计算机科学技术学院实验报告

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **课程名称** | 面向对象程序设计 | | | **学 号** |  |
| **实验项目** | 实验六 | | | **姓 名** |  |
| **学 时** | 2 | **项目性质** | 综合型 | **班 级** | 2405111B |
| **指导教师** | 蒋振刚 | **实验地点** | 实训楼424 | **日 期** | 2025年4月27日 |
| 1. **实验目的和要求**   第6个实验：虚函数。  目的：练习虚函数的使用方法，体会虚函数的作用。  要求：以DList为基类，派生队列类：DQueue。结合上次实验的DStack，完成：  1）在DList中增加与插入位置无关的纯虚函数Insert()和Delete()；  1）由用户指定一种数据结构类型；  2）根据用户指定的数据结构类型动态创建对应的数据结构；  3）分别由Insert和Delete对应入栈和出栈或者入队和出队操作。  4）由测试代码验证以上操作。 | | | | | |
| 1. **实验环境**   系统：fedora 42  环境：g++ | | | | | |
| 1. **实验内容与过程**   **DQueue.h:**  **#ifndef DQeueue\_H**  **#define DQeueue\_H**  **#include "/home/yohane-mashiro/cpp\_homework/work2/include/Dlist.h"**  **//如需使用自行修改路径**  **class DQueue:public DList{**  **public:**  **DQueue(); // 构造函数**  **using DList::DList;**  **~DQueue() {}**  **// 获取队头元素**  **bool front(int &e) const;**  **// 判断队列是否为空**  **bool isEmpty() const;**  **// 获取队列的大小**  **int size() const;**    **// 实现基类的纯虚函数**  **bool Insert(int e) override; // 对应入队操作**  **bool Delete(int &e) override; // 对应出队操作**  **};**  **DQueue::DQueue() : DList() {}**  **bool DQueue::front(int &e) const {**  **if (isEmpty()) {**  **std::cout << "队列为空，无法获取队头元素！" << std::endl;**  **return false;**  **}**  **return getElem(1, e); // 获取头部元素**  **}**  **bool DQueue::isEmpty() const {**  **return DList::isEmpty(); // 调用基类的isEmpty方法**  **}**  **int DQueue::size() const {**  **return DList::size(); // 调用基类的size方法**  **}**  **bool DQueue::Insert(int e) {**  **return insert(size() + 1, e);**  **}**  **bool DQueue::Delete(int &e) {**  **if (isEmpty()) {**  **std::cout << "队列为空，无法出队！" << std::endl;**  **return false;**  **}**  **return remove(1, e);**  **}**  **#endif**  **Dlist:**  **#ifndef DLIST\_H**  **#define DLIST\_H**  **#include <iostream>**  **class DList {**  **public:**  **// 构造函数**  **DList();**    **// 析构函数**  **~DList();**    **// 清空线性表**  **void clear();**    **// 判断线性表是否为空**  **bool isEmpty() const;**    **// 获取线性表长度**  **int size() const;**    **// 获取指定位置的元素**  **bool getElem(int i, int &e) const;**    **// 查找元素位置（返回第一次出现的位置，未找到返回0）**  **int locate(int e) const;**    **// 在指定位置插入元素**  **bool insert(int i, int e);**    **// 删除指定位置的元素**  **bool remove(int i, int &e);**    **// 打印线性表**  **void display() const;**  **protected:**  **struct Node {**  **int data;**  **Node \*prev;**  **Node \*next;**  **};**  **Node \*head;**  **Node \*tail;**  **int length;**  **};**  **DList::DList() : length(0) {**  **head = new Node;**  **tail = new Node;**  **head->next = tail;**  **tail->prev = head;**  **}**  **DList::~DList() {**  **clear();**  **delete head;**  **delete tail;**  **}**  **void DList::clear() {**  **Node \*p = head->next;**  **while (p != tail) {**  **Node \*q = p;**  **p = p->next;**  **delete q;**  **}**  **head->next = tail;**  **tail->prev = head;**  **length = 0;**  **}**  **bool DList::isEmpty() const {**  **return length == 0;**  **}**  **int DList::size() const {**  **return length;**  **}**  **bool DList::getElem(int i, int &e) const {**  **if (i < 1 || i > length) {**  **std::cout << "位置无效！" << std::endl;**  **return false;**  **}**  **Node \*p = head->next;**  **for (int j = 1; j < i; j++) {**  **p = p->next;**  **}**  **e = p->data;**  **return true;**  **}**  **int DList::locate(int e) const {**  **Node \*p = head->next;**  **for (int i = 1; p != tail; i++) {**  **if (p->data == e) {**  **return i;**  **}**  **p = p->next;**  **}**  **return 0;**  **}**  **bool DList::insert(int i, int e) {**  **if (length >= 100) {**  **std::cout << "线性表已满，无法插入！" << std::endl;**  **return false;**  **}**  **if (i < 1 || i > length + 1) {**  **std::cout << "插入位置无效！" << std::endl;**  **return false;**  **}**  **Node \*p = head;**  **for (int j = 0; j < i; j++) {**  **p = p->next;**  **}**  **Node \*q = new Node;**  **q->data = e;**  **q->prev = p->prev;**  **q->next = p;**  **p->prev->next = q;**  **p->prev = q;**  **length++;**  **return true;**  **}**  **bool DList::remove(int i, int &e) {**  **if (i < 1 || i > length) {**  **std::cout << "位置无效！" << std::endl;**  **return false;**  **}**  **Node \*p = head->next;**  **for (int j = 1; j < i; j++) {**  **p = p->next;**  **}**  **e = p->data;**  **p->prev->next = p->next;**  **p->next->prev = p->prev;**  **delete p;**  **length--;**  **return true;**  **}**  **void DList::display() const {**  **Node \*p = head->next;**  **while (p != tail) {**  **std::cout << p->data << " ";**  **p = p->next;**  **}**  **std::cout << std::endl;**  **}**    **#endif**  **Main.cpp**  **#include<iostream>**  **#include"include/DQueue.h"**  **using namespace std;**  **int main() {**  **DQueue queue;**  **int e;**  **std::cout << "测试队列基本操作：" << std::endl;**  **// 测试入队操作**  **std::cout << "\n===== 测试入队操作 =====" << std::endl;**  **queue.Insert(10);**  **queue.Insert(20);**  **queue.Insert(30);**  **queue.display();**  **// 测试获取队头元素**  **std::cout << "\n===== 测试获取队头元素 =====" << std::endl;**  **if (queue.front(e)) {**  **std::cout << "队头元素是：" << e << std::endl;**  **}**  **// 测试出队操作**  **std::cout << "\n===== 测试出队操作 =====" << std::endl;**  **if (queue.Delete(e)) {**  **std::cout << "出队元素是：" << e << std::endl;**  **}**  **queue.display();**  **// 测试其他操作**  **std::cout << "\n===== 测试其他操作 =====" << std::endl;**  **std::cout << "队列的大小：" << queue.size() << std::endl;**  **std::cout << "队列是否为空：" << (queue.isEmpty() ? "是" : "否") << std::endl;**  **// 测试清空操作**  **std::cout << "\n===== 测试清空操作 =====" << std::endl;**  **queue.clear();**  **std::cout << "清空后，队列的大小：" << queue.size() << std::endl;**  **std::cout << "清空后，队列是否为空：" << (queue.isEmpty() ? "是" : "否") << std::endl;**  **return 0;**  **}** | | | | | |
| 1. **实验结果与分析**   **测试代码验证了DQueue队列的FIFO特性及虚函数的多态实现。入队操作（Insert()）依次插入10、20、30后，队列显示顺序为10→20→30，符合队列尾部插入的逻辑；front()成功获取队头元素10，未破坏队列结构；出队操作（Delete()）移除并返回队头元素10，队列更新为20→30，证明队列头部删除的规则有效。此外，size()正确统计队列长度（初始3，出队后2，清空后0），isEmpty()和clear()功能正常，清空后队列状态准确。**  **通过将基类DList的Insert()和Delete()声明为纯虚函数，强制派生类（DQueue和DStack）根据数据结构特性重写方法：DQueue的Insert()在链表尾部追加（模拟入队），Delete()从头部移除（模拟出队）；而若为DStack，则Insert()对应头部插入（入栈），Delete()为头部删除（出栈），实现LIFO逻辑。这种设计通过虚函数统一接口，隐藏底层差异，用户可通过动态绑定的基类指针操作不同数据结构。**  **测试代码未显式包含动态创建逻辑（如根据输入类型实例化DQueue/DStack），但通过虚函数机制已为扩展奠定基础。若结合工厂模式，可进一步实现用户指定类型、动态创建对象的功能，增强代码灵活性。实验表明，虚函数有效分离接口与实现，符合开闭原则，为多态数据结构的扩展提供了清晰框架。**  **电脑萤幕画面  AI 生成的内容可能不正确。** | | | | | |
| 1. **实验心得**   通过本次实验，我深入理解了虚函数在面向对象编程中的核心作用及其对多态性的支持。在实现DQueue类继承DList基类时，纯虚函数**Insert()**和**Delete()**的设计强制派生类根据数据结构特性（队列的FIFO或栈的LIFO）重写方法，既规范了接口的统一性，又隐藏了实现细节。例如，DQueue中**Insert()**需在链表尾部追加元素，而DStack则需在头部插入，这种差异通过虚函数动态绑定得以无缝衔接，体现了“同一接口，多种实现”的多态思想。  实验过程中，如何确保基类与派生类方法的逻辑一致性是一大挑战。通过反复调试队列的出队和入队操作，并结合**display()**方法验证元素顺序，我进一步掌握了队列与栈的行为差异。此外，尽管测试代码未显式实现动态创建对象的功能，但基于虚函数的设计为后续扩展（如工厂模式动态实例化DQueue或DStack）提供了清晰的框架，符合开闭原则，增强了代码的可维护性。  此次实验让我认识到，虚函数不仅是实现多态的工具，更是解耦代码、提升灵活性的关键。通过抽象基类定义通用接口，派生类专注具体实现，能够有效降低模块间的依赖，为复杂系统的扩展奠定基础。这为今后设计可复用的数据结构库提供了重要启示。 | | | | | |
| 1. **教师评语** | | | | | |
| 1. **实验成绩**   教师签名： 蒋振刚 批阅日期： 2025 年 4 月 27日 | | | | | |

注：项目性质为 演示型、验证型、设计型、综合型和创新型。