1.一家小公司的所有员工都可以分为“销售员”和“技术员”。

销售人员工资=基本工资+销售额\*提成率。

技术员工资=基本工资+工作时间\*每小时报酬。

设计一个抽象类(employee)和两个具体类(Salesman和Technician)来定义所有员工。

定义以下函数来实现多态:

1)函数salary():计算员工的工资。

2)函数display():显示员工的所有信息，如(如果有)

id (int)， name (string)， basesalary (double)， workinghours(int)， paymentph(double)， amountofsales (int)， commissionRate(double∈(0,1))

#include <iostream>

#include <string>

using namespace std;

// 抽象类 Employee

class Employee {

public:

Employee(int id, string name, double baseSalary) : id(id), name(name), baseSalary(baseSalary) {};

virtual double salary() = 0;

virtual void display() {

cout << "ID: " << id << ", Name: " << name << ", Base Salary: " << baseSalary << endl;

};

protected:

int id;

string name;

double baseSalary;

};

// Salesman 类（继承自 Employee）

class Salesman : public Employee {

public:

Salesman(int id, string name, double baseSalary, int amountOfSales, double commissionRate)

: Employee(id, name, baseSalary), amountOfSales(amountOfSales), commissionRate(commissionRate) {};

double salary() override {

return baseSalary + amountOfSales \* commissionRate;

}

void display() override {

Employee::display();

cout << "Amount of Sales: " << amountOfSales << ", Commission Rate: " << commissionRate << endl;

}

private:

int amountOfSales;

double commissionRate;

};

// Technician 类（继承自 Employee）

class Technician : public Employee {

public:

Technician(int id, string name, double baseSalary, int workingHours, double paymentPerHour)

: Employee(id, name, baseSalary), workingHours(workingHours), paymentPerHour(paymentPerHour) {};

double salary() override {

return baseSalary + workingHours \* paymentPerHour;

}

void display() override {

Employee::display();

cout << "Working Hours: " << workingHours << ", Payment Per Hour: " << paymentPerHour << endl;

}

private:

int workingHours;

double paymentPerHour;

};

int main() {

// 创建对象并进行测试

Employee\* employee1 = new Salesman(101, "John Smith", 5000, 10000, 0.05);

Employee\* employee2 = new Technician(102, "Mary Johnson", 4000, 160, 25);

employee1->display();

cout << "Salary: " << employee1->salary() << endl;

employee2->display();

cout << "Salary: " << employee2->salary() << endl;

// 释放内存

delete employee1;

delete employee2;

return 0;

}

定义一个名为Matrix的类模板。模板可以实例化任何元素类型的矩阵。以函数模板的形式重载几个基本操作符，包括:

1)重载>>和<<操作符:分别使能矩阵的输入和输出。

2)重载+、-、\*和=运算符:实现两个矩阵的加、减、乘和赋值。

实现这两个类定义，并使以下名为" main "的函数能够正确输出。

注意:

a) main函数应该演示int元素矩阵(类matrix <int>)和double元素矩阵(类matrix <double>)的实例化。

b)在类定义中使用动态内存分配技术。

#include <iostream>

using namespace std;

// 矩阵模板类

template<class T>

class Matrix {

public:

// 构造函数，参数为矩阵行数和列数

Matrix(int rows, int columns) : rows(rows), columns(columns) {

data = new T\*[rows];

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

data[i] = new T[columns]{0}; } }

// 拷贝构造函数

Matrix(const Matrix& other) : rows(other.rows), columns(other.columns) {

data = new T\*[rows];

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

data[i] = new T[columns];

for (int j = 0; j < columns; j++) {

data[i][j] = other.data[i][j]; } }}

// 析构函数

~Matrix() {

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

delete[] data[i];

}

delete[] data; }

// 重载 << 运算符

friend ostream& operator<<(ostream& os, const Matrix& matrix) {

for (int i = 0; i < matrix.rows; i++) {

for (int j = 0; j < matrix.columns; j++) {

os << matrix.data[i][j] << " "; }

os << endl; }

return os; }

// 重载 >> 运算符

friend istream& operator>>(istream& is, Matrix& matrix) {

for (int i = 0; i < matrix.rows; i++) {

for (int j = 0; j < matrix.columns; j++) {

is >> matrix.data[i][j];

} }

return is; }

// 重载 + 运算符

Matrix operator+(const Matrix& other) const {

if (rows != other.rows || columns != other.columns) {

cerr << "Error: two matrices to add must have the same size." << endl;

exit(1); }

Matrix result(rows, columns);

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

for (int j = 0; j < columns; j++) {

result.data[i][j] = data[i][j] + other.data[i][j]; } } return result; }

// 重载 - 运算符

Matrix operator-(const Matrix& other) const {

if (rows != other.rows || columns != other.columns) {

cerr << "Error: two matrices to subtract must have the same size." << endl;

exit(1); }

Matrix result(rows, columns);

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

for (int j = 0; j < columns; j++) {

result.data[i][j] = data[i][j] - other.data[i][j]; } } return result; }

// 重载 \* 运算符

Matrix operator\*(const Matrix& other) const {

if (columns != other.rows) {

cerr << "Error: #columns of left matrix must equal to #rows of right matrix." << endl;

exit(1); }

Matrix result(rows, other.columns);

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

for (int j = 0; j < other.columns; j++) {

for (int k = 0; k < columns; k++) {

result.data[i][j] += data[i][k] \* other.data[k][j]; }} }

return result; }

// 重载 = 运算符

Matrix& operator=(const Matrix& other) {

if (this == &other) {

return \*this; }

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

delete[] data[i]; }

delete[] data;

rows = other.rows;

columns = other.columns;

data = new T\*[rows];

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

data[i] = new T[columns];

for (int j = 0; j < columns; j++) {

data[i][j] = other.data[i][j]; } }

return \*this; }

private:

T\*\* data;

int rows, columns;

};

int main() {

// 测试 int 矩阵

Matrix<int> matrix1(2, 3);

cin >> matrix1;

cout << matrix1;

Matrix<int> matrix2(2, 3);

cin >> matrix2;

cout << matrix2;

Matrix<int> sum = matrix1 + matrix2;

cout << sum;

Matrix<int> difference = matrix1 - matrix2;

cout << difference;

Matrix<int> product = matrix1 \* matrix2;

cout << product;

// 测试 double 矩阵

Matrix<double> matrix3(3, 2);

cin >> matrix3;

cout << matrix3;

Matrix<double> matrix4(2, 3);

cin >> matrix4;

cout << matrix4;

Matrix<double> product2 = matrix3 \* matrix4;

cout << product2;

return 0;

}

设计一个**类模板** MyArray<T>，实现一个动态数组类。该类包含以下成员函数：

构造函数 MyArray(int size)：创建大小为 size 的动态数组。

拷贝构造函数 MyArray(const MyArray& other)：完成动态数组的深度复制。

析构函数 ~MyArray()：释放动态数组占用的内存。

重载运算符 []：支持用下标访问数组元素的操作。

成员函数 getSize()：返回数组的大小。

成员函数 resize(int size)：修改数组的大小，并保留原有数据。

#include <iostream>

#include <cstring>

template<class T>

class MyArray {

public:

// 构造函数

MyArray(int size) : size(size) {

data = new T[size];

}

// 拷贝构造函数

MyArray(const MyArray& other) : size(other.size) {

data = new T[size];

memcpy(data, other.data, sizeof(T) \* size);

}

// 析构函数

~MyArray() {

delete[] data;

}

// 重载运算符 []

T& operator[](int index) {

if (index >= size || index < 0) {

std::cerr << "Index out of range." << std::endl;

exit(1);

}

return data[index];

}

// 返回数组的大小

int getSize() const {

return size;

}

// 修改数组的大小，保留原有数据

void resize(int newSize) {

if (newSize == size) {

return;

}

T\* newData = new T[newSize];

int minSize = size < newSize ? size : newSize;

memcpy(newData, data, sizeof(T) \* minSize);

delete[] data;

data = newData;

size = newSize;

}

private:

T\* data;

int size;

};

int main() {

MyArray<int> arr(3);

arr[0] = 1;

arr[1] = 2;

arr[2] = 3;

for (int i = 0; i < arr.getSize(); i++) {

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

}

std::cout << std::endl;

MyArray<std::string> arr2(2);

arr2[0] = "Hello";

arr2[1] = "World";

for (int i = 0; i < arr2.getSize(); i++) {

std::cout << arr2[i] << " ";

}

std::cout << std::endl;

arr2.resize(3);

arr2[2] = "!";

for (int i = 0; i < arr2.getSize(); i++) {

std::cout << arr2[i] << " ";

}

std::cout << std::endl;

return 0;

}