**哈尔滨工程大学**

**《计算思维与问题求解（二）》实验报告**

**基础实践**

|  |  |  |  |
| --- | --- | --- | --- |
|  | 班级： | 20230641 |  |
|  | 学号： | 2023064108 |  |
|  | 姓名： | 高明亮 |  |

实验时间： 2024 年 第14-17周

|  |  |
| --- | --- |
| 成绩 |  |

**哈尔滨工程大学计算机学院**

实验一：类的继承

**一、实验环境及语言**

1. 开发环境：Visual studio，Dev cpp；

2. 编程语言：C++语言；

**二、实验目的**

1. 学习声明和使用类的继承关系，声明派生类。
2. 熟悉不同继承方式下对基类成员的访问控制。
3. 学习利用虚基类解决二义性问题。

**三、实验任务**

1. 声明一个基类Animal，有私有类成员变量age，构造其派生类dog，在其成员函数SetAge(int n)中直接给age赋值，会出现什么问题，把age改为公有成员变量，问题还存在吗。通过编程实现。

2. 声明一个基类BaseClass，有整型成员变量Number，构造其派生类DerivedClass，观察构造函数和析构函数的执行情况，用cout输出提示信息。

3. 声明一个车（vehicle）基类，具有MaxSpeed、Weight等成员变量，Run、Stop等成员函数，由此派生出自行车（bicycle）类、汽车（motorcar）类。自行车（bicycle）类有高度（Height）等属性，汽车（motorcar）类有座位数（SeatNum）等属性。从bicycle和motorcar派生出摩托车（motorcycle）类，在继承过程中，注意把vehicle设置为虚基类。如果不把vehicle设置为虚基类，会出现什么问题。通过编程说明。

4. 类图如图1所示，实现图中类之间的关系，注意虚基类的使用，重载相应的成员函数，并测试这些类。

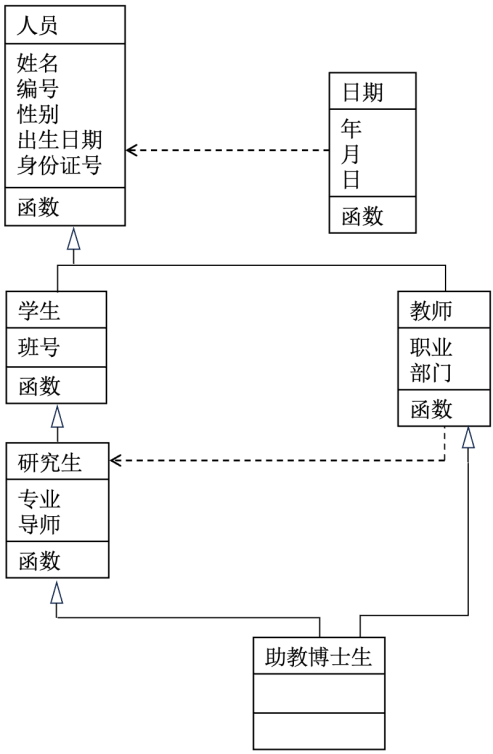


图1 类图

**三、设计思想**

1. 基类 Animal 中的 age 成员被设置为私有，这意味着在派生类 dog 外部不能直接访问 age。但是由于派生类 dog 继承了基类 Animal，所以在派生类内部是可以直接访问的。

2. 这段代码主要展示了基类和派生类之间的构造函数和析构函数的调用顺序，以及继承关系的基本特性。

3. 在 bicycle 和 motorcar 类中，使用了虚继承 virtual public vehicle。这样做可以避免在多重继承时产生二义性，确保只有一个 vehicle 类的实例被继承到 motorcycle 中。

4. 多重继承和虚继承在C++中的应用，以及类之间的组合关系。

**四、实验代码及注释**

**1.** #include<iostream>

using namespace std;

class Animal {

public: //这里将age设置为私有成员时，派生类公有继承时不可见，将age改为公有成员即可

int age;

};

class dog : public Animal {

public:

void SetAge(int n) {

cout << "请输入年龄:";

cin >> age;

cout << "年龄是:" << age;

}

};

int main() {

dog DOG;

cout << "年龄默认初始设置为0" << endl;

DOG.SetAge(0);

}

**2.** #include<iostream>

using namespace std;

class BaseClass {

public:

int Number;

BaseClass() {

cout << "BaseClass created" << endl;

}

~BaseClass() {

cout << "BaseClass destoryed" << endl;

}

};

class DerivedClass:public BaseClass{

public:

DerivedClass() {

cout << "DerivedClass created" << endl;

}

~DerivedClass() {

cout << "DerivedClass destoryed" << endl;

}

};

int main() {

//BaseClass base;

DerivedClass derived;

}

**3.** #include <iostream>

using namespace std;

class vehicle {

public:

int MaxSpeed;

float Weight;

vehicle(int maxSpeed, float weight) : MaxSpeed(maxSpeed), Weight(weight) {}

void Run() { cout << "Vehicle is running." << endl; }

void Stop() { cout << "Vehicle has stopped." << endl; }

};

class bicycle : virtual public vehicle {

public:

float Height;

bicycle(int maxSpeed, float weight, float height) : vehicle(maxSpeed, weight), Height(height) {}

};

class motorcar : virtual public vehicle {

public:

int SeatNum;

motorcar(int maxSpeed, float weight, int seatNum) : vehicle(maxSpeed, weight), SeatNum(seatNum) {}

};

class motorcycle : public bicycle, public motorcar {

public:

motorcycle(int maxSpeed, float weight, float height, int seatNum) : vehicle(maxSpeed, weight), bicycle(maxSpeed, weight, height), motorcar(maxSpeed, weight, seatNum) {}

};

int main() {

motorcycle moto(120, 200.5, 1.2, 2);

// 访问车辆属性

cout << "Max Speed: " << moto.MaxSpeed << endl;

moto.Run();

return 0;

}

**4.** #include<iostream>

#include<string>

using namespace std;

class date {

public:

int year, month, day;

date(int y = 0, int m = 0, int d = 0) :year(y), month(m), day(d) {}

};

class people :public date {

public:

string name;

int number;

string gender;

int id;

people(string name = "0", int number = 0, string gender = "0", int id = 0) :name(name), number(number), gender(gender), id(id) {}

virtual void run() {

cout << "people class running";

}

};

class student :virtual public people {

public:

int classnumber;

student(string name = "1", int number = 1, string gender = "1", int id = 1, int classnumber = 1) :people(name, number, gender, id), classnumber(classnumber) {}

};

class teacher :virtual public people {

public:

string department;

teacher(string name = "2", int number = 2, string gender = "2", int id = 2, string department = "2") :people(name, number, gender, id), department(department) {}

void run() {

cout << "teacher class running"; //重载run函数

}

};

class researchstudent :virtual public student {

public:

teacher a = teacher("0", 1, "0", 0, "0");

researchstudent(string name = "3", int number = 3, string gender = "3", int id = 3, int classnumer = 3) :student(name, number, gender, id, classnumber) {}

};

class doctor :public researchstudent, public teacher {

};

int main() {

doctor abc;

cout << abc.id << endl; //显示的是people类的成员值

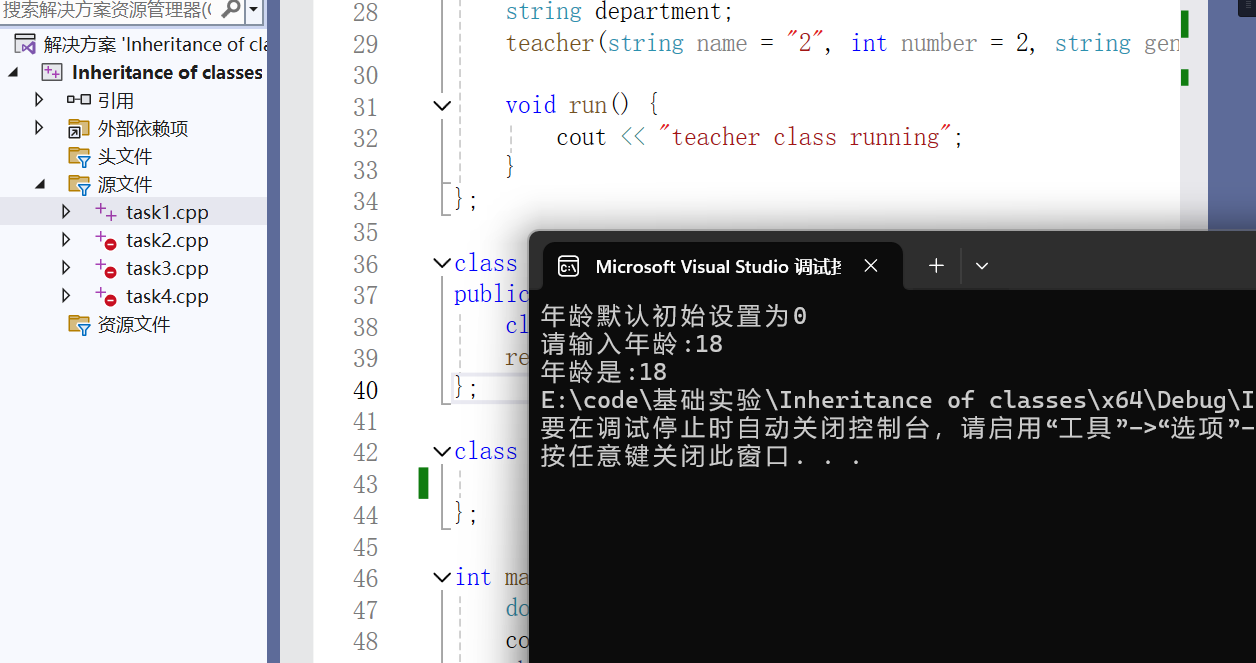
abc.run(); //调用距离最近的函数，在teacher类而并非people类

return 0;

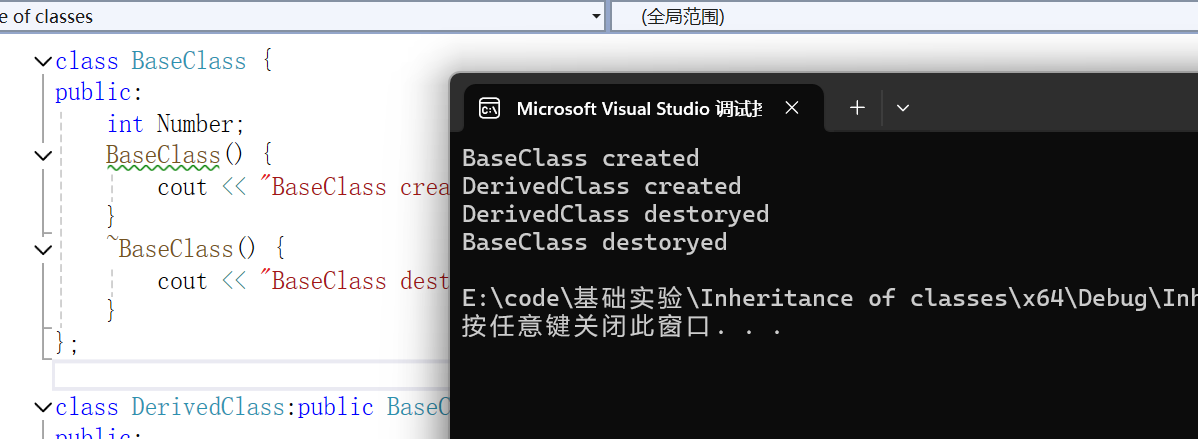
}

**五、验证与结论**

**1.**



**2.**



**3.**

文本

描述已自动生成

**4.**



**六、总结与心得体会**

继承是面向对象编程中非常重要的概念之一，通过继承，可以实现代码的重用，减少重复编写相似功能的代码。在公有继承中，基类的公有成员在派生类中仍然是公有的，可以被派生类和外部访问；在私有继承中，基类的公有成员在派生类中变为私有的，只能在派生类内部访问；在保护继承中，基类的公有成员在派生类中变为保护的，只能在派生类和其派生类的成员函数中访问。多重继承可能会导致二义性问题，即一个派生类同时继承了多个具有相同成员的基类。为了解决这个问题，可以使用虚基类。虚基类是通过在继承关系中使用虚继承来实现的，它可以确保在派生类中只包含一份虚基类的实例。这样就避免了多重继承导致的二义性问题，确保了继承关系的清晰性和可维护性。通过上述实践，对类的继承关系有了更深入的理解，同时也掌握了处理继承中可能出现的访问控制和二义性问题的方法。

实验二：多态性

**一、实验环境及语言**

1. 开发环境：Visual studio，Dev cpp；

2. 编程语言：C++语言；

**二、实验目的**

1. 掌握运算符重载方法。

2. 学习使用虚函数实现动态多态性。

**三、实验任务**

1. 声明Point类，有坐标\_x，\_y两个成员变量。对Point类重载”++”（自增）、”--”（自减）运算符，实现对坐标值的改变。

2. 声明一个车（vehicle）基类，具有Run、Stop等成员函数。由此派生出自行车（bicycle）类、汽车（motorcar）类，从bicycle和motorcar派生出摩托车（motorcycle）类，它们都有Run、Stop等成员函数。在main()函数中声明vehicle、bicycle、motorcar、motorcycle的对象，调用其Run()、Stop()函数，观察其执行情况。再分别用vehicle类型的指针来调用这几个对象的成员函数，看看能否成功；把Run、Stop声明为虚函数，再试试看。

3. 对实验一任务4中的people类重载”==”运算符和”=”运算符，其中”==”运算符判断两个people类对象的id属性大小，”=”运算符实现people类对象的赋值操作。

**三、设计思想**

1. 实现如何重载C++中的前置和后置自增（++）和自减（--）运算符，并观察它们在类Point中的行为。通过实验，可以了解这些运算符在对象上的作用以及它们在前置和后置形式下的区别。

2. 实现C++中的虚函数机制、虚继承、多重继承以及通过基类指针访问派生类对象的行为，了解这些概念在类层次结构中的应用。

3. 展示C++中如何通过重载运算符实现对象的比较和赋值操作。在这个实验中，主要通过对于people类中==和=运算符的重载，并创建具体的实例去验证重载后的功能。

**四、实验代码及注释**

**1.** #include <iostream>

using namespace std;

class Point {

private:

int \_x;

int \_y;

public:

Point(int x, int y) : \_x(x), \_y(y) {}

void display() const {

cout << "(" << \_x << ", " << \_y << ")" << endl;

}

Point& operator++() { // 重载前置自增运算符 ++

++\_x;

++\_y;

return \*this;

}

Point& operator--() { // 重载前置自减运算符 --

--\_x;

--\_y;

return \*this;

}

Point operator++(int) { // 重载后置自减运算符 --

Point old = \*this;

\_x++;

\_y++;

return old;

}

Point operator--(int) { // 重载后置自减运算符 --

Point old = \*this;

\_x--;

\_y--;

return old;

}

};

int main() {

int a, b;

cout << "请输入初始坐标:";

cin >> a >> b;

Point p( a,b );

cout << "初始坐标:";

p.display();

cout << "前置自增:";

(++p).display(); // 前置自增

cout << "前置自减:";

(--p).display(); // 前置自减

cout << "后置自增:";

(p++).display(); // 后置自增

cout << "后置自减:";

(p--).display(); // 后置自减

return 0;

}

**2.** #include <iostream>

using namespace std;

class vehicle {

public:

int MaxSpeed;

float Weight;

vehicle(int maxSpeed, float weight) : MaxSpeed(maxSpeed), Weight(weight) {}

virtual void Run() { cout << "Vehicle is running." << endl; } //若没有声明虚函数，则通过vehicle指针无法运行

virtual void Stop() { cout << "Vehicle has stopped." << endl; } //若没有声明虚函数，则通过vehicle指针无法运行

virtual void show() { cout << MaxSpeed << "," << Weight << endl; } //若没有声明虚函数，则通过vehicle指针无法运行

};

class bicycle : virtual public vehicle {

public:

float Height;

bicycle(int maxSpeed, float weight, float height) : vehicle(maxSpeed, weight), Height(height) {}

void Run() override { cout << "bicycle is running." << endl; }

void Stop() override { cout << "bicycle has stopped." << endl; }

void show() override { cout << MaxSpeed << "," << Weight << "," << Height << endl; }

};

class motorcar : virtual public vehicle {

public:

int SeatNum;

motorcar(int maxSpeed, float weight, int seatNum) : vehicle(maxSpeed, weight), SeatNum(seatNum) {}

void Run() override { cout << "motorcar is running." << endl; }

void Stop() override { cout << "motorcar has stopped." << endl; }

void show() override { cout << MaxSpeed << "," << Weight << "," << SeatNum << endl; }

};

class motorcycle : public bicycle, public motorcar {

public:

motorcycle(int maxSpeed, float weight, float height, int seatNum) : vehicle(maxSpeed, weight), bicycle(maxSpeed, weight, height), motorcar(maxSpeed, weight, seatNum) {}

void Run() override { cout << "motorcycle is running." << endl; }

void Stop() override { cout << "motorcycle has stopped." << endl; }

void show() override { cout << MaxSpeed << "," << Weight << "," << Height << "," << SeatNum << endl; }

};

int main() {

vehicle v(100, 100);

v.Run();

v.Stop();

v.show();

cout << endl;

//正常直接调用

bicycle b(40, 10, 10);

b.Run();

b.Stop();

b.show();

cout << endl;

motorcar m(150, 200, 5);

m.Run();

m.Stop();

m.show();

cout << endl;

motorcycle mc(120, 150, 20, 3);

mc.Run();

mc.Stop();

mc.show();

cout << endl;

//使用vehicle类型指针调用

vehicle\* vPtr;

vPtr = &b;

vPtr->Run();

vPtr->Stop();

vPtr->show();

cout << endl;

vPtr = &m;

vPtr->Run();

vPtr->Stop();

vPtr->show();

cout << endl;

vPtr = &mc;

vPtr->Run();

vPtr->Stop();

vPtr->show();

cout << endl;

return 0;

}

**3.** #include<iostream>

#include<string>

using namespace std;

// date类，表示日期

class date {

public:

int year, month, day;

date(int y = 0, int m = 0, int d = 0) :year(y), month(m), day(d) {}

};

// people类，继承自date类，表示人

class people : public date {

public:

string name;

int number;

string gender;

int id;

// 构造函数

people(string name = "0", int number = 0, string gender = "0", int id = 0)

: name(name), number(number), gender(gender), id(id) {}

// 重载==运算符，比较两个people对象的id属性

bool operator==(const people& other) const {

return this->id == other.id;

}

// 重载赋值运算符，实现people类对象的赋值操作

people& operator=(const people& other) {

if (this != &other) {

this->name = other.name;

this->number = other.number;

this->gender = other.gender;

this->id = other.id;

this->year = other.year;

this->month = other.month;

this->day = other.day;

}

return \*this;

}

};

// student类，虚继承自people类，表示学生

class student : virtual public people {

public:

int classnumber;

student(string name = "1", int number = 1, string gender = "1", int id = 1, int classnumber = 1)

: people(name, number, gender, id), classnumber(classnumber) {}

};

// teacher类，虚继承自people类，表示老师

class teacher : virtual public people {

public:

string department;

teacher(string name = "2", int number = 2, string gender = "2", int id = 2, string department = "2")

: people(name, number, gender, id), department(department) {}

// 示例成员函数

void run() {

cout << "teacher class running" << endl;

}

};

// researchstudent类，虚继承自student类，表示研究生

class researchstudent : virtual public student {

public:

teacher a = teacher("0", 1, "0", 0, "0");

researchstudent(string name = "3", int number = 3, string gender = "3", int id = 3, int classnumber = 3)

: student(name, number, gender, id, classnumber) {}

};

// doctor类，继承自researchstudent和teacher类，表示博士生

class doctor : public researchstudent, public teacher {

public:

doctor(string name = "4", int number = 4, string gender = "4", int id = 4, int classnumber = 4, string department = "4")

: people(name, number, gender, id), researchstudent(name, number, gender, id, classnumber), teacher(name, number, gender, id, department) {}

};

int main() {

// 创建doctor类对象

doctor def("HEU", 123, "Male", 211, 789, "Physics");

doctor abc("HIT", 456, "Female", 985, 101, "Math");

cout << "HEU's ID before assignment: " << def.id << endl;

cout << "HIT's ID before assignment: " << abc.id << endl;

if (abc == def) {

cout << "HEU and HIT have the same ID." << endl;

}

else {

cout << "HEU and HIT have different ID." << endl;

}

// 测试赋值运算符

def = abc;

cout << "HEU's ID after assignment: " << def.id << endl;

// 测试等于运算符

if (abc == def) {

cout << "HEU and HIT have the same ID." << endl;

}

else {

cout << "HEU and HIT have different ID." << endl;

}

}

**五、验证与结论**

**1.**

文本

描述已自动生成

**2.**

图片包含 日程表

描述已自动生成

**3.**

文本

描述已自动生成

**六、总结与心得体会**

在上述三个实验中，展示了C++面向对象编程的核心概念，包括继承、多态、运算符重载和虚函数。

在第一个实验中，使用虚函数来确保基类指针或引用能够调用派生类的重载方法，多态性使得程序具有更高的灵活性和可扩展性。虚函数的使用使得我们可以通过基类指针或引用来调用实际派生类的实现。例如，vehicle指针可以调用bicycle、motorcar和motorcycle的Run、Stop和show方法，通过直接调用和基类指针调用两种方式来测试虚函数的行为，验证了多态性的实现。

在第二个试验里，重载了Point类内的前置和后置的自增和自减运算符，可以直接使用++和--来操作Point的实例对象，而不需要调用特定的成员函数，简化了操作的同时也实现了功能的多样化和自定义。通过打印操作前后的坐标值，我们验证了前置和后置运算符的正确性。

在第三个实验中，重载了相等判断运算符(==)和赋值运算符(=)，展示了如何在继承结构中实现运算符重载。特别地，通过重载==运算符，我们可以方便地比较两个people对象的id属性。通过重载=运算符，我们可以确保对象赋值时所有属性都正确复制。同时还展示了虚继承和多重继承的应用，解决了多重继承中的菱形继承问题，确保基类people的属性在派生类doctor中只有一份实例。借助虚继承，可以在复杂的继承层次结构中保持类设计的清晰性和数据的一致性。在复杂的继承层次结构中保持类设计的清晰性，解决了多重继承中的菱形继承问题，确保数据的一致性。

实验三：模板与群体类

**一、实验环境及语言**

1. 开发环境：Visual studio，Dev cpp；

2. 编程语言：C++语言；

**二、实验目的**

1. 了解链表类的声明与实现，学习其使用方法。
2. 了解栈类的声明与实现，学习其使用方法。
3. 了解队列类的声明与实现，学习其使用方法
4. 掌握对数组元素排序的方法。
5. 掌握对数组元素查找的方法。

**三、实验任务**

1. 编写程序link.h，实现教材例9-6的链表类，在测试程序中声明两个整型链表A和B，分别插入5个元素，并把B中元素加入A的尾部。

2. 下面任务二选一，也可双选：

（1）编写程序queue.h，用链表实现队列类。在测试程序中声明一个整型队列对象，插入5个整数，压入队列，再依据顺序取出并显示出来。

（2）编写程序stack.h，用链表实现栈类。在测试程序中声明一个整型栈对象，插入5个整数，压入栈，再依据相应顺序取出并显示出来。

3. 将直接插入排序、直接选择排序、冒泡排序与顺序查找函数封装到数组类中，作为成员函数，实现并设计这个类。

4. 声明一个对people类对象数组按编号排序的函数，一个按编号查找people对象的函数。在测试程序中使用前面实验得到的结果声明教师数组和学生数组，分别对你教师数组和学生数组进行排序和查找。

**三、设计思想**

**1.链表类**

链表是一种由一系列节点组成的常见的数据结构，每个节点中包含两个部分：数据域和指向下一个节点的指针域，在设计LinkList.h时，需要封装链表操作，对外提供统一接口，隐藏内部细节，且使用模版设计，提高类的通用性和灵活性来适应任意类型的数据。链表类里提供多种插入和删除操作函数，如insertFront、insertRear、insertAt、insertAfter、deleteFront和deleteCurrent，而借助getSize，isEmpty，currentPosition函数，可以方便的获取到链表的各项信息，以及当前遍历节点的位置。

**2.队列类和栈类**

(1)队列类是一种允许在一端添加元素（队尾），并在另一端移除元素（队首）的遵循先进先出的数据结构，与链表类的模版设计类似，同样能够提高类的灵活性和通用型，提供统一外部接口，隐藏内部细节。提供入队（insert），出队（remove），访问队首元素（getFront），清空队列（clear）等基本操作函数，同时也包括了测试队列是否为空（isEmpty），队列为满（isFull）和获取队列长度（getLength）的状态查询函数。

(2)栈类与队列不同，栈允许在一端对数据进行添加和移除操作（栈顶），所以遵循先进后出的原则，同样使用模版化，使其能够存储任意类型的数据，利用栈顶指针使得栈的操作变得简单直观，使用类操作将栈的操作封装起来，对外提供统一的接口，隐藏了内部实现细节。提供操作函数包括压栈（push）、弹栈（pop）、访问栈顶元素（peek）和清空栈（clear），以及测试栈是否为空（isEmpty）和是否已满（isFull）的方法，方便检查栈的状态。

**3.数组排序**

数组类提供了几种基本的排序算法实现，如插入排序（insertionSort）、选择排序（selectionSort）和冒泡排序（bubbleSort），调用这些函数可以直接对数组元素进行排序。同时设计了一些辅助函数来扩展类的功能，包括线性搜索（linearSearch）方法，可以在数组中查找特定值的位置；(addElement)方法来添加新元素到数组，以及(printArray)方法来打印数组内容。

**4.类对象数组排序**

通过面向对象的设计，创建了表示人员（包括学生和教师）信息的类，并实现了对这些对象的排序和查找功能，学生类和教师类继承自人员类，复用了人员类的成员变量和方法

排序功能 (sortPeopleArray) 基于number成员变量对人员对象数组进行排序，使用lambda表达式作为排序准则，比较两个人员对象的number值。查找功能 (searchPeopleByNumber)通过遍历人员对象数组，查找number与指定值匹配的对象，找到则返回指向该对象的指针，否则返回nullptr。

通过在主函数中创建教师和学生数组，调用sortPeopleArray函数，对教师和学生数组进行排序。经过排序后显示教师和学生信息，遍历排序后的数组，输出每个对象的name, number, 和生日信息。按number查找教师，通过输入的number值，调用searchPeopleByNumber函数查找对应的教师和学生对象并输出查找结果。

**四、实验代码及注释**

**1.链表类**

//Node.h

#ifndef NODE\_H

#define NODE\_H

//类模板的定义

template <class T>

class Node {

private:

Node<T>\* next; //指向后继结点的指针

public:

T data; //数据域

Node(const T& data, Node<T>\* next = 0); //构造函数

void insertAfter(Node<T>\* p); //在本结点之后插入一个同类结点p

Node<T>\* deleteAfter(); //删除本结点的后继结点，并返回其地址

Node<T>\* nextNode(); //获取后继结点的地址

const Node<T>\* nextNode() const; //获取后继结点的地址

};

//类的实现部分

//构造函数，初始化数据和指针成员

template <class T>

Node<T>::Node(const T& data, Node<T>\* next/\* = 0 \*/) : data(data), next(next) { }

//返回后继结点的指针

template <class T>

Node<T>\* Node<T>::nextNode() {

return next;

}

//返回后继结点的指针

template <class T>

const Node<T>\* Node<T>::nextNode() const {

return next;

}

//在当前结点之后插入一个结点p

template <class T>

void Node<T>::insertAfter(Node<T>\* p) {

p->next = next; //p结点指针域指向当前结点的后继结点

next = p; //当前结点的指针域指向p

}

//删除当前结点的后继结点，并返回其地址

template <class T>

Node<T>\* Node<T>::deleteAfter() {

Node<T>\* tempPtr = next; //将欲删除的结点地址存储到tempPtr中

if (next == 0) //如果当前结点没有后继结点，则返回空指针

return 0;

next = tempPtr->next; //使当前结点的指针域指向tempPtr的后继结点

return tempPtr; //返回被删除的结点的地址

}

#endif //NODE\_H

#pragma once

//LinkList.h

#ifndef LINKEDLIST\_H

#define LINKEDLIST\_H

#include "Node.h"

#include<iostream>

using namespace std;

template <class T>

class LinkedList {

private:

//数据成员：

Node<T> \*front, \*rear; //表头和表尾指针

Node<T> \*prevPtr, \*currPtr; //记录表当前遍历位置的指针，由插入和删除操作更新

int size; //表中的元素个数

int position; //当前元素在表中的位置序号。由函数reset使用

//函数成员：

//生成新结点，数据域为item，指针域为ptrNext

Node<T> \*newNode(const T &item,Node<T> \*ptrNext=NULL);

//释放结点

void freeNode(Node<T> \*p);

//将链表L 拷贝到当前表（假设当前表为空）。

//被拷贝构造函数、operator = 调用

void copy(const LinkedList<T>& L);

public:

LinkedList(); //构造函数

LinkedList(const LinkedList<T> &L); //拷贝构造函数

~LinkedList(); //析构函数

LinkedList<T> & operator = (const LinkedList<T> &L); //重载赋值运算符

int getSize() const; //返回链表中元素个数

bool isEmpty() const; //链表是否为空

void reset(int pos = 0);//初始化游标的位置

void next(); //使游标移动到下一个结点

bool endOfList() const; //游标是否到了链尾

int currentPosition() const; //返回游标当前的位置

void insertFront(const T &item); //在表头插入结点

void insertRear(const T &item); //在表尾添加结点

void insertAt(const T &item); //在当前结点之前插入结点

void insertAfter(const T &item); //在当前结点之后插入结点

T deleteFront(); //删除头结点

void deleteCurrent(); //删除当前结点

T& data(); //返回对当前结点成员数据的引用

const T& data() const; //返回对当前结点成员数据的常引用

//清空链表：释放所有结点的内存空间。被析构函数、operator= 调用

void clear();

};

template <class T>

Node<T> \*LinkedList<T>::newNode(const T& item, Node<T>\* ptrNext) //生成新结点

{

Node<T> \*p;

p = new Node<T>(item, ptrNext);

if (p == NULL)

{

cout << "Memory allocation failure!\n";

exit(1);

}

return p;

}

template <class T>

void LinkedList<T>::freeNode(Node<T> \*p) //释放结点

{

delete p;

}

template <class T>

void LinkedList<T>::copy(const LinkedList<T>& L) //链表复制函数

{

Node<T> \*p = L.front; //P用来遍历L

int pos;

while (p != NULL) //将L中的每一个元素插入到当前链表最后

{

insertRear(p->data);

p = p->nextNode();

}

if (position == -1) //如果链表空,返回

return;

//在新链表中重新设置prevPtr和currPtr

prevPtr = NULL;

currPtr = front;

for (pos = 0; pos != position; pos++)

{

prevPtr = currPtr;

currPtr = currPtr->nextNode();

}

}

template <class T> //构造一个新链表，将有关指针设置为空，size为0，position为-1

LinkedList<T>::LinkedList() : front(NULL), rear(NULL),

prevPtr(NULL), currPtr(NULL), size(0), position(-1)

{}

template <class T>

LinkedList<T>::LinkedList(const LinkedList<T>& L) //拷贝构造函数

{

front = rear = NULL;

prevPtr = currPtr = NULL;

size = 0;

position = -1;

copy(L);

}

template <class T>

LinkedList<T>::~LinkedList() //析构函数

{

clear();

}

template <class T>

LinkedList<T>& LinkedList<T>::operator=(const LinkedList<T>& L)//重载"="

{

if (this == &L) // 不能将链表赋值给它自身

return \*this;

clear();

copy(L);

return \*this;

}

template <class T>

int LinkedList<T>::getSize() const //返回链表大小的函数

{

return size;

}

template <class T>

bool LinkedList<T>::isEmpty() const //判断链表为空否

{

return size == 0;

}

template <class T>

void LinkedList<T>::reset(int pos) //将链表当前位置设置为pos

{

int startPos;

if (front == NULL) // 如果链表为空，返回

return;

if (pos < 0 || pos > size - 1) // 如果指定位置不合法，中止程序

{

cerr << "Reset: Invalid list position: " << pos << endl;

return;

}

// 设置与遍历链表有关的成员

if (pos == 0) // 如果pos为0，将指针重新设置到表头

{

prevPtr = NULL;

currPtr = front;

position = 0;

}

else // 重新设置 currPtr, prevPtr, 和 position

{

currPtr = front->nextNode();

prevPtr = front;

startPos = 1;

for (position = startPos; position != pos; position++)

{

prevPtr = currPtr;

currPtr = currPtr->nextNode();

}

}

}

template <class T>

void LinkedList<T>::next() //将prevPtr和currPtr向前移动一个结点

{

if (currPtr != NULL)

{

prevPtr = currPtr;

currPtr = currPtr->nextNode();

position++;

}

}

template <class T>

bool LinkedList<T>::endOfList() const // 判断是否已达表尾

{

return currPtr == NULL;

}

template <class T>

int LinkedList<T>::currentPosition() const // 返回当前结点的位置

{

return position;

}

template <class T>

void LinkedList<T>::insertFront(const T& item) // 将item插入在表头

{

if (front != NULL) // 如果链表不空则调用Reset

reset();

insertAt(item); // 在表头插入

}

template <class T>

void LinkedList<T>::insertRear(const T& item) // 在表尾插入结点

{

Node<T> \*nNode;

prevPtr = rear;

nNode = newNode(item); // 创建新结点

if (rear == NULL) // 如果表空则插入在表头

front = rear = nNode;

else

{

rear->insertAfter(nNode);

rear = nNode;

}

currPtr = rear;

position = size;

size++;

}

template <class T>

void LinkedList<T>::insertAt(const T& item) // 将item插入在链表当前位置

{

Node<T> \*nNode;

if (prevPtr == NULL) // 插入在链表头，包括将结点插入到空表中

{

nNode = newNode(item, front);

front = nNode;

}

else // 插入到链表之中. 将结点置于prevPtr之后

{

nNode = newNode(item);

prevPtr->insertAfter(nNode);

}

if (prevPtr == rear) //正在向空表中插入，或者是插入到非空表的表尾

{

rear = nNode; //更新rear

position = size; //更新position

}

currPtr = nNode; //更新currPtr

size++; //使size增值

}

template <class T>

void LinkedList<T>::insertAfter(const T& item) // 将item 插入到链表当前位置之后

{

Node<T> \*p;

p = newNode(item);

if (front == NULL) // 向空表中插入

{

front = currPtr = rear = p;

position = 0;

}

else // 插入到最后一个结点之后

{

if (currPtr == NULL)

currPtr = prevPtr;

currPtr->insertAfter(p);

if (currPtr == rear)

{

rear = p;

position = size;

}

else

position++;

prevPtr = currPtr;

currPtr = p;

}

size++; // 使链表长度增值

}

template <class T>

T LinkedList<T>::deleteFront() // 删除表头结点

{

T item;

reset();

if (front == NULL)

{

cerr << "Invalid deletion!" << endl;

exit(1);

}

item = currPtr->data;

deleteCurrent();

return item;

}

template <class T>

void LinkedList<T>::deleteCurrent() // 删除链表当前位置的结点

{

Node<T> \*p;

if (currPtr == NULL) // 如果表空或达到表尾则出错

{

cerr << "Invalid deletion!" << endl;

exit(1);

}

if (prevPtr == NULL) // 删除将发生在表头或链表之中

{

p = front; // 保存头结点地址

front = front->nextNode(); //将其从链表中分离

}

else //分离prevPtr之后的一个内部结点，保存其地址

p = prevPtr->deleteAfter();

if (p == rear) // 如果表尾结点被删除

{

rear = prevPtr; //新的表尾是prevPtr

position--; //position自减

}

currPtr = p->nextNode(); // 使currPtr越过被删除的结点

freeNode(p); // 释放结点，并

size--; //使链表长度自减

}

template <class T>

T& LinkedList<T>::data() //返回一个当前结点数值的引用

{

if (size == 0 || currPtr == NULL) // 如果链表为空或已经完成遍历则出错

{

cerr << "Data: invalid reference!" << endl;

exit(1);

}

return currPtr->data;

}

template <class T>

void LinkedList<T>::clear() //清空链表

{

Node<T> \*currPosition, \*nextPosition;

currPosition = front;

while (currPosition != NULL)

{

nextPosition = currPosition->nextNode(); //取得下一结点的地址

freeNode(currPosition); //删除当前结点

currPosition = nextPosition; //当前指针移动到下一结点

}

front = rear = NULL;

prevPtr = currPtr = NULL;

size = 0;

position = -1;

}

#endif //LINKEDLIST\_H

#pragma once

//main.cpp

//9\_6.cpp

#include <iostream>

#include "LinkedList.h"

using namespace std;

int main() {

LinkedList<int>A;

cout << "请输入链表A的内容：";

for (int i = 0; i < 5; i++) {

int node;

cin >> node;

A.insertFront(node);

}

//输出链表

cout << "LIST A: ";

A.reset();

//输出各节点

while (!A.endOfList()) {

cout << A.data() << " ";

A.next();

}

cout << endl << endl;

LinkedList<int>B;

cout << "请输入链表B的内容：";

for (int i = 0; i < 5; i++) {

int node;

cin >> node;

B.insertFront(node);

}

//输出链表

cout << "LIST B: ";

B.reset();

//输出各节点

while (!B.endOfList()) {

cout << B.data() << " ";

B.next();

}

cout << endl << endl;

// 遍历 B 并把 B 中元素加入 A 的尾部

for (B.reset(); !B.endOfList(); B.next()) {

A.insertRear(B.data());

}

// 验证

cout << "添加过后的LIST A: ";

for (A.reset(); !A.endOfList(); A.next()) {

cout << A.data() << " ";

}

cout << endl;

return 0;

}

**2.（1）队列类**

//queue.h

#ifndef QUEUE\_H

#define QUEUE\_H

#include <cassert>

//类模板的定义

template <class T, int SIZE = 50>

class Queue {

private:

int front, rear, count; //队头指针、队尾指针、元素个数

T list[SIZE]; //队列元素数组

public:

Queue(); //构造函数，初始化队头指针、队尾指针、元素个数

void insert(const T& item); //新元素入队

T remove(); //元素出队

void clear(); //清空队列

const T& getFront() const; //访问队首元素

//测试队列状态

int getLength() const; //求队列长度（元素个数）

bool isEmpty() const; //判队队列空否

bool isFull() const; //判断队列满否

};

//构造函数，初始化队头指针、队尾指针、元素个数

template <class T, int SIZE>

Queue<T, SIZE>::Queue() : front(0), rear(0), count(0) { }

template <class T, int SIZE>

void Queue<T, SIZE>::insert(const T& item) { //向队尾插入元素（入队）

assert(count != SIZE);

count++; //元素个数增1

list[rear] = item; //向队尾插入元素

rear = (rear + 1) % SIZE; //队尾指针增1，用取余运算实现循环队列

}

template <class T, int SIZE>

T Queue<T, SIZE>::remove() { //删除队首元素，并返回该元素的值（出队）

assert(count != 0);

int temp = front; //记录下原先的队首指针

count--; //元素个数自减

front = (front + 1) % SIZE; //队首指针增1。取余以实现循环队列

return list[temp]; //返回首元素值

}

template <class T, int SIZE>

const T& Queue<T, SIZE>::getFront() const { //访问队列首元素（返回其值）

return list[front];

}

template <class T, int SIZE>

int Queue<T, SIZE>::getLength() const { //返回队列元素个数

return count;

}

template <class T, int SIZE>

bool Queue<T, SIZE>::isEmpty() const { //测试队空否

return count == 0;

}

template <class T, int SIZE>

bool Queue<T, SIZE>::isFull() const { //测试队满否

return count == SIZE;

}

template <class T, int SIZE>

void Queue<T, SIZE>::clear() { //清空队列

count = 0;

front = 0;

rear = 0;

}

#endif //QUEUE\_H

//main.cpp

#include<iostream>

#include"queue.h"

using namespace std;

int main() {

Queue<int, 5> queue;

cout << "请输入5个整数进入队列:";

for (int i = 0; i < 5; i++) {

int num;

cin >> num;

queue.insert(num);

}

cout << endl;

cout << "队列依次输出的数:";

while (!queue.isEmpty()) {

cout << queue.getFront() << " ";

queue.remove();

}

cout << endl;

}

**2.（2）栈类**

//stack.h

#ifndef STACK\_H

#define STACK\_H

#include <cassert>

//模板的定义，SIZE为栈的大小

template <class T, int SIZE = 50>

class Stack {

private:

T list[SIZE]; //数组，用于存放栈的元素

int top; //栈顶位置（数组下标）

public:

Stack(); //构造函数，初始化栈

void push(const T& item); //将元素item压入栈

T pop(); //将栈顶元素弹出栈

void clear(); //将栈清空

const T& peek() const; //访问栈顶元素

bool isEmpty() const; //测试是否栈满

bool isFull() const; //测试是否栈空

};

//模板的实现

template <class T, int SIZE>

Stack<T, SIZE>::Stack() : top(-1) { } //构造函数，栈顶初始化为-1

template <class T, int SIZE>

void Stack<T, SIZE>::push(const T& item) { //将元素item压入栈

assert(!isFull()); //如果栈满了，则报错

list[++top] = item; //将新元素压入栈顶

}

template <class T, int SIZE>

T Stack<T, SIZE>::pop() { //将栈顶元素弹出栈

assert(!isEmpty()); //如果栈为空，则报错

return list[top--]; //返回栈顶元素，并将其弹出栈顶

}

template <class T, int SIZE>

const T& Stack<T, SIZE>::peek() const { //访问栈顶元素

assert(!isEmpty()); //如果栈为空，则报错

return list[top]; //返回栈顶元素

}

template <class T, int SIZE>

bool Stack<T, SIZE>::isEmpty() const { //测试栈是否空

return top == -1;

}

template <class T, int SIZE>

bool Stack<T, SIZE>::isFull() const { //测试是否栈满

return top == SIZE - 1;

}

template <class T, int SIZE>

void Stack<T, SIZE>::clear() { //清空栈

top = -1;

}

//main.cpp

#include <iostream>

#include "Stack.h" // 包含头文件

using namespace std;

int main() {

Stack<int, 5> intStack; // 声明一个大小为5的整型栈对象

// 将五个整数压入栈

cout << "请输入五个数压入栈:";

for (int i = 0; i < 5; i++) {

int num;

cin >> num;

intStack.push(num);

}

cout << endl;

// 依次弹出并显示栈中的整数

cout << "弹出栈的数:";

while (!intStack.isEmpty()) {

cout << intStack.pop() << " ";

}

cout << endl;

return 0;

}

1. **数组排序**

//ArrayClass.h

#ifndef ARRAYCLASS\_H

#define ARRAYCLASS\_H

#include <iostream>

using namespace std;

template <typename T>

class ArrayClass {

private:

T\* arr; // 数组元素

int size; // 当前数组元素数量

int capacity; // 数组最大容量

// 私有成员函数：交换两个数组元素

void swapElements(int index1, int index2) {

T temp = arr[index1];

arr[index1] = arr[index2];

arr[index2] = temp;

}

public:

// 构造函数

ArrayClass(int cap) : size(0), capacity(cap) {

arr = new T[capacity];

}

// 析构函数

~ArrayClass() {

delete[] arr;

}

// 成员函数：直接插入排序

void insertionSort() {

for (int i = 1; i < size; i++) {

T key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

// 成员函数：直接选择排序

void selectionSort() {

for (int i = 0; i < size - 1; i++) {

int minIndex = i;

for (int j = i + 1; j < size; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

if (minIndex != i) {

swapElements(i, minIndex);

}

}

}

// 成员函数：冒泡排序

void bubbleSort() {

for (int i = 0; i < size - 1; i++) {

for (int j = 0; j < size - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

swapElements(j, j + 1);

}

}

}

}

// 成员函数：顺序查找

int linearSearch(const T& value) {

for (int i = 0; i < size; i++) {

if (arr[i] == value) {

return i;

}

}

return -1; // 元素未找到

}

// 成员函数：添加元素

bool addElement(const T& value) {

if (size >= capacity) {

return false; // 数组已满

}

arr[size++] = value;

return true;

}

// 成员函数：打印数组

void printArray() {

for (int i = 0; i < size; i++) {

cout << arr[i] << " ";

}

cout << endl;

}

// 成员函数：获取数组大小

int getSize() const {

return size;

}

};

#endif // ARRAYCLASS\_H

//main.cpp

#include "ArrayClass.h"

int main() {

using namespace std;

// 创建一个整数类型的数组类实例，容量为10

ArrayClass<int> myArray(10);

// 向数组中添加一些元素

cout << "请输入五个原始数据：" << endl;

for (int i = 0; i < 5; i++) {

int num;

cin >> num;

myArray.addElement(num);

}

// 打印原始数组

cout << "原始数组: ";

myArray.printArray();

// 执行直接插入排序

myArray.insertionSort();

cout << "插入排序后: ";

myArray.printArray();

// 执行直接选择排序

myArray.selectionSort();

cout << "选择排序后: ";

myArray.printArray();

// 执行冒泡排序

myArray.bubbleSort();

cout << "冒泡排序后: ";

myArray.printArray();

// 测试顺序查找

cout << "输入要查找的数据：" << endl;

int valueToFind;

cin >> valueToFind;

int foundIndex = myArray.linearSearch(valueToFind);

if (foundIndex != -1) {

cout << "元素 " << valueToFind << " 找到，位于数组下标 " << foundIndex << endl;

}

else {

cout << "元素 " << valueToFind << " 未找到。" << endl;

}

return 0;

}

1. **类对象数组排序**

#include <iostream>

#include <string>

#include <algorithm>

using namespace std;

// 定义日期类

class date {

public:

int year, month, day;

date(int y = 0, int m = 0, int d = 0) : year(y), month(m), day(d) {}

};

// 定义人员类，包含日期对象

class people {

public:

string name;

int number;

string gender;

int id;

date birthday; // 包含日期对象

people(string name = "0", int number = 0, string gender = "0", int id = 0, date birthday = date())

: name(name), number(number), gender(gender), id(id), birthday(birthday) {}

};

// 定义学生类，继承人员类

class student : public people {

public:

int classnumber;

student(string name = "1", int number = 1, string gender = "1", int id = 1, int classnumber = 1, date birthday = date())

: people(name, number, gender, id, birthday), classnumber(classnumber) {}

};

// 定义教师类，继承人员类

class teacher : public people {

public:

string department;

teacher(string name = "2", int number = 2, string gender = "2", int id = 2, string department = "2", date birthday = date())

: people(name, number, gender, id, birthday), department(department) {}

};

// 按 number 属性对人员对象数组进行排序的函数

void sortPeopleArray(people\* peopleArray[], int size) {

sort(peopleArray, peopleArray + size, [](people\* p1, people\* p2) {

return p1->number < p2->number;

});

}

// 按 number 属性查找人员对象的函数

people\* searchPeopleByNumber(people\* peopleArray[], int size, int number) {

for (int i = 0; i < size; ++i) {

if (peopleArray[i]->number == number) {

return peopleArray[i];

}

}

return nullptr;

}

int main() {

// 创建教师数组

teacher\* teachers[] = {

new teacher("A", 3, "Female", 1001, "Math", date(1980, 5, 1)),

new teacher("B", 1, "Male", 1002, "Science", date(1975, 8, 23)),

new teacher("C", 2, "Male", 1003, "History", date(1982, 11, 15))

};

const int teachersSize = sizeof(teachers) / sizeof(teachers[0]);

// 创建学生数组

student\* students[] = {

new student("D", 6, "Male", 2001, 101, date(2000, 2, 14)),

new student("E", 5, "Female", 2002, 102, date(2001, 7, 30)),

new student("F", 4, "Male", 2003, 103, date(1999, 12, 5))

};

const int studentsSize = sizeof(students) / sizeof(students[0]);

// 按 number 对教师和学生进行排序

sortPeopleArray(reinterpret\_cast<people\*\*>(teachers), teachersSize);

sortPeopleArray(reinterpret\_cast<people\*\*>(students), studentsSize);

// 显示排序后的教师

cout << "排序后的教师:" << endl;

for (int i = 0; i < teachersSize; ++i) {

cout << teachers[i]->name << " (number: " << teachers[i]->number << ", birthday: " << teachers[i]->birthday.year << "-" << teachers[i]->birthday.month << "-" << teachers[i]->birthday.day << ")" << endl;

}

// 显示排序后的学生

cout << "排序后的学生:" << endl;

for (int i = 0; i < studentsSize; ++i) {

cout << students[i]->name << " (number: " << students[i]->number << ", birthday: " << students[i]->birthday.year << "-" << students[i]->birthday.month << "-" << students[i]->birthday.day << ")" << endl;

}

// 按 number 查找教师和学生

cout << "按 number 查找教师:";

int searchNumber1;

cin >> searchNumber1;

people\* foundTeacher = searchPeopleByNumber(reinterpret\_cast<people\*\*>(teachers), teachersSize, searchNumber1);

if (foundTeacher) {

cout << "找到教师: " << foundTeacher->name << " (number: " << foundTeacher->number << ")" << endl;

}

else {

cout << "未找到编号为 " << searchNumber1 << " 的教师" << endl;

}

cout << "按 number 查找学生:";

int searchNumber2;

cin >> searchNumber2;

people\* foundStudent = searchPeopleByNumber(reinterpret\_cast<people\*\*>(students), studentsSize, searchNumber2);

if (foundStudent) {

cout << "找到学生: " << foundStudent->name << " (number: " << foundStudent->number << ")" << endl;

}

else {

cout << "未找到编号为 " << searchNumber2 << " 的学生" << endl;

}

// 清理动态分配的内存

for (int i = 0; i < teachersSize; ++i) {

delete teachers[i];

}

for (int i = 0; i < studentsSize; ++i) {

delete students[i];

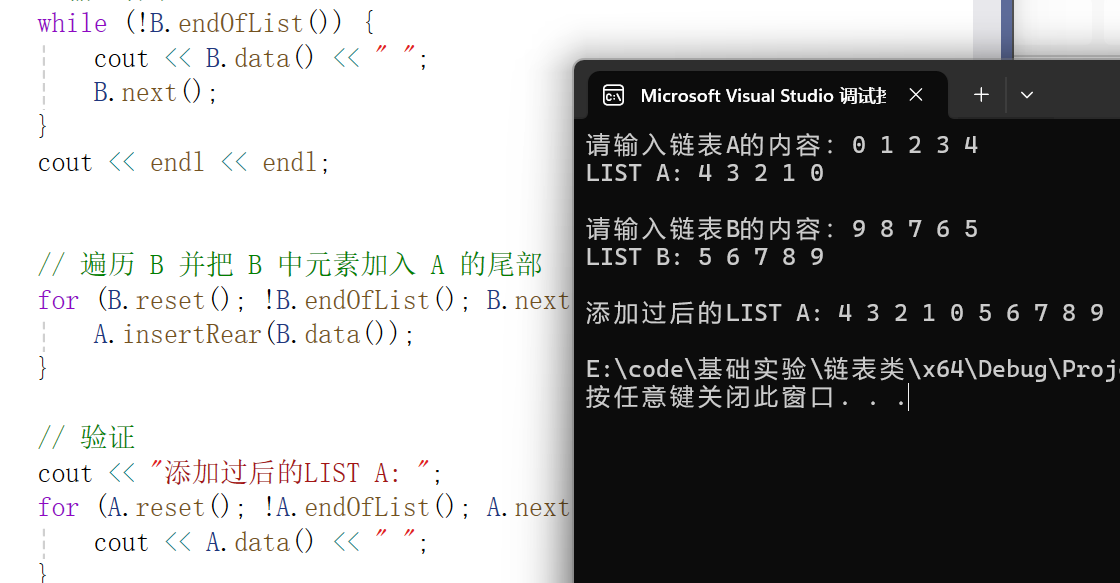
}

return 0;

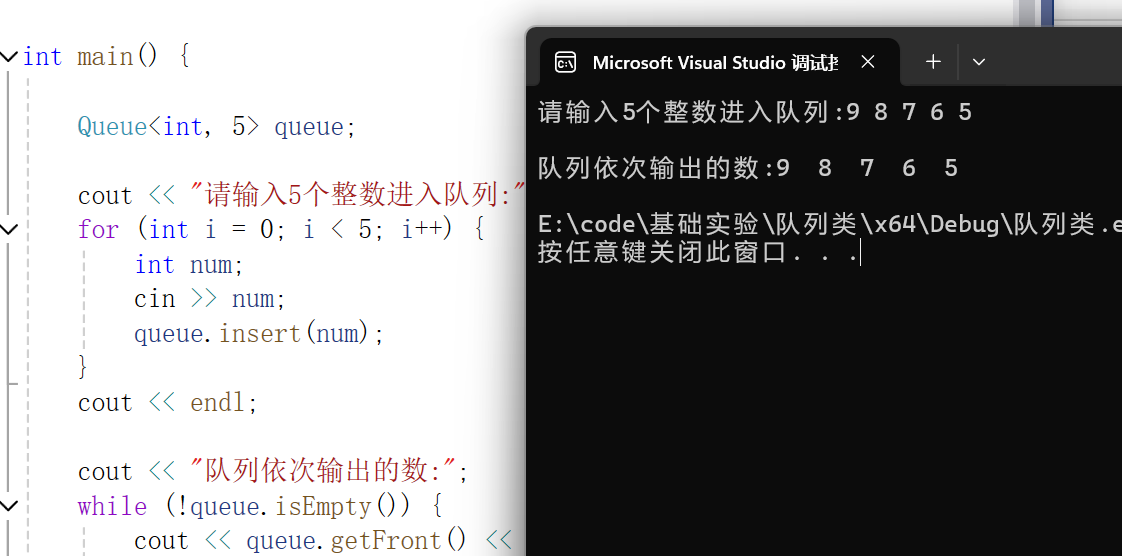
}

**五、验证与结论**

**1.链表类**



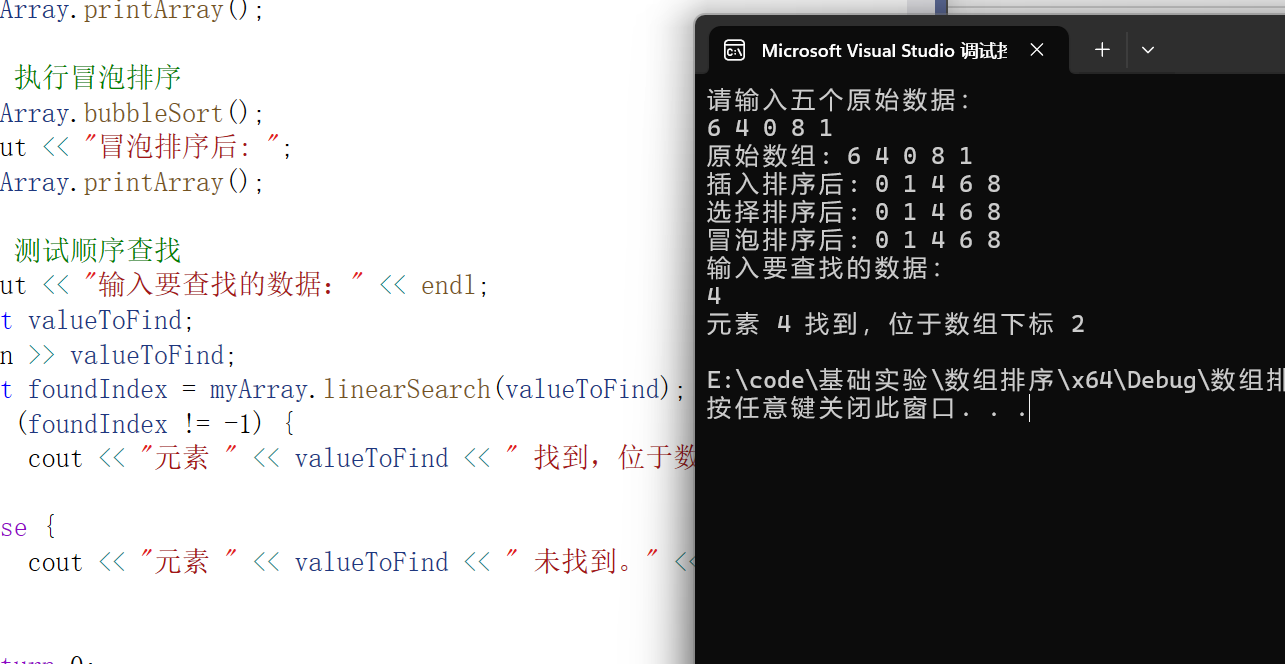
**2.队列类和栈类**



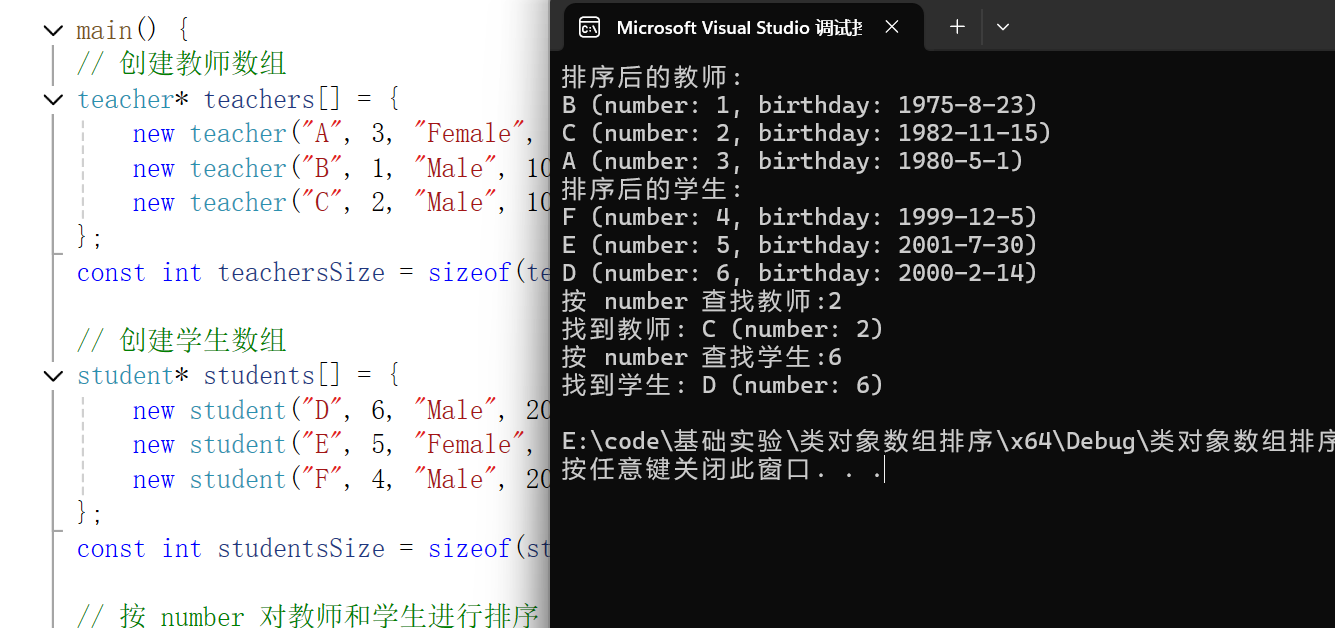
文本

描述已自动生成

**3.数组排序**



**4.类对象数组排序**



**六、总结与心得体会**

经过上述4个基础实验，我深刻理解并尝试使用面向对象的编程语言分别对链表类，队列类，栈类的模版的编写，以及如何实现对一个类的封装并提供外部接口以实现提高代码的安全性和复用率的功能，更加深入地理解一些基础的计算机科学基础知识和算法原理。在栈类中，我明白了这种线性数据结构的特性，体会到出栈入栈这种操作的独特性，以及其在某些特定场合的重要作用；在链表类中，每个节点之间的相关性，指向关系都是明确的，通过对节点的动态分配和释放，处理头指针和尾指针以及节点的插入和删除操作，更好地理解了这种数据结构的逻辑关系和原理；而队列类与栈类有相似之处的同时也有着显著的区别，通过数组实现的队列需要处理循环数组的问题，在广度优先，任务调度等方面也有着广泛应用。在不同情境下，选择合适且恰当的数据结构会极大的提高效率和简化代码。除此之外，在排序算法，以及实现对类的数组的排序时，通过编写，使得熟悉了各种排序算法的原理及实现过程，对数据的操作和处理会更加熟练。总之，在整个实验过程中，每一种算法的实现都让我更进一步的深入培养算法思维和编程能力，理解各种数据结构的核心原理。