

**Lab Report**

**实验报告**

|  |  |
| --- | --- |
| **Course**: | Class Libraries and Data Structures |
| **Semester**: | 1st semester of the academic year **2023-2024** |
| **Major**: | Software Engineering |
| **Class**: | 2022 |
| **Student Name**: | 朱昊 |
| **Student ID:** | 222022321062008 |
| **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，2023 | 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/>  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/>）给出一个通用链式容器较为完整地实现。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/>  例如对splice功能的测试可参考：    =============================================================================  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.  list() {  head = new Node; // 创建头节点  head->data = 0; // 初始化头节点数据  head->next = head; // 头节点的下一个节点指向自身  head->prev = head; // 头节点的前一个节点指向自身  this->size = 0; // 初始化链表大小为 0  }  2.  //Postcondition: this list has been destroyed  ~list ();  ~list() {  // 释放链表中所有节点的内存  for (int i = this->size; i > 0; i--) {  this->pop\_front();// 删除链表头部元素  }  delete head; // 释放头节点内存  }  3.  //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  list(const list<T>& n) {  // 使用默认构造函数创建头节点  head = new Node;  head->data = 0;  head->next = head;  head->prev = head;  this->size = 0;  // 使用迭代器遍历原链表，逆序将元素添加到当前链表中  Iterator a = n.end();  a--;  for (; !(a == n.end()); a--) {  this->push\_front(\*a);// 在当前链表头部插入元素  }  }  4.  //Postcondition: x has been inserted at the front of this list.  void push\_front (const T& x);  void push\_front(const T& x) {  Node\* temp = new Node;  temp->data = x;  // 将新节点插入到头节点之后  temp->next = head->next;  temp->prev = head;  head->next->prev = temp;  head->next = temp;  this->size++;// 增加链表大小}  5.  //Postcondition: x has been inserted at the back of this list.  void push\_back(const T& x);  void push\_back(const T& x) {  Node\* temp = new Node;  temp->data = x;// 设置新节点的数据  // 将新节点插入到尾部  temp->next = head;  Iterator a = this->end();  a--;//迭代器a指向有元素的最后一个节点  temp->prev = a.curr;  (a.curr)->next = temp;  head->prev = temp;  this->size++;// 增加链表大小  }  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.  Iterator insert(Iterator position, const T& x) {  Node\* temp = new Node;  temp->data = x;  // 将新节点插入到尾部  temp->next = (position.curr);  temp->prev = ((position.curr)->prev);  ((position.curr)->prev)->next = temp;  (position.curr)->prev = temp;  this->size++;// 增加链表大小  return Iterator(temp);  }  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();  void pop\_front() {  if (this->empty()) {  cout << "错误指令，当前链表无元素" << endl;  }  else {  Node\* temp = head->next;// 指向要删除的节点  head->next = temp->next;// 调整节点指针  (temp->next)->prev = head;  delete temp; // 释放节点内存  size--; // 减小链表大小  }  }  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();  void pop\_back() {  if (this->empty()) {  cout << "错误指令，当前链表无元素" << endl;  }  else {  Node\* temp = head->prev; // 指向要删除的节点  head->prev = temp->prev; // 调整节点指针  (temp->prev)->next = head;  delete temp; // 释放节点内存  size--; // 减小链表大小  }  }  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.  void erase (iterator position);  Note: The worstTime(n) is constant.  void erase(Iterator position) {  ((position.curr)->prev)->next = (position.curr)->next; // 调整节点指针  ((position.curr)->next)->prev = (position.curr)->prev;  delete position.curr; // 释放节点内存  size--; // 减小链表大小  }  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.  void erase (iterator first, iterator last);  Note: The time for this method is proportional to the number of items removed.  void erase(Iterator first, Iterator last) {  while (!(first == last)) {  Iterator temp(first.curr);// 存储当前迭代器  ++first; // 先递增迭代器，避免悬挂指针问题  this->erase(temp);  }  }  11.  //Postcondition: the number of items in this list has been returned.  unsigned size() const;  unsigned getsize() const {  return this->size;  }  12.  //Postcondition: true has been returned if this list is empty; Otherwise, false has been returned.  bool empty() const;  bool empty() const {  return size == 0;  }  13.  //Postcondition: an iterator positioned at the front of this list has been returned.  iterator begin();  Iterator begin() const{  return Iterator(head->next);  }  14.  //Postcondition: an iterator positioned **AFTER** the last item in this list has been returned.  iterator end();  Iterator end() const{  return Iterator(head);  }  15.  //Postcondition: this list contains a copy of x,  //and a reference to this list has been returned  list<T>& operator=(const list<T>& x)  list<T>& operator=(const list<T>& x) {  Iterator temp = x.end();  temp--;  Iterator a = this->begin();  Iterator b = this->end();  this->erase(a, b); //删除当前链表中的所有元素  for (; !( temp == x.end() ); temp--) {  this->push\_front(\*temp);// 在当前链表头部插入元素  }  return \*this;  }  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.  // 将另一个链表的所有元素插入到指定位置之前  void splice(Iterator position, list<T>& x) {  Iterator a = x.end();  a--;  //通过循环将x里的每个元素提取插入  for (; !(a == x.end()); a--) {  position = this->insert(position, \*a);// 在指定位置插入元素  }  }  ===============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”.  Iterator& operator++() {  this->curr = curr->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)  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”.  Iterator operator++(int) {  Iterator temp;  temp.curr = this->curr;  this->curr = curr->next;  return temp;  }  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  Iterator& operator--() {  this->curr = curr->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); //post-decrement  Iterator operator--(int) {  Iterator temp;  temp.curr = this->curr;  this->curr = curr->prev;  return temp;  }  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”.  T& operator\*() {  return curr->data;  }  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);  bool operator==(const Iterator& x) {  return this->curr == x.curr;  } | | | | | |
| 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：1. 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。）   以下是main.cpp的测试代码：  #include <iostream>  #include "list.cpp"  using namespace std;  template <typename T>  void print(list<T> &a) {  list<int>::Iterator q = a.begin();  list<int>::Iterator n = a.end();  for (; !(q == a.end()); q++) {  cout << \*q << endl;  }  cout << "------------" << endl;  }  int main() {  list <int> a;  a.push\_front(1);  a.push\_front(2);  a.push\_front(3);  a.push\_front(4);  a.push\_front(5);  print(a);  a.pop\_front();  a.pop\_front();  print(a);  a.push\_back(5);  a.push\_back(4);  print(a);  list<int>::Iterator q = a.begin();  list<int>::Iterator p = a.begin();  list<int>::Iterator n = a.end();  q++; q++; q++;  a.insert(q, 10);  print(a);  a.erase(q);  print(a);  p++; p++;  a.erase(p, n);  print(a);  //list<int> b;  list<int> b(a);  print(b);  a.splice(n, b);  print(a);  b = a;  print(b);    return 0;  }      该链表的结构如下：    这与实验的要求有一点不同的地方，就是没有用到tail指向尾。因为我是参考书上list容器实现的一种方式实现的。每个链表初始化时，头指针都指向头节点，并且当没有元素赋值时，头节点的prev和next也是指向头节点，所以说，这样的结构更倾向于一个环，而不是一条链，所以没有用到tail。  实验中存在的问题（难点）：当链表的某个节点被erase后，原来指向那个节点的迭代器还是存在，但是指向的地址应该时非法且无法访问的了。我在main里测试时，erase后还是用了那个迭代器，这样就会出bug报错，让我找了好久才发现这个问题。  思考题：  异同：简而言之，商用的list容器在初始化list对象时，就会分配一大块内存—--通常是1kb。这块内存称作缓冲区，缓冲区里都是前后无指向，没有存数据但创建好了的空节点。而push\_front、insert和push\_back都是直接使用缓冲区里的空节点。当缓冲区满了以后，会自动分配一个相同大小的新缓冲区，并且有专门的节点类型保存和指向这些缓冲区。而商用的list类里也有专门的字段free\_list来指向最近被删除节点的指针，从而记住所有被删除的节点。也就是说删除并不是直接释放，相反还可以保存下来，并循环利用。  并且商用list里，无论是迭代器的功能，还是list里的功能接口都更多。如reverse可以把把链表的顺序颠倒，merge可以合并已排序的链表，sort可以把链表中的元素进行排序等等。 | | | | | |
| Comments & Evaluation | Content & Design (A-E) | | |  | |
| Procedure & Codes (A-E) | | |  | |
| Results (A-E) | | |  | |
| Analysis & Discussion (A-E) | | |  | |
| Score (A-E):  Feedback comments: | | | | |