//----------------------------------------------------------------------------//

// Name: Chris Kruki //

// Student ID: 1132759 //

// Assignment: #3 //

//----------------------------------------------------------------------------//

#define PARTINFO\_CNT 15

#include <fstream>

#include <iomanip>

#include <iostream>

#include <string>

using namespace std;

// Car class definition

class Car

{

public:

// Constructor prototypes

Car();

Car(const string& fBrand, const string& fModel, const int& fYear);

// Setter and Getter prototypes

void setBrand(const string& fBrand);

void setModel(const string& fModel);

void setYear(const int& fYear);

string getBrand()const;

string getModel()const;

int getYear()const;

protected:

string brand, model;

int year;

};

// Car Constructor definitions

Car::Car() {

brand = "";

model = "";

year = 0;

}

Car::Car(const string& fBrand, const string& fModel, const int& fYear) {

brand = fBrand;

model = fModel;

year = fYear;

}

// Car Getter and Setter definitions

void Car::setBrand(const string& fBrand) {

brand = fBrand;

}

void Car::setModel(const string& fModel) {

model = fModel;

}

void Car::setYear(const int& fYear) {

year = fYear;

}

string Car::getBrand()const {

return brand;

}

string Car::getModel()const {

return model;

}

int Car::getYear()const {

return year;

}

// Manufacturer class definition

class Manufacturer

{

public:

// Constructor prototypes

Manufacturer();

Manufacturer(const string& fManufacturer);

// Getter and Setter prototypes

void setManufacturer(const string& fManufacturer);

string getManufacturer()const;

protected:

string manufacturer;

};

// Manufacturer Constructor definitions

Manufacturer::Manufacturer() {

manufacturer = "";

}

Manufacturer::Manufacturer(const string& fManufacturer) {

manufacturer = fManufacturer;

}

// Manufacturer Getter and Setter definitions

void Manufacturer::setManufacturer(const string& fManufacturer) {

manufacturer = fManufacturer;

}

string Manufacturer::getManufacturer()const {

return manufacturer;

}

// Parts class definition - contains all common attributes of all parts

class Parts

{

public:

// Manufacturer object delclaration - has a manufacturer

Manufacturer manufacturer;

// Constructor prototypes

Parts();

Parts(const string& fDescription, const float& fPrice, const string& fManufacturer, const int& fQtySold);

// Getter and Setter prototypes

void setDescription(const string& fDescription);

void setPrice(const float& fPrice);

void setQtySold(const int& fQtySold);

string getDescription()const;

float getPrice()const;

int getQtySold()const;

float getRevenue()const;

string getManufacturer()const;

protected:

string description;

float price;

int qtySold;

float revenue;

};

// Parts Constructor defintions

Parts::Parts() : description(""), price(0.0f), qtySold(0), manufacturer(), revenue(0.0f) {}

Parts::Parts(const string& fDescription, const float& fPrice, const string& fManufacturer, const int& fQtySold) :

description(fDescription), price(fPrice), manufacturer(fManufacturer), qtySold(fQtySold), revenue(fPrice \* fQtySold) {}

// Parts Setter and Getter definitions

void Parts::setDescription(const string& fDescription) {

description = fDescription;

}

void Parts::setPrice(const float& fPrice) {

price = fPrice;

}

void Parts::setQtySold(const int& fQtySold) {

qtySold = fQtySold;

}

string Parts::getDescription()const {

return description;

}

float Parts::getPrice()const {

return price;

}

int Parts::getQtySold()const {

return qtySold;

}

float Parts::getRevenue()const {

return revenue;

}

string Parts::getManufacturer()const {

return manufacturer.getManufacturer();

}

// Brakes class definition - is a Part, has a car

class Brakes : public Parts

{

public:

// Has a car - car object declaration

Car car;

// Constructor Prototypes

Brakes();

Brakes(const string& fDescription, const float& fPrice, const string& fManufacturer, const int& fQtySold, const string& fBrand, const string& fModel, const int& fYear, const string fMaterial);

// Getter and setter prototypes

void setMaterial(const string& fMaterial);

string getMaterial()const;

string getCategory()const;

string getBrand()const;

string getModel()const;

int getYear()const;

private:

string material, category;

};

// Constructor definitions

Brakes::Brakes() : Parts(), car(), material(""), category("Brakes") {}

Brakes::Brakes(const string& fDescription, const float& fPrice, const string& fManufacturer, const int& fQtySold, const string& fBrand, const string& fModel, const int& fYear, const string fMaterial) :

Parts(fDescription, fPrice, fManufacturer, fQtySold), car(fBrand, fModel, fYear), material(fMaterial), category("Brakes") {}

// Getter and setter prototypes

void Brakes::setMaterial(const string& fMaterial) {

material = fMaterial;

}

string Brakes::getMaterial()const {

return material;

}

string Brakes::getCategory()const {

return category;

}

string Brakes::getBrand()const {

return car.getBrand();

}

string Brakes::getModel()const {

return car.getModel();

}

int Brakes::getYear()const {

return car.getYear();

}

// Lights class definition - is a Part, has a car

class Lights : public Parts

{

public:

// Has a car - car class declaration

Car car;

// Constructor Prototypes

Lights();

Lights(const string& fDescription, const float& fPrice, const string& fManufacturer, const int& fQtySold, const string& fBrand, const string& fModel, const int& fYear, const int& fWatts);

// Getter and setter prototypes

void setWatts(const int& fWatts);

int getWatts()const;

string getCategory()const;

string getBrand()const;

string getModel()const;

int getYear()const;

private:

int watts;

string category;

};

// Constructor definitions

Lights::Lights() : Parts(), car(), watts(0), category("Lights") {}

Lights::Lights(const string& fDescription, const float& fPrice, const string& fManufacturer, const int& fQtySold, const string& fBrand, const string& fModel, const int& fYear, const int& fWatts) :

Parts(fDescription, fPrice, fManufacturer, fQtySold), car(fBrand, fModel, fYear), watts(fWatts), category("Lights") {}

// Getter and Setter definitions

void Lights::setWatts(const int& fWatts) {

watts = fWatts;

}

int Lights::getWatts()const {

return watts;

}

string Lights:: getCategory()const {

return category;

}

string Lights::getBrand()const {

return car.getBrand();

}

string Lights::getModel()const {

return car.getModel();

}

int Lights::getYear()const {

return car.getYear();

}

// Oil class definition - is a Part

class Oil : public Parts{

public:

// Constructor Prototypes

Oil();

Oil(const string& fDescription, const float& fPrice, const string& fManufacturer, const int& fQtySold, const string& fWeight, const string& fType, const int& fQuarts);

// Getter and Setter prototypes

void setWeight(const string& fWeight);

void setType(const string& fType);

void setQuarts(const int& fQuarts);

string getWeight()const;

string getType()const;

int getQuarts()const;

string getCategory()const;

private:

string weight, type, category;

int quarts;

};

// Oil constructor definitions

Oil::Oil() : Parts(), weight(""), type(""), quarts(0), category("Oil") {}

Oil::Oil(const string& fDescription, const float& fPrice, const string& fManufacturer, const int& fQtySold, const string& fWeight, const string& fType, const int& fQuarts) :

Parts(fDescription, fPrice, fManufacturer, fQtySold), weight(fWeight), type(fType), quarts(fQuarts), category("Oil") {}

// Getter and Setter Prototypes

void Oil::setWeight(const string& fWeight) {

weight = fWeight;

}

void Oil::setType(const string& fType) {

type = fType;

}

void Oil::setQuarts(const int& fQuarts) {

quarts = fQuarts;

}

string Oil::getWeight()const {

return weight;

}

string Oil::getType()const {

return type;

}

int Oil::getQuarts()const {

return quarts;

}

string Oil::getCategory()const {

return category;

}

// Tires class definition - is a Part

class Tires : public Parts

{

public:

// Constructor Prototypes

Tires();

Tires(const string& fDescription, const float& fPrice, const string& fManufacturer, const int& fQtySold, const string& fSize, const int& fWarranty);

// Getter and Setter Prototypes

void setSize(const string& fSize);

void setWarranty(const int& fWarranty);

string getSize()const;

int getWarranty()const;

string getCategory()const;

private:

string size, category;

int warranty;

};

// Tires Constructor definitions

Tires::Tires() : Parts(), size(""), warranty(0), category("Tires") {}

Tires::Tires(const string& fDescription, const float& fPrice, const string& fManufacturer, const int& fQtySold, const string& fSize, const int& fWarranty) :

Parts(fDescription, fPrice, fManufacturer, fQtySold), size(fSize), warranty(fWarranty), category("Tires") {}

// Tires Getter and Setter definitions

void Tires::setSize(const string& fSize) {

size = fSize;

}

void Tires::setWarranty(const int& fWarranty) {

warranty = fWarranty;

}

string Tires::getSize()const {

return size;

}

int Tires::getWarranty()const {

return warranty;

}

string Tires::getCategory()const {

return category;

}

// Global Function Prototypes

void parseLineToTokens(string lineText, string tokens[]);

int fileLineCount(ifstream& fileIn);

void objectCount(ifstream& fileIn, int& brakeCount, int& lightCount, int& oilCount, int& tireCount);

void objectStore(ifstream& fileIn, Brakes\* brakeArray, Lights\* lightArray, Oil\* oilArray, Tires\* tireArray);

void sortArrays(Brakes\* brakeArray, Lights\* lightArray, Oil\* oilArray, Tires\* tireArray, const int& brakeCount, const int& lightCount, const int& oilCount, const int& tireCount);

void outputIndividualBest(ofstream& fileOut, Brakes\* brakeArray, Lights\* lightArray, Oil\* oilArray, Tires\* tireArray);

void outputTotalBest(ofstream& fileOut, Brakes\* brakeArray, Lights\* lightArray, Oil\* oilArray, Tires\* tireArray, const int& brakeCount, const int& lightCount, const int& oilCount, const int& tireCount);

int main()

{

// Open the file from which to read the data

ifstream fileIn("Parts\_List.txt");

int brakeCount = 0, lightCount = 0, oilCount = 0, tireCount = 0;

// Call a global function to find out how many objects of each type to create

if (fileIn.is\_open()) {

cout << "File found." << endl;

objectCount(fileIn, brakeCount, lightCount, oilCount, tireCount);

}

else {

cerr << "File not found.";

exit(1);

}

// Create arrays to contain the necessary objects

Brakes\* brakeArray = new Brakes[brakeCount];

Lights\* lightArray = new Lights[lightCount];

Oil\* oilArray = new Oil[oilCount];

Tires\* tireArray = new Tires[tireCount];

// Global function to read information from the file into the arrays of objects

objectStore(fileIn, brakeArray, lightArray, oilArray, tireArray);

fileIn.close(); // Close file input stream

// Function to sort arrays from most to least revenue

sortArrays(brakeArray, lightArray, oilArray, tireArray, brakeCount, lightCount, oilCount, tireCount);

// Output Functions

ofstream fileIndividualOut("Best\_Individual\_Parts.txt");

ofstream fileTotalOut("Best\_Total\_Parts.txt");

outputIndividualBest(fileIndividualOut, brakeArray, lightArray, oilArray, tireArray);

outputTotalBest(fileTotalOut, brakeArray, lightArray, oilArray, tireArray, brakeCount, lightCount, oilCount, tireCount);

cout << "Files written." << endl;

fileIndividualOut.close(); // Close output files

fileTotalOut.close();

// Delete dynamic arrays

delete[] brakeArray;

delete[] lightArray;

delete[] oilArray;

delete[] tireArray;

}

// Parse a line of text into tokens and store them in an array of strings

void parseLineToTokens(string lineText, string tokens[]) // get line of text from file, then pass it into this function

{

int end, start;

start = -2;

for (int j = 0; j < PARTINFO\_CNT; j++)

{

start = start + 3;

end = lineText.find('"', start);

tokens[j] = lineText.substr(start, end - start);

start = end;

}

// token array will have all of the values sorted,

// 15 different items ( columns of things )

}

// Counts number of total lines in File and returns that number to caller

int fileLineCount(ifstream& fileIn) {

int lineCount = 0;

string trash = "";

while (fileIn.good()) {

getline(fileIn, trash); // Read through a line and store in trash

lineCount++; // Increase line count

}

fileIn.clear(); // Reset EOF bit

fileIn.seekg(ios::beg); // Set file pointer to beginning

return lineCount; // Return line count

}

// Counts the number of individual objects in the file and sets the parameters to that number respectively

void objectCount(ifstream& fileIn, int& brakeCount, int& lightCount, int& oilCount, int& tireCount) {

int lineCount = fileLineCount(fileIn); //

string tempLine, tempTokens[PARTINFO\_CNT];

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

getline(fileIn, tempLine); // Get line and store in tempLine var

parseLineToTokens(tempLine, tempTokens); // Parse the tempLine into tempTokens

// If statements to check the tempLine's first token (category) and add to the appropriate item count

if (tempTokens[0] == "Brakes") {

brakeCount++;

}

else if (tempTokens[0] == "Lights") {

lightCount++;

}

else if (tempTokens[0] == "Oil") {

oilCount++;

}

else if (tempTokens[0] == "Tires") {

tireCount++;

}

}

fileIn.clear(); // Reset EOF bit

fileIn.seekg(ios::beg); // Set file pointer to beginning

// Debug

//cout << "Brake count: " << brakeCount << endl;

//cout << "Light count: " << lightCount << endl;

//cout << "Oil count: " << oilCount << endl;

//cout << "Tire count: " << tireCount << endl;

}

// Reads the file and using similar techniques in objectCount, instantiates objects of appropriate type and stores them in the passed item arrays

void objectStore(ifstream& fileIn, Brakes\* brakeArray, Lights\* lightArray, Oil\* oilArray, Tires\* tireArray) {

int lineCount = fileLineCount(fileIn);

int brakeCount = 0, lightCount = 0, oilCount = 0, tireCount = 0;

string tempLine, tempTokens[PARTINFO\_CNT];

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

getline(fileIn, tempLine); // Get line and store in tempLine var

parseLineToTokens(tempLine, tempTokens); // Parse the tempLine into tempTokens

// If statements to check the tempLine's first token (category) and populates appropriate item array and # with item instance

if (tempTokens[0] == "Brakes") {

brakeArray[brakeCount] = Brakes(tempTokens[1], stof(tempTokens[2]), tempTokens[3], stoi(tempTokens[4]), tempTokens[5], tempTokens[6], stoi(tempTokens[7]), tempTokens[8]);

brakeCount++;

}

else if (tempTokens[0] == "Lights") {

lightArray[lightCount] = Lights(tempTokens[1], stof(tempTokens[2]), tempTokens[3], stoi(tempTokens[4]), tempTokens[5], tempTokens[6], stoi(tempTokens[7]), stoi(tempTokens[9]));

lightCount++;

}

else if (tempTokens[0] == "Oil") {

oilArray[oilCount] = Oil(tempTokens[1], stof(tempTokens[2]), tempTokens[3], stoi(tempTokens[4]), tempTokens[10], tempTokens[11], stoi(tempTokens[12]));

oilCount++;

}

else if (tempTokens[0] == "Tires") {

tireArray[tireCount] = Tires(tempTokens[1], stof(tempTokens[2]), tempTokens[3], stoi(tempTokens[4]), tempTokens[13], stoi(tempTokens[14]));

tireCount++;

}

}

fileIn.clear(); // Reset EOF bit

fileIn.seekg(ios::beg); // Set file pointer to beginning

}

// Sorts arrays from most revenue to least revenue

void sortArrays(Brakes\* brakeArray, Lights\* lightArray, Oil\* oilArray, Tires\* tireArray, const int& brakeCount, const int& lightCount, const int& oilCount, const int& tireCount) {

// Brake Array Sort

int i, j, largestIndex;

float largest;

for (i = 0; i < brakeCount; i++) { // Loop through all revenues in Array

largest = brakeArray[i].getRevenue(); // Set first revenue as largest VALUE

largestIndex = i; // Set var to the larges value's INDEX

for (j = i; j < brakeCount; j++) { // Loop through the whole array again - this includes the same iteration

if (brakeArray[j].getRevenue() > largest) { // if the iterated revenue is larger than the current largest, set the value and index to this iteration

largest = brakeArray[j].getRevenue();

largestIndex = j;

}

}

Brakes temp = brakeArray[i]; // Swap the two objects

brakeArray[i] = brakeArray[largestIndex];

brakeArray[largestIndex] = temp;

}

// Light Array Sort

for (i = 0; i < lightCount; i++) { // Loop through all revenues in Array

largest = lightArray[i].getRevenue(); // Set first revenue as largest VALUE

largestIndex = i; // Set var to the largest value's INDEX

for (j = i; j < lightCount; j++) { // Loop through the whole array again - this includes the same iteration

if (lightArray[j].getRevenue() > largest) { // if the iterated revenue is larger than the current largest, set the value and index to this iteration

largest = lightArray[j].getRevenue();

largestIndex = j;

}

}

Lights temp = lightArray[i]; // Swap the two objects

lightArray[i] = lightArray[largestIndex];

lightArray[largestIndex] = temp;

}

// Oil Array Sort

for (i = 0; i < oilCount; i++) { // Loop through all revenues in Array

largest = oilArray[i].getRevenue(); // Set first revenue as largest VALUE

largestIndex = i; // Set var to the largest value's INDEX

for (j = i; j < oilCount; j++) { // Loop through the whole array again - this includes the same iteration

if (oilArray[j].getRevenue() > largest) { // if the iterated revenue is larger than the current largest, set the value and index to this iteration

largest = oilArray[j].getRevenue();

largestIndex = j;

}

}

Oil temp = oilArray[i]; // Swap the two objects

oilArray[i] = oilArray[largestIndex];

oilArray[largestIndex] = temp;

}

// Tires Array Sort

for (i = 0; i < tireCount; i++) { // Loop through all revenues in Array

largest = tireArray[i].getRevenue(); // Set first revenue as largest VALUE

largestIndex = i; // Set var to the largest value's INDEX

for (j = i; j < tireCount; j++) { // Loop through the whole array again - this includes the same iteration

if (tireArray[j].getRevenue() > largest) { // if the iterated revenue is larger than the current largest, set the value and index to this iteration

largest = tireArray[j].getRevenue();

largestIndex = j;

}

}

Tires temp = tireArray[i]; // Swap the two objects

tireArray[i] = tireArray[largestIndex];

tireArray[largestIndex] = temp;

}

}

void outputIndividualBest(ofstream& fileOut, Brakes\* brakeArray, Lights\* lightArray, Oil\* oilArray, Tires\* tireArray) {

// Best Seller of Brakes

fileOut << "\*\* " << brakeArray[0].getCategory() << " \*\*" << endl;

fileOut << "\tDescription: " << brakeArray[0].getDescription() << endl;

fileOut << "\tPrice: " << brakeArray[0].getPrice() << endl;

fileOut << "\tManufacturer: " << brakeArray[0].getManufacturer() << endl;

fileOut << "\tQty Sold: " << brakeArray[0].getQtySold() << endl;

fileOut << "\tCar:" << endl;

fileOut << "\t\tBrand: " << brakeArray[0].getBrand() << endl;

fileOut << "\t\tModel: " << brakeArray[0].getModel() << endl;

fileOut << "\t\tYear: " << brakeArray[0].getYear() << endl;

fileOut << "\tMaterial: " << brakeArray[0].getMaterial() << endl << endl;

// Best Seller of Lights

fileOut << "\*\* " << lightArray[0].getCategory() << " \*\*" << endl;

fileOut << "\tDescription: " << lightArray[0].getDescription() << endl;

fileOut << "\tPrice: " << lightArray[0].getPrice() << endl;

fileOut << "\tManufacturer: " << lightArray[0].getManufacturer() << endl;

fileOut << "\tQty Sold: " << lightArray[0].getQtySold() << endl;

fileOut << "\tCar:" << endl;

fileOut << "\t\tBrand: " << lightArray[0].getBrand() << endl;

fileOut << "\t\tModel: " << lightArray[0].getModel() << endl;

fileOut << "\t\tYear: " << lightArray[0].getYear() << endl;

fileOut << "\tWatts: " << lightArray[0].getWatts() << endl << endl;

// Best Seller of Oil

fileOut << "\*\* " << oilArray[0].getCategory() << " \*\*" << endl;

fileOut << "\tDescription: " << oilArray[0].getDescription() << endl;

fileOut << "\tPrice: " << oilArray[0].getPrice() << endl;

fileOut << "\tManufacturer: " << oilArray[0].getManufacturer() << endl;

fileOut << "\tQty Sold: " << oilArray[0].getQtySold() << endl;

fileOut << "\tWeight: " << oilArray[0].getWeight() << endl;

fileOut << "\tType: " << oilArray[0].getType() << endl;

fileOut << "\tQuarts: " << oilArray[0].getQuarts() << endl << endl;

// Best Seller of Tires

fileOut << "\*\* " << tireArray[0].getCategory() << " \*\*" << endl;

fileOut << "\tDescription: " << tireArray[0].getDescription() << endl;

fileOut << "\tPrice: " << tireArray[0].getPrice() << endl;

fileOut << "\tManufacturer: " << tireArray[0].getManufacturer() << endl;

fileOut << "\tQty Sold: " << tireArray[0].getQtySold() << endl;

fileOut << "\tSize: " << tireArray[0].getSize() << endl;

fileOut << "\tWarranty: " << tireArray[0].getWarranty() << endl << endl;

}

void outputTotalBest(ofstream& fileOut, Brakes\* brakeArray, Lights\* lightArray, Oil\* oilArray, Tires\* tireArray, const int& brakeCount, const int& lightCount, const int& oilCount, const int& tireCount) {

// Brake Revenue Revenues

int i;

fileOut << brakeArray[0].getCategory() << endl;

for (i = 0; i < brakeCount; i++) {

fileOut << "\t" << setw(35) << left << brakeArray[i].getDescription() << " $" << setw(10) << left << setprecision(2) << fixed << brakeArray[i].getRevenue() << endl;

}

// Light Array Revenues

fileOut << lightArray[0].getCategory() << endl;

for (i = 0; i < lightCount; i++) {

fileOut << "\t" << setw(35) << left << lightArray[i].getDescription() << " $" << setw(10) << left << setprecision(2) << fixed << lightArray[i].getRevenue() << endl;

}

// Oil Array Revenues

fileOut << oilArray[0].getCategory() << endl;

for (i = 0; i < oilCount; i++) {

fileOut << "\t" << setw(35) << left << oilArray[i].getDescription() << " $" << setw(10) << left << setprecision(2) << fixed << oilArray[i].getRevenue() << endl;

}

// Tire Array Revenues

fileOut << tireArray[0].getCategory() << endl;

for (i = 0; i < tireCount; i++) {

fileOut << "\t" << setw(35) << left << tireArray[i].getDescription() << " $" << setw(10) << left << setprecision(2) << fixed << tireArray[i].getRevenue() << endl;

}

}

// Notes:

//

// Example:

// To create an array of 'cnt' Books items, where 'cnt' can only be determined at the time the program is run:

// Books \*booksList = new Books[cnt];

//

// To clean up dynamically allocated memory when done with the dynamically allocated array:

// delete [] booksList;

//

//

// To go back and read from the beginning of the file that was already opened and read till the EOF

// bookFile.clear(); // reset the EOF state

// bookFile.seekg(0, ios::beg); // set pointer at the beginning of the file