

**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 | | C++ Template Containers and Iterators  C++模板容器和迭代器 | | | |
| Date | | Sep/Oct，2024 | Type | | ☑Confirmatory （验证确认型）  ☑Design（设计型）  🗆Comprehensive（综合型） |
| 1. **Objective & Requirements（实验目的）**    1. Understand the concept of containers; can use template to define generic containers   理解C++容器的概念，学会使用模板定义通用容器   * 1. Understand the difference between contiguous memory allocation and linked memory allocation; grasp the implementation of linked storage   理解链式和连续式两种容器底层的存储方式及其区别; 掌握链式存储的实现方式   * 1. Understand the use of iterator and grasp its implementation   理解迭代器的概念和用途，掌握其实现原理   * 1. Can use iterator to traverse a list to finish a certain task;   Understand the concept of generic algorithms  能够使用容器和迭代器编写泛型算法 | | | | | |
| 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） 2. Task 1 （任务1）   For this task, you are provided with a template container with linked storage. Based on the source codes, implement the addHead() method which adds a new element at the head of the linked list of the container. Then, based on the addHead() method, implement the inputEmployee() method for the Company class, which takes users’ inputs and stores the input Employee into the Company’s container. Test your implementation in the main function.  基于所给代码，为链式容器类实现addHead()，用以在容器内链表的头部添加元素；基于addHead(), 为公司类实现inputEmployee()方法，用以接收用户的输入并将其存入公司的容器中。在main函数中测试你的实现。  补充的代码如下：      测试结果如下：     1. Task 2（任务2）   Read and comprehend the source code provided to you on the implementation of the inner iterator class for the linked container. Especially, try to understand the functionality of the public methods of the iterator class. Then, based on your implementations in Task 1 and the public interfaces of the iterator, implement the findBestPaid() method for the Company class to get the best paid employee. Test your implementation in the main() funtion.  阅读并理解链式容器类的内部迭代器类，掌握其公共接口的功能及其实现原理。基于迭代器类和任务1，进一步为公司类编写findBestPaid()方法以实现查找收入最高的员工。在main函数中测试你的实现。  补充的代码如下：    测试结果如下：     1. Task 3（任务3）      1. Based on the source codes in Task 1 and Task 2, design a container with doubly linked list storage that has a structure similar to the one in the above picture. For this you need to design the node structure and its fields for doubly linked list, design the fields and method interfaces of the container class, and design the fields and methods of the iterator inner class for the container. Then implement the container and its associated iterator inner class.   基于任务1和任务2，设计一个底层存储为双向链表的容器及其相应的迭代器，该双向链表的架构应和上图中所示类似。为此，需要设计链式存储的节点结构、容器类的数据成员和方法接口、对应迭代器内部类的数据成员和方法接口。最终实现完整的双向链式容器及其迭代器。  DoublyLinkedList.h:  #ifndef DOUBLYLINKEDLIST\_H  #define DOUBLYLINKEDLIST\_H  template<class T>  class DoublyLinkedList  {  private:  struct Node  {  T data;  Node \*prev;  Node \*next;  };  Node \*head;  Node \*tail;  int size;  public:  DoublyLinkedList(); // constructor  ~DoublyLinkedList(); // destructor  int getSize() const; // get the number of elements  bool isEmpty() const; // check whether the container is empty  void addHead(const T& newData); // add a node at the head  void addTail(const T& newData); // add a node at the tail  class Iterator  {  friend class DoublyLinkedList<T>;  private:  Node \*curr;  Iterator(Node \*ptr); // constructor with Node parameter  public:  Iterator(); // default constructor  Iterator operator++(int); // post-increment  Iterator operator--(int); // post-decrement  T& operator\*() const;  bool operator==(const Iterator other) const;  bool operator!=(const Iterator other) const;  };  Iterator Begin() const;  Iterator End() const;  Iterator rBegin() const; // reverse begin  Iterator rEnd() const; // reverse end  };  #include "DoublyLinkedList.cpp"  #endif  #include "DoublyLinkedList.h"  template<class T>  DoublyLinkedList<T>::DoublyLinkedList()  {  head = nullptr;  tail = nullptr;  size = 0;  }  template<class T>  DoublyLinkedList<T>::~DoublyLinkedList()  {  Node \*current = head;  Node \*temp;  while (current != nullptr)  {  temp = current;  current = current->next;  delete temp;  }  }  template<class T>  int DoublyLinkedList<T>::getSize() const  {  return size;  }  template<class T>  bool DoublyLinkedList<T>::isEmpty() const  {  return size == 0;  }  DoublyLinkedList.cpp  template<class T>  void DoublyLinkedList<T>::addHead(const T& newData)  {  Node \*newNode = new Node;  newNode->data = newData;  newNode->prev = nullptr;  newNode->next = head;  if (head != nullptr)  {  head->prev = newNode;  }  head = newNode;  if (tail == nullptr)  {  tail = newNode;  }  size++;  }  template<class T>  void DoublyLinkedList<T>::addTail(const T& newData)  {  Node \*newNode = new Node;  newNode->data = newData;  newNode->next = nullptr;  newNode->prev = tail;  if (tail != nullptr)  {  tail->next = newNode;  }  tail = newNode;  if (head == nullptr)  {  head = newNode;  }  size++;  }  template<class T>  DoublyLinkedList<T>::Iterator::Iterator()  {  curr = nullptr;  }  template<class T>  DoublyLinkedList<T>::Iterator::Iterator(Node \*ptr)  {  curr = ptr;  }  template<class T>  typename DoublyLinkedList<T>::Iterator DoublyLinkedList<T>::Iterator::operator++(int)  {  Iterator temp = \*this;  if (curr != nullptr)  {  curr = curr->next;  }  return temp;  }  template<class T>  typename DoublyLinkedList<T>::Iterator DoublyLinkedList<T>::Iterator::operator--(int)  {  Iterator temp = \*this;  if (curr != nullptr)  {  curr = curr->prev;  }  return temp;  }  template<class T>  T& DoublyLinkedList<T>::Iterator::operator\*() const  {  return curr->data;  }  template<class T>  bool DoublyLinkedList<T>::Iterator::operator==(const Iterator other) const  {  return curr == other.curr;  }  template<class T>  bool DoublyLinkedList<T>::Iterator::operator!=(const Iterator other) const  {  return curr != other.curr;  }  template<class T>  typename DoublyLinkedList<T>::Iterator DoublyLinkedList<T>::Begin() const  {  return Iterator(head);  }  template<class T>  typename DoublyLinkedList<T>::Iterator DoublyLinkedList<T>::End() const  {  return Iterator();  }  template<class T>  typename DoublyLinkedList<T>::Iterator DoublyLinkedList<T>::rBegin() const  {  return Iterator(tail);  }  template<class T>  typename DoublyLinkedList<T>::Iterator DoublyLinkedList<T>::rEnd() const  {  return Iterator();  }   1. Based on your implementation of the container with doubly linked storage, design and implement the method findBestPaidReverse() method for the Company class, which traverses the container from the tail to the head to find the employee with the highest salary. The findBestPaidReverse() method should be implemented using your designed and implemented iterators of the container. This means that your iterator should support the operator--. Besides, this also means that you should define two more special iterators, i.e. the reverse versions of Begin() and End(), to support the reverse traversal.   基于所实现的双向链式容器和迭代器，改写公司类，设计实现findBestPaidReverse()方法，以从尾至头的方式查找收入最高的员工。这意味着你实现的迭代器应支持--运算（重载operator --）；此外，这还意味着你应该实现Begin()和End()这两个特殊迭代器的反向版本，以配合operator--支持反向遍历。  Company.h:  #ifndef COMPCONT\_H  #define COMPCONT\_H  #include "employee.h"  #include "DoublyLinkedList.h"  class Company  {  private:  Employee bestPaid;  DoublyLinkedList<Employee> empContainer;  public:  void inputEmployee(); // input employee from keyboard and store in the container  void findBestPaid(); // find the best paid employee  void findBestPaidReverse(); // find the best paid employee from tail to head  void printBestPaid() const;  };  #endif  Company.cpp:  #include <iostream>  #include "company.h"  using namespace std;  void Company::inputEmployee()  {  while (true) {  Employee emp;  emp.input();  if (emp.getName() == "\*") {  break;  }  empContainer.addHead(emp);  }  }  void Company::findBestPaid()  {  if (empContainer.isEmpty()) {  cout << "No employees in the container." << endl;  return;  }  DoublyLinkedList<Employee>::Iterator it = empContainer.Begin();  bestPaid = \*it;  auto iteratorNotEqual = [](const DoublyLinkedList<Employee>::Iterator& it1, const DoublyLinkedList<Employee>::Iterator& it2) {  return !(it1 == it2);  };  for (it = empContainer.Begin(); iteratorNotEqual(it, empContainer.End()); it++) {  if (\*it > bestPaid) {  bestPaid = \*it;  }  }  }  void Company::findBestPaidReverse()  {  if (empContainer.isEmpty()) {  cout << "No employees in the container." << endl;  return;  }  DoublyLinkedList<Employee>::Iterator it = empContainer.rBegin();  bestPaid = \*it;  auto iteratorNotEqual = [](const DoublyLinkedList<Employee>::Iterator& it1, const DoublyLinkedList<Employee>::Iterator& it2) {  return !(it1 == it2);  };  for (it = empContainer.rBegin(); iteratorNotEqual(it, empContainer.rEnd()); it--) {  if (\*it > bestPaid) {  bestPaid = \*it;  }  }  }  void Company::printBestPaid() const  {  cout << "The best-paid employee and gross pay: " << endl;  bestPaid.output();  }   1. Test your implementation in the main function.   在主函数中测试你的实现。  main.cpp:  #include <iostream>  #include "Company.h"  using namespace std;  int main()  {  Company cmp;  cmp.inputEmployee();  cmp.findBestPaid();  cout << "Search for the highest earning employee from start to finish:" << endl;  cmp.printBestPaid();  cmp.findBestPaidReverse();  cout << "Search for the highest earning employee from finish to start:" << endl;  cmp.printBestPaid();  return 0;  }  运行结果如下： | | | | | |
| 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：   a). 在本实验Task2所实现的链式容器中，为什么将容器类定义为其内部迭代器类的友元类？  b). 双向链表是否需要头指针Head，还需要一个尾指针Tail，才能实现双向遍历？如果只有头指针，如何逆序遍历整个链表？   * 1. **在本实验所实现的链式容器中，将容器类定义为其内部迭代器类的友元类是为了允许迭代器类访问容器类的私有成员。也就是说，迭代器类需要访问容器类的私有数据结构（如链表的节点）以便正确地遍历容器中的元素。**   2. **双向链表通常使用头指针（Head）和尾指针（Tail）来实现双向遍历。头指针指向链表的第一个节点，尾指针指向链表的最后一个节点。这样可以方便地从任意一端开始遍历链表。**   3. **如果只有头指针，要逆序遍历整个链表，可以先遍历到链表的末尾，然后从末尾开始逆序遍历。具体步骤如下：**   **1.从头指针开始，遍历链表直到找到最后一个节点。**  **2.从最后一个节点开始，使用前驱指针（prev）逆序遍历链表。** | | | | | |
| Comments & Evaluation | Content & Design (A-E) | | |  | |
| Procedure & Codes (A-E) | | |  | |
| Results (A-E) | | |  | |
| Analysis & Discussion (A-E) | | |  | |
| Score (A-E):  Feedback comments: | | | | |