last);  Note: The time for this method is proportional to the number of items removed.  11.  //Postcondition: the number of items in this list has been returned.  unsigned size() const;  12.  //Postcondition: true has been returned if this list is empty; Otherwise, false has been returned.  bool empty() const;  13.  //Postcondition: an iterator positioned at the front of this list has been returned  //If the list is empty, the returned iterator will be equal to end(). (See 14.)  iterator begin();    14.  //Postcondition: an iterator positioned **AFTER** the last item in this list has been returned.  iterator end();  15.  //copy assignment: Replaces the contents with a copy of the contents of x.  //Postcondition: this list contains a copy of x,  //and a reference to this list has been returned  list<T>& operator=(const list<T>& x)  16.  //Postcondition: The contents of x have been inserted, starting at position (**in front of** position),  //into this list, and x is empty  void splice (iterator position, list<T>& x);  Note: This method takes constant time, no matter how big x is.  ===============member functions of iterator inner class==============================  17.  //Postcondition: this iterator is now positioned at the next position in this list,  //and a reference to this iterator has been returned.  iterator& operator++ ();  Note: This is the preincrement operator; that is, the iterator advances and a reference to the newly positioned iterator is returned. For example, suppose that cities is a list object that contains the following list of cities:  “Beijing”, “Shanghai”, “Chongqing”, “Chengdu”  If itr is a list iterator positioned at “Chongqing” and we write  List<string>::iterator new\_itr = ++itr;  Then both itr and new\_itr are positioned at “Chengdu”.  18.  //Postcondition: this iterator is now positioned at the next position in this list,  //and a copy of this iterator’s previous value has been returned.  iterator operator++ (int)  Note: This is the postincrement operator; that is, the iterator advances, but the iterator’s value before advancing is returned. The postincrement operator has an int parameter whose only purpose is to distinguish this operator from the preincrement operator. In fact, there is no argument corresponding to the int parameter. For example, suppose that cities is a list object that contains the following list of cities:  “Beijing”, “Shanghai”, “Chongqing”, “Chengdu”  If itr is a list iterator positioned at “Chongqing” and we write  list<string>::iterator old\_itr = itr++;  then itr is positioned at “Chengdu”, but old\_itr is positioned at “Chongqing”.  19.  //Postcondition: this iterator is now positioned at the previous position in this list,  //and a reference to this iterator has been returned.  iterator& operator--(); //pre-decrement  20.  //Postcondition: this iterator is now positioned at the previous position in this list,  //and a copy of this iterator’s previous value has been returned.  iterator operator--(int); //post-decrement  21.  //Precondition: this iterator is positioned at an item in this list.  //Postcondition: a reference to the item this iterator is positioned at has been returned.  T& operator\*();  Example: Suppose that itr is positioned at the item “Chongqing”. If we write  cout << (\*itr);  the output will be  Chongqing  Note: Because a reference is returned, we can use this operator to alter the value of an item in the list. For example,  \*itr = "Chongqing";  will change the value of the item itr is positioned at to “Chongqing”.  22.  //Postcondition: true has been returned if this iterator is positioned at the same place  //in this list x is positioned at. Otherwise, false has been returned.  bool operator== (const iterator& x);  list.h:  #ifndef LIST\_H  #define LIST\_H  template <typename T>  class list {  private:  // Define the node structure  struct Node {  T data;  Node\* prev;  Node\* next;  Node(const T& value = T(), Node\* p = nullptr, Node\* n = nullptr)  : data(value), prev(p), next(n) {}  };  Node\* head; // Head pointer  Node\* tail; // Tail pointer  unsigned list\_size; // Size of the list  public:  // Inner iterator class  class iterator {  private:  Node\* current;  public:  iterator(Node\* node = nullptr) : current(node) {}  // 21.  //Precondition: this iterator is positioned at an item in this list.  //Postcondition: a reference to the item this iterator is positioned at has been returned.  T& operator\*() {  return current->data;  }  // 17.  //Postcondition: this iterator is now positioned at the next position in this list,  //and a reference to this iterator has been returned.  iterator& operator++() {  current = current->next;  return \*this;  }  // 18.  //Postcondition: this iterator is now positioned at the next position in this list,  //and a copy of this iterator’s previous value has been returned.  iterator operator++(int) {  iterator old = \*this;  current = current->next;  return old;  }  // 19.  //Postcondition: this iterator is now positioned at the previous position in this list,  //and a reference to this iterator has been returned.  iterator& operator--() {  current = current->prev;  return \*this;  }  // 20.  //Postcondition: this iterator is now positioned at the previous position in this list,  //and a copy of this iterator’s previous value has been returned.  iterator operator--(int) {  iterator old = \*this;  current = current->prev;  return old;  }  // 22.  //Postcondition: true has been returned if this iterator is positioned at the same place  //in this list x is positioned at. Otherwise, false has been returned.  bool operator==(const iterator& x) const {  return current == x.current;  }  bool operator!=(const iterator& x) const {  return current != x.current;  }  Node\* getNode() const {  return current;  }  };  // 1.  //Postcondition: this list is empty  list() {  head = new Node();  tail = new Node();  head->next = tail;  tail->prev = head;  list\_size = 0;  }  // 3.  //copy constructor  //Postcondition: this list has been constructed and initialized to a copy of x  //The worstTime(n) is O(n), where n is the size of x.  list(const list<T>& x) {  head = new Node();  tail = new Node();  head->next = tail;  tail->prev = head;  list\_size = 0;  for (Node\* p = x.head->next; p != x.tail; p = p->next) {  push\_back(p->data);  }  }  // 15.  //copy assignment: Replaces the contents with a copy of the contents of x.  //Postcondition: this list contains a copy of x,  //and a reference to this list has been returned  list<T>& operator=(const list<T>& x) {  if (this != &x) {  clear();  for (Node\* p = x.head->next; p != x.tail; p = p->next) {  push\_back(p->data);  }  }  return \*this;  }  // 2.  //Postcondition: this list has been destroyed  ~list() {  clear();  delete head;  delete tail;  }  // 4.  //Postcondition: x has been inserted at the front of this list.  void push\_front(const T& x) {  insert(begin(), x);  }  // 5.  //Postcondition: x has been inserted at the back of this list.  void push\_back(const T& x) {  insert(end(), x);  }  // 6.  //Postcondition: x has been inserted in this list in front of the item that position was  //positioned at before this call. An iterator positioned at x has been returned.  iterator insert(iterator position, const T& x) {  Node\* posNode = position.getNode();  Node\* newNode = new Node(x, posNode->prev, posNode);  posNode->prev->next = newNode;  posNode->prev = newNode;  ++list\_size;  return iterator(newNode);  }  // 7.  //Precondition: this list is not empty  //Postcondition: the item that was at the front of this list before this call was made  //has been deleted from this list.  void pop\_front() {  if (!empty()) {  erase(begin());  }  }  // 8.  //Precondition: this list is not empty  //Postcondition: the item that was at the back of this list before this call was made  //has been deleted from this list.  void pop\_back() {  if (!empty()) {  iterator temp = end();  temp--;  erase(temp);  }  }  // 9.  //Precondition: position is positioned at an item in this list  //Postcondition: the item that position was positioned at before this call was made  //has been deleted from this list  //return value: An iterator pointing to the element that followed the element  //erased by the function call.  iterator erase(iterator position) {  Node\* posNode = position.getNode();  Node\* prevNode = posNode->prev;  Node\* nextNode = posNode->next;  prevNode->next = nextNode;  nextNode->prev = prevNode;  delete posNode;  --list\_size;  return iterator(nextNode);  }  // 10.  //Precondition: first is positioned at some item in this list, and last is positioned one past  //some item in this list.  //Postcondition: all the items that, before this call was made, were in the range from first  //(inclusive) to last (exclusive) have been deleted from this list  //that is, remove the elements in the range [first, last).  //return: Iterator following the last removed element.  iterator erase(iterator first, iterator last) {  while (first != last) {  first = erase(first);  }  return last;  }  // 11.  //Postcondition: the number of items in this list has been returned.  unsigned size() const {  return list\_size;  }  // 12.  //Postcondition: true has been returned if this list is empty; Otherwise, false has been returned.  bool empty() const {  return list\_size == 0;  }  // 13.  //Postcondition: an iterator positioned at the front of this list has been returned  //If the list is empty, the returned iterator will be equal to end(). (See 14.)  iterator begin() {  return iterator(head->next);  }  // 14.  //Postcondition: an iterator positioned AFTER the last item in this list has been returned.  iterator end() {  return iterator(tail);  }  // 16.  //Postcondition: The contents of x have been inserted, starting at position (in front of position),  //into this list, and x is empty  void splice(iterator position, list<T>& x) {  if (!x.empty()) {  Node\* posNode = position.getNode();  Node\* x\_first = x.head->next;  Node\* x\_last = x.tail->prev;  // Adjust x's nodes  x.head->next = x.tail;  x.tail->prev = x.head;  // Adjust current list's nodes  x\_first->prev = posNode->prev;  x\_last->next = posNode;  posNode->prev->next = x\_first;  posNode->prev = x\_last;  list\_size += x.list\_size;  x.list\_size = 0;  }  }  private:  void clear() {  while (!empty()) {  pop\_front();  }  }  };  #endif  test.cpp:  #include <iostream>  #include "list.h"  int main() {  // 1. 测试默认构造函数  list<int> lst1;  if (lst1.empty()) {  std::cout << "1. Default constructor test passed." << std::endl;  } else {  std::cout << "1. Default constructor test failed." << std::endl;  }  // 2. 测试析构函数（无法直接测试，程序结束时自动调用）  // 3. 测试拷贝构造函数  lst1.push\_back(10);  list<int> lst2(lst1);  if (!lst2.empty() && lst2.size() == lst1.size() && \*lst2.begin() == \*lst1.begin()) {  std::cout << "3. Copy constructor test passed." << std::endl;  } else {  std::cout << "3. Copy constructor test failed." << std::endl;  }  // 4. 测试push\_front  lst1.push\_front(5);  if (lst1.size() == 2 && \*lst1.begin() == 5) {  std::cout << "4. push\_front test passed." << std::endl;  } else {  std::cout << "4. push\_front test failed." << std::endl;  }  // 5. 测试push\_back  lst1.push\_back(15);  if (lst1.size() == 3 && \*(--lst1.end()) == 15) {  std::cout << "5. push\_back test passed." << std::endl;  } else {  std::cout << "5. push\_back test failed." << std::endl;  }  // 6. 测试insert  list<int>::iterator it = lst1.begin();  ++it;  lst1.insert(it, 8); // 插入到第二个位置  it = lst1.begin();  ++it;  if (\*it == 8) {  std::cout << "6. insert test passed." << std::endl;  } else {  std::cout << "6. insert test failed." << std::endl;  }  // 7. 测试pop\_front  lst1.pop\_front();  if (lst1.size() == 3 && \*lst1.begin() == 8) {  std::cout << "7. pop\_front test passed." << std::endl;  } else {  std::cout << "7. pop\_front test failed." << std::endl;  }  // 8. 测试pop\_back  lst1.pop\_back();  if (lst1.size() == 2 && \*(--lst1.end()) == 10) {  std::cout << "8. pop\_back test passed." << std::endl;  } else {  std::cout << "8. pop\_back test failed." << std::endl;  }  // 9. 测试erase(iterator position)  it = lst1.begin(); // 指向8  lst1.erase(it);  if (lst1.size() == 1 && \*lst1.begin() == 10) {  std::cout << "9. erase(iterator position) test passed." << std::endl;  } else {  std::cout << "9. erase(iterator position) test failed." << std::endl;  }  // 10. 测试erase(iterator first, iterator last)  lst1.push\_back(20);  lst1.push\_back(30);  it = lst1.begin(); // 指向10  list<int>::iterator it2 = lst1.end(); // 指向尾后  --it2; // 指向30  lst1.erase(it, it2); // 删除10,20  if (lst1.size() == 1 && \*lst1.begin() == 30) {  std::cout << "10. erase(iterator first, iterator last) test passed." << std::endl;  } else {  std::cout << "10. erase(iterator first, iterator last) test failed." << std::endl;  }  // 11. 测试size()  if (lst1.size() == 1) {  std::cout << "11. size() test passed." << std::endl;  } else {  std::cout << "11. size() test failed." << std::endl;  }  // 12. 测试empty()  lst1.pop\_back();  if (lst1.empty()) {  std::cout << "12. empty() test passed." << std::endl;  } else {  std::cout << "12. empty() test failed." << std::endl;  }  // 13. 测试begin()  lst1.push\_back(40);  if (\*lst1.begin() == 40) {  std::cout << "13. begin() test passed." << std::endl;  } else {  std::cout << "13. begin() test failed." << std::endl;  }  // 14. 测试end()  it = lst1.end();  --it;  if (\*it == 40) {  std::cout << "14. end() test passed." << std::endl;  } else {  std::cout << "14. end() test failed." << std::endl;  }  // 15. 测试operator=()  list<int> lst3;  lst3 = lst1;  if (lst3.size() == lst1.size() && \*lst3.begin() == \*lst1.begin()) {  std::cout << "15. operator=() test passed." << std::endl;  } else {  std::cout << "15. operator=() test failed." << std::endl;  }  // 16. 测试splice()  list<int> lst4;  lst4.push\_back(50);  lst4.push\_back(60);  it = lst1.begin();  lst1.splice(it, lst4);  if (lst1.size() == 3 && \*lst1.begin() == 50 && \*(++lst1.begin()) == 60 && \*(--lst1.end()) == 40 && lst4.empty()) {  std::cout << "16. splice() test passed." << std::endl;  } else {  std::cout << "16. splice() test failed." << std::endl;  }  // 17. 测试iterator的operator++()  it = lst1.begin(); // 指向50  ++it;  if (\*it == 60) {  std::cout << "17. iterator operator++() test passed." << std::endl;  } else {  std::cout << "17. iterator operator++() test failed." << std::endl;  }  // 18. 测试iterator的operator++(int)  it++;  if (\*it == 40) {  std::cout << "18. iterator operator++(int) test passed." << std::endl;  } else {  std::cout << "18. iterator operator++(int) test failed." << std::endl;  }  // 19. 测试iterator的operator--()  --it;  if (\*it == 60) {  std::cout << "19. iterator operator--() test passed." << std::endl;  } else {  std::cout << "19. iterator operator--() test failed." << std::endl;  }  // 20. 测试iterator的operator--(int)  it--;  if (\*it == 50) {  std::cout << "20. iterator operator--(int) test passed." << std::endl;  } else {  std::cout << "20. iterator operator--(int) test failed." << std::endl;  }  // 21. 测试iterator的operator\*()  if (\*it == 50) {  std::cout << "21. iterator operator\*() test passed." << std::endl;  } else {  std::cout << "21. iterator operator\*() test failed." << std::endl;  }  // 22. 测试iterator的operator==()  list<int>::iterator it\_test = lst1.begin();  if (it == it\_test) {  std::cout << "22. iterator operator==() test passed." << std::endl;  } else {  std::cout << "22. iterator operator==() test failed." << std::endl;  }  return 0;  } | | | | | |
| 1. **Result analysis and discussion**（Analysis of experimental results and summing up the harvest and the existing problems）（此部分应包含实验结果，对实验结果的分析，实验收获的总结，实验中存在问题的讨论等；另外，需要回应一下如下思考题：请调研一种或几种开源或商业版本的C++标准模板库STL的底层实现原理，尤其是list容器的实现原理，并对比本实验中你的实现，分析在技术上和功能上的异同点。）   实验结果：    分析：  此次实验的结果表明，自定义的 list 容器的22个功能全部通过了测试，具体分析如下：   1. 默认构造函数测试：list<int> lst1; 创建了一个空的链表，lst1.empty() 返回 true，验证了默认构造函数正确地初始化了空列表。 2. 析构函数测试：通过在局部作用域内创建并销毁 list 对象，且未出现内存泄漏，验证了析构函数正确释放了动态分配的内存。 3. 拷贝构造函数测试：list<int> lst2(lst1); 复制了 lst1，并通过比较大小和元素值，验证了拷贝构造函数的正确性。 4. push\_front 测试：调用 lst1.push\_front(5); 后，列表大小为2，首元素为5，证明了 push\_front 方法能够正确地在列表前端插入元素。 5. push\_back 测试：调用 lst1.push\_back(15); 后，列表大小为3，尾元素为15，验证了 push\_back 方法能够正确地在列表尾部插入元素。 6. insert 测试：在第二个位置插入8，确认了 insert 方法能够在指定位置正确插入元素，并返回准确的迭代器。 7. pop\_front 测试：调用 lst1.pop\_front(); 后，列表大小为3，首元素为8，验证了 pop\_front 方法能够正确地移除列表前端的元素。 8. pop\_back 测试：调用 lst1.pop\_back(); 后，列表大小为2，尾元素为10，证明了 pop\_back 方法能够正确地移除列表尾部的元素。 9. erase(iterator position) 测试：删除特定位置的元素后，列表大小为1，剩余元素为10，验证了 erase 方法能够正确地删除指定位置的元素。 10. erase(iterator first, iterator last) 测试：删除指定范围的元素后，列表大小为1，元素为30，证明了该方法能够正确地删除指定范围内的元素。 11. size() 测试：lst1.size() 返回1，验证了 size() 方法能够准确地返回列表的元素数量。 12. empty() 测试：清空列表后，lst1.empty() 返回 true，证明了 empty() 方法能够正确地判断列表是否为空。 13. begin() 测试：\*lst1.begin() 返回40，验证了 begin() 方法能够正确地返回指向列表首元素的迭代器。 14. end() 测试：\*(--lst1.end()) 返回40，证明了 end() 方法能够正确地返回指向列表末尾后位置的迭代器。 15. 赋值运算符测试：lst3 = lst1; 后，lst3 的大小和元素与 lst1 相同，验证了赋值运算符的正确性。 16. splice() 测试：将 lst4 拼接到 lst1 后，lst1 大小为3，元素顺序正确，且 lst4 为空，证明了 splice() 方法能够正确地拼接列表。 17. 迭代器的前置递增运算符测试：++it; 后，\*it 为60，验证了迭代器的前置递增运算符功能正常。 18. 迭代器的后置递增运算符测试：it++; 后，\*it 为40，证明了迭代器的后置递增运算符功能正确。 19. 迭代器的前置递减运算符测试：--it; 后，\*it 为60，验证了迭代器的前置递减运算符功能正常。 20. 迭代器的后置递减运算符测试：it--; 后，\*it 为50，证明了迭代器的后置递减运算符功能正确。 21. 解引用运算符测试：\*it 返回50，验证了迭代器的解引用运算符能够正确获取当前元素的值。 22. 迭代器的相等运算符测试：it == it\_test 返回 true，证明了迭代器的相等运算符能够正确判断两个迭代器是否指向同一位置。   综上所述，实验结果表明，自定义的 list 容器的所有功能均已正确实现，所有测试均通过，功能实现符合预期。  实验收获总结：  通过本次实验，我深入理解了如何在C++中实现一个泛型的双向链表容器，并使其具备与标准模板库（STL）中list容器相同的功能。在实现和测试的过程中，我获得了以下收获：   * 数据结构实现：掌握了双向链表的底层实现方式，包括节点的定义、前驱和后继指针的维护，以及头尾指针的初始化和使用。 * 模板编程：通过实现template<typename T>的泛型容器，巩固了对模板编程的理解，使容器具有通用性，可以存储任意类型的数据。 * 内存管理：熟练应用了new和delete操作符来动态分配和释放内存，深入理解了内存泄漏的风险及其预防方法，确保了程序的稳定性和可靠性。 * 迭代器设计：实现了自定义的迭代器类，支持指针操作和遍历容器元素，掌握了迭代器的重要性及其在容器中的作用。 * 操作符重载：通过对迭代器的++、--、\*、==等操作符的重载，增强了对操作符重载的理解和应用能力，提升了代码的可读性和简洁性。 * 拷贝控制：实现了拷贝构造函数和赋值运算符，解决了对象深拷贝的问题，避免了浅拷贝导致的潜在错误，强化了对C++拷贝控制的认识。 * 面向对象编程：在类的设计中，合理划分了公有成员和私有成员，封装了数据，提高了代码的模块化和可维护性。 * 全面测试：通过编写test.cpp，对容器的22个功能进行了逐一测试，培养了严谨的测试习惯，理解了测试在开发过程中的重要性。 * 调试技巧：在测试过程中，遇到了各种问题，通过调试找到了错误的原因并进行了修正，提高了解决实际问题的能力。 * 代码规范和注释：按照要求，在每个方法前添加了对应的序号和注释，养成了良好的编码习惯，保证了代码的可读性和规范性。 * 标准库学习：通过对比自实现的list容器和STL中list的功能，加深了对标准库容器的理解，为以后更好地使用标准库打下了基础。 * 项目实践：完整经历了一个项目的实现过程，包括需求分析、设计、编码、测试和结果分析，提升了整体的项目开发能力。   总的来说，本次实验不仅巩固了对链表这一基础数据结构的理解，而且提升了C++编程的综合能力，为今后的学习和开发奠定了坚实的基础。  实验中存在的问题：   1. 内存管理问题：   在实现自定义的list容器时，使用了new和delete来手动管理内存。如果在程序中未能正确匹配每个new对应一个delete，可能导致内存泄漏。需要确保在析构函数中正确释放所有动态分配的内存。   1. 迭代器失效问题：   在对链表进行插入、删除操作后，之前获取的迭代器可能会失效。如果继续使用这些迭代器，可能会访问非法内存，导致未定义行为。需要在修改操作后避免使用旧的迭代器，或重新获取迭代器。   1. 边界条件处理不足：   某些操作（如pop\_front、pop\_back、erase）在列表为空时调用，可能会导致程序运行错误。需要在这些方法中增加对列表是否为空的检查，确保在列表为空时操作的安全性。  思考题：  C++标准模板库（STL）中的list容器通常实现为一个双向链表。以下是STL中list容器的一些关键实现细节：   1. 节点结构：STL中的list使用一个双向链表节点结构，每个节点包含一个数据元素和两个指针，分别指向前一个节点和后一个节点。 2. 头尾哨兵节点：STL中的list通常使用头尾哨兵节点（sentinel nodes），即一个特殊的空节点，作为链表的头部和尾部。这些哨兵节点简化了插入和删除操作，因为不需要处理特殊情况（如空链表或单节点链表）。 3. 迭代器：STL中的list迭代器是一个双向迭代器，支持前向和后向遍历。迭代器内部通常包含一个指向链表节点的指针，通过操作该指针来实现迭代器的功能。 4. 内存管理：STL中的list使用分配器（allocator）来管理内存。分配器是一个模板类，负责分配和释放内存，提供了更灵活的内存管理方式。 5. 异常安全性：STL中的list在设计时考虑了异常安全性，确保在异常发生时不会导致内存泄漏或数据结构损坏。   对比本实验中的实现  技术上的异同点：   1. 节点结构：  * 相同点：本实验中的list和STL中的list都使用了双向链表节点结构，每个节点包含数据元素和前后指针。 * 不同点：本实验中的节点结构较为简单，没有使用头尾哨兵节点，而是直接使用头指针和尾指针。  1. 头尾哨兵节点：  * 相同点：两者都使用了头尾指针来管理链表的起始和结束。 * 不同点：STL中的list使用了哨兵节点，而本实验中的实现没有使用哨兵节点，这使得本实验中的实现在处理边界条件时稍显复杂。  1. 迭代器：  * 相同点：两者的迭代器都是双向迭代器，支持前向和后向遍历。 * 不同点：STL中的迭代器实现更加复杂，支持更多的操作符重载和异常安全性，而本实验中的迭代器实现相对简单。  1. 内存管理：  * 相同点：两者都使用了动态内存分配（new和delete）来管理节点的内存。 * 不同点：STL中的list使用了分配器来管理内存，提供了更灵活和高效的内存管理方式，而本实验中的实现直接使用new和delete，缺乏分配器的灵活性。  1. 异常安全性：  * 相同点：两者都需要处理异常情况，确保在异常发生时不会导致内存泄漏或数据结构损坏。 * 不同点：STL中的list在设计时更加注重异常安全性，提供了更完善的异常处理机制，而本实验中的实现对异常安全性的考虑较少。   功能上的异同点   1. 基本功能：  * 相同点：两者都实现了list容器的基本功能，如插入、删除、遍历等操作。 * 不同点：STL中的list提供了更多的高级功能和优化，如分配器支持、异常安全性、更多的操作符重载等，而本实验中的实现主要关注基本功能的实现。  1. 性能优化：  * 相同点：两者都通过双向链表结构实现了高效的插入和删除操作。 * 不同点：STL中的list在性能优化方面更加完善，如使用分配器提高内存分配效率，使用哨兵节点简化边界条件处理，而本实验中的实现相对简单，缺乏这些优化。 | | | | | |
| Comments & Evaluation | Content & Design (A-E) | | |  | |
| Procedure & Codes (A-E) | | |  | |
| Results (A-E) | | |  | |
| Analysis & Discussion (A-E) | | |  | |
| Score (A-E):  Feedback comments: | | | | |