Inheritance Qno 1

Imagine a publishing company that markets both book and audiocassette versions of its works. Create a class publication that stores the title (a string) and price (type float) of a publication. From this class derive two classes: book, which adds a page count (type int), and tape, which adds a playing time in minutes (type float). Each of these three classes should have a getdata() function to get its data from the user at the keyboard, and a putdata() function to display its data

Answer

#include <iostream>

#include <string>

using namespace std;

class Publication {

protected:

string title;

float price;

public:

void getdata() {

cout << "Enter the title: ";

getline(cin, title);

cout << "Enter the price: ";

cin >> price;

cin.ignore(); // Clear the input buffer

}

void putdata() {

cout << "Title: " << title << endl;

cout << "Price: " << price << endl;

}

};

class Book : public Publication {

private:

int page\_count;

public:

void getdata() {

Publication::getdata();

cout << "Enter the page count: ";

cin >> page\_count;

cin.ignore(); // Clear the input buffer

}

void putdata() {

Publication::putdata();

cout << "Page Count: " << page\_count << endl;

}

};

class Tape : public Publication {

private:

float playing\_time;

public:

void getdata() {

Publication::getdata();

cout << "Enter the playing time (in minutes): ";

cin >> playing\_time;

cin.ignore(); // Clear the input buffer

}

void putdata() {

Publication::putdata();

cout << "Playing Time: " << playing\_time << " minutes" << endl;

}

};

int main() {

Book book;

cout << "Enter book details: " << endl;

book.getdata();

cout << "\nBook details:" << endl;

book.putdata();

cout << endl;

Tape tape;

cout << "Enter tape details: " << endl;

tape.getdata();

cout << "\nTape details:" << endl;

tape.putdata();

return 0;

}

Derive a class called employee2 from the employee class in the EMPLOY program in this chapter. This new class should add a type double data item called compensation, and also an enum type called period to indicate whether the employee is paid hourly, weekly, or monthly. For simplicity you can change the manager, scientist, and laborer classes so they are derived from employee2 instead of employee. However, note that in many circumstances it might be more in the spirit of OOP to create a separate base class called compensation and three new classes manager2, scientist2, and laborer2, and use multiple inheritance to derive these three classes from the original manager, scientist, and laborer classes and from compensation. This way none of the original classes needs to be modified

#include <iostream>

// Base class Employee

class Employee {

protected:

std::string name;

int employeeId;

public:

Employee(const std::string& name, int employeeId)

: name(name), employeeId(employeeId) {}

void displayInfo() const {

std::cout << "Name: " << name << std::endl;

std::cout << "Employee ID: " << employeeId << std::endl;

}

};

// Derived class Employee2

class Employee2 : public Employee {

protected:

double compensation;

enum Period { HOURLY, WEEKLY, MONTHLY } period;

public:

Employee2(const std::string& name, int employeeId, double compensation, Period period)

: Employee(name, employeeId), compensation(compensation), period(period) {}

void displayInfo() const {

Employee::displayInfo(); // Call base class method

std::cout << "Compensation: " << compensation << std::endl;

std::cout << "Period: ";

switch (period) {

case HOURLY:

std::cout << "Hourly";

break;

case WEEKLY:

std::cout << "Weekly";

break;

case MONTHLY:

std::cout << "Monthly";

break;

}

std::cout << std::endl;

}

};

// Separate base class Compensation

class Compensation {

protected:

double compensation;

enum Period { HOURLY, WEEKLY, MONTHLY } period;

public:

Compensation(double compensation, Period period)

: compensation(compensation), period(period) {}

};

// Derived classes Manager2, Scientist2, Laborer2 using multiple inheritance

class Manager2 : public Employee, public Compensation {

public:

Manager2(const std::string& name, int employeeId, double compensation, Period period)

: Employee(name, employeeId), Compensation(compensation, period) {}

};

class Scientist2 : public Employee, public Compensation {

public:

Scientist2(const std::string& name, int employeeId, double compensation, Period period)

: Employee(name, employeeId), Compensation(compensation, period) {}

};

class Laborer2 : public Employee, public Compensation {

public:

Laborer2(const std::string& name, int employeeId, double compensation, Period period)

: Employee(name, employeeId), Compensation(compensation, period) {}

};

int main() {

// Test Employee2 class

Employee2 emp2("John Doe", 12345, 1500.0, Employee2::WEEKLY);

emp2.displayInfo();

std::cout << std::endl;

// Test Manager2 class

Manager2 mgr2("Jane Smith", 67890, 5000.0, Manager2::MONTHLY);

mgr2.displayInfo();

std::cout << std::endl;

return 0;

}

Chapter Pointers

Q1: Write a program that reads a group of numbers from the user and places them in an array of type float. Once the numbers are stored in the array, the program should average them and print the result. Use pointer notation wherever possible

Answer:

#include <iostream>

int main() {

int size;

std::cout << "Enter the number of elements: ";

std::cin >> size;

float\* numbers = new float[size]; // Dynamically allocate an array

// Read numbers from the user

std::cout << "Enter " << size << " numbers:\n";

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

std::cin >> \*(numbers + i); // Equivalent to numbers[i]

}

// Calculate the average

float sum = 0.0;

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

sum += \*(numbers + i); // Equivalent to numbers[i]

}

float average = sum / size;

// Print the average

std::cout << "Average: " << average << std::endl;

delete[] numbers; // Deallocate the dynamically allocated array

return 0;

}

In this program, we use pointer notation to access array elements. Instead of using array indexing (**numbers[i]**), we use pointer arithmetic (**\*(numbers + i)**). The program prompts the user to enter the number of elements, dynamically allocates an array of floats, reads the numbers from the user, calculates the average using pointer notation, and prints the result. Finally, the dynamically allocated array is deallocated using **delete[]**.

Qno 2:

Create a class that allows you to treat the 10 separate arrays in Exercise 10 as a single one-dimensional array, using array notation with a single index. That is, statements in main() can access their elements using expressions like a[j], even though the class member functions must access the data using the two-step approach. Overload the subscript operator [] (see Chapter 9, “Inheritance”) to achieve this result. Fill the arrays with test data and then display it. Although array notation is used in the class interface in main() to access “array” elements, you should use only pointer notation for all the operations in the implementation (within the class member functions).

Answer :  
#include <iostream>

const int SIZE = 10; // Size of each array

class ArrayWrapper {

private:

int\* data[SIZE]; // Array of pointers to integers

public:

ArrayWrapper() {

// Allocate memory for each array

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

data[i] = new int[SIZE];

}

}

~ArrayWrapper() {

// Deallocate memory for each array

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

delete[] data[i];

}

}

int\* operator[](int index) {

return data[index];

}

};

int main() {

ArrayWrapper arr;

// Fill the arrays with test data

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

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

arr[i][j] = i \* SIZE + j; // Equivalent to \*(arr[i] + j)

}

}

// Display the arrays using array notation

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

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

std::cout << arr[i][j] << ' '; // Equivalent to \*(arr[i] + j)

}

std::cout << std::endl;

}

return 0;

}

In this implementation, the ArrayWrapper class manages an array of pointers to integers. The constructor allocates memory for each array using new, and the destructor deallocates the memory using delete[]. The operator[] is overloaded to return the pointer to the specified array, allowing you to use array notation (arr[i][j]) to access elements.

In the main() function, the arrays are filled with test data, and then the content of the arrays is displayed using array notation. Note that all operations within the class member functions (operator[], constructor, destructor) use pointer notation (data[i], \*(arr[i] + j)) as required.

Chapter : virtual functions

Q1:

In the Distance class, as shown in the FRENGL and FRISQ examples in this chapter, create an overloaded \* operator so that two distances can be multiplied together. Make it a friend function so that you can use such expressions as Wdist1 = 7.5 \* dist2; You’ll need a one-argument constructor to convert floating-point values into Distance values. Write a main() program to test this operator in several ways

Answer:

#include <iostream>

class Distance {

private:

int feet;

float inches;

public:

Distance(int ft = 0, float in = 0) : feet(ft), inches(in) {}

void display() const {

std::cout << feet << " feet, " << inches << " inches" << std::endl;

}

friend Distance operator\*(float value, const Distance& dist);

};

Distance operator\*(float value, const Distance& dist) {

int newFeet = value \* dist.feet;

float newInches = value \* dist.inches;

// Convert excess inches to feet

if (newInches >= 12.0) {

newFeet += static\_cast<int>(newInches / 12.0);

newInches = std::fmod(newInches, 12.0);

}

return Distance(newFeet, newInches);

}

int main() {

Distance dist1(5, 6.5);

Distance dist2(3, 10.2);

std::cout << "Distance 1: ";

dist1.display();

std::cout << "Distance 2: ";

dist2.display();

Distance dist3 = 7.5 \* dist2;

std::cout << "Distance 3: ";

dist3.display();

return 0;

}

In this implementation, the **Distance** class has a one-argument constructor to convert floating-point values into Distance values. The **display()** member function is used to print the Distance.

The **operator\*** is declared as a friend function of the **Distance** class to allow multiplication of a Distance object by a floating-point value. The friend function performs the multiplication operation, taking care of the conversion of excess inches to feet if necessary. The resulting Distance object is returned.

In the **main()** function, two Distance objects (**dist1** and **dist2**) are created and displayed. Then, the **\*** operator is tested by multiplying **dist2** by 7.5 and storing the result in **dist3**. Finally, **dist3** is displayed.

The program demonstrates using the overloaded **\*** operator with expressions like **7.5 \* dist2** where the floating-point value is on the left side of the operator.

Q2:

Start with the Distance class from the ENGLCON example in Chapter 6, “Objects and Classes.” Using a loop similar to that in the DISKFUN example in this chapter, get a number of Distance values from the user, and write them to a disk file. Append them to existing values in the file, if any. When the user signals that no more values will be input, read the file and display all the values

Answer:

#include <iostream>

#include <fstream>

class Distance {

private:

int feet;

float inches;

public:

Distance(int ft = 0, float in = 0) : feet(ft), inches(in) {}

void display() const {

std::cout << feet << " feet, " << inches << " inches" << std::endl;

}

friend std::ostream& operator<<(std::ostream& os, const Distance& dist);

friend std::istream& operator>>(std::istream& is, Distance& dist);

};

std::ostream& operator<<(std::ostream& os, const Distance& dist) {

os << dist.feet << ' ' << dist.inches;

return os;

}

std::istream& operator>>(std::istream& is, Distance& dist) {

is >> dist.feet >> dist.inches;

return is;

}

int main() {

std::ofstream outFile("distance.txt", std::ios::app);

if (!outFile) {

std::cerr << "Error opening file!" << std::endl;

return 1;

}

char choice;

Distance dist;

do {

std::cout << "Enter distance (feet inches): ";

std::cin >> dist;

outFile << dist << std::endl;

std::cout << "Do you want to enter more distances? (y/n): ";

std::cin >> choice;

} while (choice == 'y' || choice == 'Y');

outFile.close();

std::ifstream inFile("distance.txt");

if (!inFile) {

std::cerr << "Error opening file!" << std::endl;

return 1;

}

Distance readDist;

std::cout << "\nDistances from file:" << std::endl;

while (inFile >> readDist) {

readDist.display();

}

inFile.close();

return 0;

}

In this program, the Distance class is modified to add operator<< and operator>> to allow the Distance objects to be output to and read from streams, respectively. These operators are friend functions of the Distance class.

The program then prompts the user to enter Distance values (in feet and inches) and writes them to a file named "distance.txt". The user can enter multiple distances, and each distance is written on a new line in the file.

After the user signals that no more values will be input, the program opens the file for reading, reads each Distance value from the file using operator>>, and displays them using the display() member function.

Note that the file is opened in append mode (std::ios::app) to append new distances to any existing values in the file.