

**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**: | 吴孜远 |
| **Student ID:** | 222022321062009 |
| **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**（隐式调用）, for example,  list<Employee> employees;  makes employees an empty list, whose items will be of type Employee.  **函数实现：**  list<T>::list()  {  head = NULL;  tail = NULL;  }  **测试用例：**  int main()  {  list<int> first;  first.push\_back(4);  first.push\_back(3);  first.push\_back(2);  first.push\_back(1);  std::cout << "The contents of fifth are: ";  for (list<int>::iterator it = first.begin(); !(it == first.end()); it++) {  std::cout << \*it << ' ';  }  std::cout << '\n';  return 0;  }  **输出结果：**    2.  //Postcondition: this list has been destroyed  ~list ();  **函数实现：**  list<T>::~list()  {  Node\* current = head;  Node\* temp = NULL;  while (current != NULL)  {  temp = current;  current = current->next;  delete temp; //release  }  }  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**  **函数实现：**  list<T>::list(const list<T>& x)  {  Node\* current = x.head;  while (current != NULL)  {  push\_back(current->item);  current = current->next;  }  }**测试用例：**  int main()  {  list<int> first;  first.push\_back(4);  first.push\_back(3);  first.push\_back(2);  first.push\_back(1);  list<int> second(first);  std::cout << "The contents of fifth are: ";  for (list<int>::iterator it = second.begin(); !(it == second.end()); it++) {  std::cout << \*it << ' ';  }  std::cout << '\n';  return 0;  }  **输出结果：**    4.  //Postcondition: x has been inserted at the front of this list.  void push\_front (const T& x);  **函数实现：**  void list<T>::push\_front(const T& x)  {  Node\* newNode = new Node; //创建新节点  newNode->item = x;  if (empty()) {  head = newNode;  tail = newNode;  newNode->pre = NULL;  newNode->next = NULL;  }  else {  head->pre = newNode;  newNode->pre = NULL;  newNode->next = head;  head = newNode;  }  }  **测试用例：合并至push\_back函数**  5.  //Postcondition: x has been inserted at the back of this list.  void push\_back(const T& x);  **函数实现：**  void list<T>::push\_back(const T& x)  {  Node\* newNode = new Node;  newNode->item = x;  if (empty()) {  head = newNode;  tail = newNode;  newNode->pre = NULL;  newNode->next = NULL;  }  else {  newNode->pre = tail; //新节点pre指向原来的tail  tail->next = newNode; //原来的尾节点的next指向新节点  newNode->next = NULL;  tail = newNode;  }  }  **测试用例：**  int main()  {  list<int> first;  first.push\_front(4);  first.push\_back(3);  first.push\_front(2);  first.push\_back(1);  std::cout << "The contents of first are: ";  for (list<int>::iterator it = first.begin(); !(it == first.end()); it++) {  std::cout << \*it << ' ';  }  std::cout << '\n';  return 0;  }  **输出结果：**    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.  **函数实现：**  typename list<T>::iterator list<T>::insert(iterator position, const T& x) {  Node\* newNode = new Node;  newNode->item = x;  newNode->next = position.curr;  newNode->pre = (position.curr)->pre;  (position.curr)->pre->next = newNode;  position.curr->pre = newNode;  position--;  return position;  }  **测试用例：**  int main()  {  list<int> mylist;  list<int>::iterator it;  // set some initial values:  for (int i = 1; i <= 5; ++i) mylist.push\_back(i); // 1 2 3 4 5  it = mylist.begin();  ++it; // it points now to number 2  mylist.insert(it, 10); // 1 10 2 3 4 5  for (it = mylist.begin(); !(it == mylist.end()); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  }  **输出结果：**  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();  **函数实现：**  void list<T>::pop\_front() {  head = head->next; //head指向第二个元素  delete head->pre;  head->pre = NULL;  }  **测试用例：**  int main()  {  list<int> mylist;  for (int i = 1; i <= 5; ++i) mylist.push\_back(i); // 1 2 3 4 5  while (!mylist.empty())  {  mylist.pop\_front();  std::cout << "The contents are ";  for (list<int>::iterator it = mylist.begin(); !(it == mylist.end()); it++) {  std::cout << \*it << ' ';  }  std::cout << "\nFinal size of mylist is " << mylist.size() << '\n';  }  return 0;  }  **输出结果：**    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();  **函数实现：**  void list<T>::pop\_back() {  tail = tail->pre; //tail指向倒数第二个元素  delete tail->next;  tail->next = NULL;  }  **测试用例：**  int main()  {  list<int> mylist;  for (int i = 1; i <= 5; ++i) mylist.push\_back(i); // 1 2 3 4 5  while (!mylist.empty())  {  mylist.pop\_back();  std::cout << "The contents are ";  for (list<int>::iterator it = mylist.begin(); !(it == mylist.end()); it++) {  std::cout << \*it << ' ';  }  std::cout << "\nFinal size of mylist is " << mylist.size() << '\n';  }  return 0;  }  **输出结果：**    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.  **函数实现：**  typename list<T>::iterator list<T>::erase(iterator position) {  if ((position.curr) == NULL) {  return position;  }  //定位前后两个结点的位置  Node\* front = (position.curr)->pre;  Node\* behind = (position.curr)->next;  position++;  delete (position.curr)->pre; //这里的括号必须要加  front->next = behind;  behind->pre = front;  return position;  }  **测试用例：同erase (iterator first, iterator last)函数一起测试**  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.  **函数实现：**  typename list<T>::iterator list<T>::erase(iterator first, iterator last){  if (first.curr == last.curr == NULL) {  return last;  }  //定位前后两个结点的位置  Node\* front = (first.curr)->pre;  Node\* behind = (last.curr);  while (!(first == last)) {  first++;  delete (first.curr)->pre;  }  front->next = behind;  behind->pre = front;  return last;  }  **测试用例：**  int main()  {  list<int> mylist;  list<int>::iterator it1, it2;  // set some values:  for (int i = 1; i < 10; ++i) mylist.push\_back(i \* 10);  // 10 20 30 40 50 60 70 80 90  it1 = it2 = mylist.begin(); // ^^  for (int i = 0; i < 6; i++)  it2++; // ^ ^  ++it1; // ^ ^  it1 = mylist.erase(it1); // 10 30 40 50 60 70 80 90  // ^ ^  it2 = mylist.erase(it2); // 10 30 40 50 60 80 90  // ^ ^  ++it1; // ^ ^  --it2; // ^ ^  mylist.erase(it1, it2); // 10 30 60 80 90  // ^  std::cout << "mylist contains:";  for (it1 = mylist.begin(); !(it1 == mylist.end()); ++it1)  std::cout << ' ' << \*it1;  std::cout << '\n';  return 0;  }  **输出结果：**    11.  //Postcondition: the number of items in this list has been returned.  unsigned size() const;  **函数实现：**  unsigned list<T>::size() const  {  Node\* temp = head;  int size = 0;  while (temp != NULL) {  temp = temp->next;  size++;  }  return size;  }  **测试用例：**  int main()  {  list<int> mylist;  for (int i = 1; i <= 5; ++i) mylist.push\_back(i); // 1 2 3 4 5  while (!mylist.empty())  {  mylist.pop\_front();  std::cout << "The contents are ";  for (list<int>::iterator it = mylist.begin(); !(it == mylist.end()); it++) {  std::cout << \*it << ' ';  }  std::cout << "\nFinal size of mylist is " << mylist.size() << '\n';  }  return 0;  }  **输出结果：**    12.  //Postcondition: true has been returned if this list is empty; Otherwise, false has been returned.  bool empty() const;  **函数实现：**  bool list<T>::empty() const  {  return size() == 0;  **测试用例：**  int main()  {  list<int> mylist;  std::cout << "isempty? " << mylist.empty() << '\n';  for (int i = 1; i <= 10; ++i) mylist.push\_back(i);  std::cout << "isempty? " << mylist.empty() << '\n';  return 0;  }  **输出结果：**    13.  //Postcondition: an iterator positioned at the front of this list has been returned.  iterator begin();  **函数实现：**  typename list<T>::iterator list<T>::begin() const  {  return iterator(head);  }  **测试用例：进行上面的函数测试时已经顺带测试了**  **输出结果：**  14.  //Postcondition: an iterator positioned **AFTER** the last item in this list has been returned.  iterator end();  **函数实现：**  typename list<T>::iterator list<T>::end() const  {  return iterator();  }  **测试用例：进行上面的函数测试时已经顺带测试了**  **输出结果：**  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)  **函数实现：**  list<T>& list<T>::operator=(const list<T>& x)  {  //先将原来存储的元素全部删掉  Node\* current = head;  Node\* temp = NULL;  while (current != NULL)  {  temp = current;  current = current->next;  delete temp; //release  }  //将x中的所有元素都复制过来  Node\* current2 = x.head;  while (current2 != NULL)  {  push\_back(current2->item);  current2 = current2->next;  }  return \*this;  }  **测试用例：**  int main()  {  list<int> first;  list<int> second;  for (int i = 1; i <= 10; ++i) first.push\_back(i);  second = first;  std::cout << "The contents of first are: ";  for (list<int>::iterator it = first.begin(); !(it == first.end()); it++) {  std::cout << \*it << ' ';  }    std::cout << "\n";  std::cout << "The contents of second are: ";  for (list<int>::iterator it = second.begin(); !(it == second.end()); it++) {  std::cout << \*it << ' ';  }  return 0;  }  **输出结果：**    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.  **函数实现：**  void list<T>::splice(iterator position, list<T>& x) {  Node\* front = position.curr->pre;  Node\* behind = position.curr;  front->next = x.head;  x.head->pre = front;  behind->pre = x.tail;  x.tail->next = behind;  x.head = NULL;  x.tail = NULL;  }  **测试用例：**  int main()  {  list<int> mylist1, mylist2;  list<int>::iterator it;  // set some initial values:  for (int i = 1; i <= 4; ++i)  mylist1.push\_back(i); // mylist1: 1 2 3 4  for (int i = 1; i <= 3; ++i)  mylist2.push\_back(i \* 10); // mylist2: 10 20 30  it = mylist1.begin();  ++it; // points to 2  mylist1.splice(it, mylist2); // mylist1: 1 10 20 30 2 3 4  // mylist2 (empty)  // "it" still points to 2 (the 5th element)  std::cout << "mylist1 contains:";  for (it = mylist1.begin(); !(it == mylist1.end()); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  std::cout << "mylist2 contains:";  for (it = mylist2.begin(); !(it == mylist2.end()); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  }  **输出结果：**    ===============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”.  **函数实现：**  typename list<T>::iterator& list<T>::iterator::operator++()  {  this->curr = curr->next;  return \*this; //return iterator object  }  **测试用例：**  int main()  {  list<int> mylist1, mylist2;  list<int>::iterator it;  // set some initial values:  for (int i = 1; i <= 4; ++i)  mylist1.push\_back(i); // mylist1: 1 2 3 4  it = mylist1.begin();  std::cout << ' ' << \*(it);  std::cout << ' ' << \*(++it);  std::cout << ' ' << \*(it);  return 0;  }  **输出结果：**    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”.  **函数实现：**  typename list<T>::iterator list<T>::iterator::operator++(int)  {  iterator temp = \*this; //default copy constructor  this->curr = curr->next;  return temp; //return iterator object  }  **测试用例：**  int main()  {  list<int> mylist1, mylist2;  list<int>::iterator it;  // set some initial values:  for (int i = 1; i <= 4; ++i)  mylist1.push\_back(i); // mylist1: 1 2 3 4  it = mylist1.begin();  std::cout << ' ' << \*(it);  std::cout << ' ' << \*(it++);  std::cout << ' ' << \*(it);  return 0;  }  **输出结果：**    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  **函数实现：**  typename list<T>::iterator& list<T>::iterator::operator--()  {  this->curr = curr->pre;  return \*this; //return iterator object  }  **测试用例：**  int main()  {  list<int> mylist1, mylist2;  list<int>::iterator it;  // set some initial values:  for (int i = 1; i <= 4; ++i)  mylist1.push\_back(i); // mylist1: 1 2 3 4  it = mylist1.begin();  it++;  std::cout << ' ' << \*(it);  std::cout << ' ' << \*(--it);  std::cout << ' ' << \*(it);  return 0;  }  **输出结果：**    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  **函数实现：**  typename list<T>::iterator list<T>::iterator::operator--(int)  {  iterator temp = \*this; //default copy constructor  this->curr = curr->pre;  return temp; //return iterator object  }  **测试用例：**  int main()  {  list<int> mylist1, mylist2;  list<int>::iterator it;  // set some initial values:  for (int i = 1; i <= 4; ++i)  mylist1.push\_back(i); // mylist1: 1 2 3 4  it = mylist1.begin();  it++;  std::cout << ' ' << \*(it);  std::cout << ' ' << \*(it--);  std::cout << ' ' << \*(it);  return 0;  }  **输出结果：**    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”.  **函数实现：**  T& list<T>::iterator::operator\*() const  {  return curr->item; //return data reference  }  **测试用例：**  int main()  {  list<int> mylist1, mylist2;  list<int>::iterator it;  // set some initial values:  for (int i = 1; i <= 4; ++i)  mylist1.push\_back(i); // mylist1: 1 2 3 4  it = mylist1.begin();  std::cout << ' ' << \*(it);  \*(it) = 6;  std::cout << ' ' << \*(it);  return 0;  }  **输出结果：**    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);  **函数实现：**  bool list<T>::iterator::operator==(const iterator other) const  {  return curr == other.curr;  }  **测试用例：**  int main()  {  list<int> first;  for (int i = 1; i <= 10; ++i) first.push\_back(i);  std::cout << "The contents of first are: ";  for (list<int>::iterator it = first.begin(); !(it == first.end()); it++) {  std::cout << \*it << ' ';  }  return 0;  }  **输出结果：** | | | | | |
| 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。）   总结：  1. 运算符的优先级记住，否则写连续运算的时候就会纠结。  C语言运算符的优先级  2. 因为是双向链表，所以在增添或删除结点时光靠想可能不是很条理，这时候可以画图，反而效率会提高。  3. 对于一些方法，如果链表为空或者传入的参数为空，会导致不一样的执行方式或返回结果，所以要考虑是否应该将两种情况分开考虑。  调研对比：  调研了惠普的实现，其对于效率极其关注，所以比我的写法要复杂很多，但效率确实高。   1. 惠普将链表头尾相连，并在第一个结点的前面（最后一个节点的后面）定义了一个头节点，不储存数据，这样方便了在第一个结点之前插入结点等类似操作。 2. 惠普的begin和end方法直接返回的node指针，因为它的iterator类中有protect的构造方法iterator(list\_node\*x):node(x){}，可以自动执行强制类型转换 3. 惠普的实现利用了每一个node占用空间一样这个特点，每一次使用new都会产生一次中断，效率低，所以惠普令list类开发它自己的内存管理例程，也就是分配回收结点。在链表第一次插入时分配了一块缓冲区（一般是1kb），也就是好多个空结点，用于连续插入，又定义了一个指针用于指向下一个将要使用的结点吗，当这个指针指向空时就再分配一块缓存，极大的节省了时间。为了管理这些缓冲区，还设计了一个buffer\_list用于不同链表间的跳转。删除也是一样，将free的结点保存到一个free\_list中，不删除，用的时候先检测这个空闲链表，方便省时间。 4. 我的链表甚至没有length变量，惠普还是保留了，因为效率高吧。 | | | | | |
| Comments & Evaluation | Content & Design (A-E) | | |  | |
| Procedure & Codes (A-E) | | |  | |
| Results (A-E) | | |  | |
| Analysis & Discussion (A-E) | | |  | |
| Score (A-E):  Feedback comments: | | | | |