San Francisco Bay University

CS360 - Programming in C and C++
Homework Assignment #6

Due day: 4/14/2024

Replit link: https://replit.com/@TASMITA-TANJIMT/HW06MAINFILE
GITHUB LINK: https://github.com/tasmita0131/06_mainfile

1. Using classes, design an online address book to keep track of the names, addresses, phone numbers, and dates of birth of family members, close friends, and certain business associates. Your program should be able to handle a maximum of 500 entries.

   a. **Design the class *dateType* was designed to implement the date in a program, but the member function *setDate* and the constructor do not check whether the date is valid before storing the date in the member variables. Rewrite the definitions of the function *setDate* and the constructor so that the values for the month, day, and year are checked before storing the date into the member variables. Add a member function, *isLeapYear*, to check whether a year is a leap year. Moreover, write a test program to test your class.**

CODE:

```
class DateType {
public:
  void setDate(int month, int day, int year);
  bool isLeapYear() const;
  void printDate() const;
  int getMonth() const { return month; }

  DateType(int m = 1, int d = 1, int y = 1900) { setDate(m, d, y); }

private:
  int month, day, year;
  bool isValidDate(int, int, int) const;
};

bool DateType::isLeapYear() const {
  return (year % 4 == 0) && (year % 100 != 0 || year % 400 == 0);
}

void DateType::setDate(int m, int d, int y) {
  if (isValidDate(m, d, y)) {
    month = m;
    day = d;
    year = y;
  } else {
    cout << "Invalid date. Setting to default values." << endl;
    month = 1;
    day = 1;
    year = 1900;
  }
}

bool DateType::isValidDate(int m, int d, int y) const {
  int daysInMonth[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  if (y < 1 || m < 1 || m > 12 || d < 1)
    return false;
  if (isLeapYear())
    daysInMonth[1] = 29; // February in a leap year
  return d <= daysInMonth[m - 1];
}

void DateType::printDate() const {
  cout << month << "-" << day << "-" << year << endl;
}
```

b. Define a class, *addressType*, that can store a street address, city, state, and ZIP code. Use the appropriate functions to print and store the address. Also, use constructors to automatically initialize the member variables.

CODE:

```
class AddressType {
public:
  void setAddress(string str, string c, string st, string z);
  void printAddress() const;

  AddressType(string str = "", string c = "", string st = "", string z = "")
      : street(str), city(c), state(st), zip(z) {}

private:
  string street, city, state, zip;
};

void AddressType::setAddress(string str, string c, string st, string z) {
  street = str;
  city = c;
  state = st;
  zip = z;
}

void AddressType::printAddress() const {
  cout << street << ", " << city << ", " << state << " " << zip << endl;
}
```

c. **Define a class *extPersonType* using the class *personType* as follows, the class *dateType*, and the class *addressType*. Add a member variable to this class to classify the person as a family member, friend, or business associate. Also, add a member variable to store the phone number. Add (or override) the functions to print and store the appropriate information. Use constructors to automatically initialize the member variables.**

CODE:

```
class PersonType {
public:
  void print() const;
  void setName(string f, string l);
  string getFirstName() const { return firstName; }
  string getLastName() const { return lastName; }

  PersonType(string f = "", string l = "") : firstName(f), lastName(l) {}

private:
  string firstName, lastName;
};

void PersonType::print() const { cout << firstName << " " << lastName; }

void PersonType::setName(string f, string l) {
  firstName = f;
  lastName = l;
}

class ExtPersonType : public PersonType {
public:
  void setInfo(string f, string l, string p, string str, string c, string st,
             string z, string t, int m, int d, int y);
  void print() const;
  int getBirthMonth() const { return dob.getMonth(); }
  string getRelationType() const { return relationType; }

  ExtPersonType(string f = "", string l = "", string p = "", string str = "",
              string c = "", string st = "", string z = "", string t = "",
              int m = 1, int d = 1, int y = 1900);

private:
  string phoneNumber;
  AddressType address;
  DateType dob;
  string relationType; // "family", "friend", "business"
};

void ExtPersonType::setInfo(string f, string l, string p, string str, string c,
                        string st, string z, string t, int m, int d,
                        int y) {
  setName(f, l);
  phoneNumber = p;
  address.setAddress(str, c, st, z);
  relationType = t;
  dob.setDate(m, d, y);
}
```

```
void ExtPersonType::print() const {
  PersonType::print();
  cout << ", Phone: " << phoneNumber << ", ";
  address.printAddress();
  cout << "DOB: ";
  dob.printDate();
  cout << ", Type: " << relationType << endl;
}

ExtPersonType::ExtPersonType(string f, string l, string p, string str, string c,
                             string st, string z, string t, int m, int d, int y)
    : PersonType(f, l), phoneNumber(p), address(str, c, st, z), relationType(t),
      dob(m, d, y) {}
```

d. **Define the class *addressBookType* using the previously defined classes. An object of the type *addressBookType* should be able to process a maximum of 500 entries. The program should perform the following operations:**

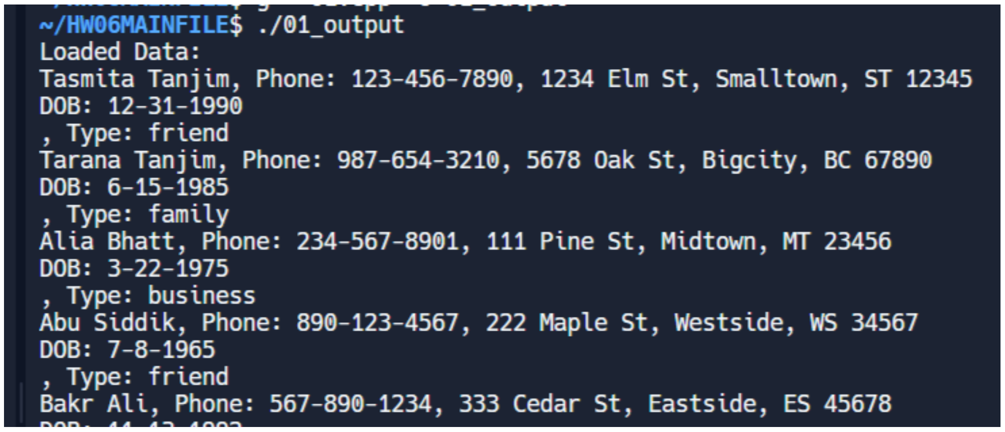
   i. Load the data into the address book from a disk.
   ii. Sort the address book by last name.
   iii. Search for a person by last name.
   iv. Print the address, phone number, and date of birth (if it exists) of a given person.
   v. Print the names of the people whose birthdays are in a given month.
   vi. Print the names of all of the people between two last names.
   vii. Depending on the user's request, print the names of all family members, friends, or business associates.

CODE:

```
class AddressBookType {
public:
  void loadData(const vector<ExtPersonType> &data);
  void sortEntries();
  void searchByLastName(string lastName) const;
  void printPersonInfo(string lastName) const;
  void printBirthdaysByMonth(int month) const;
  void printNamesByType(string type) const;
  void printNamesInRange(string start, string end) const;
  const vector<ExtPersonType> &getEntries() const { return entries; }

  AddressBookType();

private:
  vector<ExtPersonType> entries;
  static bool compareByLastName(const ExtPersonType &a,
                               const ExtPersonType &b) {
    return a.getLastName() < b.getLastName();
  }
};

void AddressBookType::loadData(const vector<ExtPersonType> &data) {
  entries = data;
}

void AddressBookType::sortEntries() {
  sort(entries.begin(), entries.end(), compareByLastName);
}

void AddressBookType::searchByLastName(string lastName) const {
  bool found = false;
  for (const auto &entry : entries) {
    if (entry.getLastName() == lastName) {
      entry.print();
      found = true;
    }
  }
  if (!found) {
    cout << "No entries found for last name: " << lastName << endl;
  }
}
```

```
void AddressBookType::printPersonInfo(string lastName) const {
  bool found = false;
  for (const auto &entry : entries) {
    if (entry.getLastName() == lastName) {
      entry.print();
      found = true;
      break;
    }
  }
  if (!found) {
    cout << "No entries found for last name: " << lastName << endl;
  }
}

void AddressBookType::printBirthdaysByMonth(int month) const {
  bool found = false;
  for (const auto &entry : entries) {
    if (entry.getBirthMonth() == month) {
      entry.print();
      found = true;
    }
  }
  if (!found) {
    cout << "No birthdays found for month: " << month << endl;
  }
}

void AddressBookType::printNamesByType(string type) const {
  bool found = false;
  for (const auto &entry : entries) {
    if (entry.getRelationType() == type) {
      entry.print();
      found = true;
    }
  }
  if (!found) {
    cout << "No entries found for type: " << type << endl;
  }
}

void AddressBookType::printNamesInRange(string start, string end) const {
  bool inRange = false;
  for (const auto &entry : entries) {
    if (entry.getLastName() == start)
      inRange = true;
    if (inRange) {
      entry.print();
    }
    if (entry.getLastName() == end)
      break;
  }
}

AddressBookType::AddressBookType() {}
```

OUTPUT:


```
~/HW06MAINFILE$ ./01_output
Loaded Data:
Tasmita Tanjim, Phone: 123-456-7890, 1234 Elm St, Smalltown, ST 12345
DOB: 12-31-1990
, Type: friend
Tarana Tanjim, Phone: 987-654-3210, 5678 Oak St, Bigcity, BC 67890
DOB: 6-15-1985
, Type: family
Alia Bhatt, Phone: 234-567-8901, 111 Pine St, Midtown, MT 23456
DOB: 3-22-1975
, Type: business
Abu Siddik, Phone: 890-123-4567, 222 Maple St, Westside, WS 34567
DOB: 7-8-1965
, Type: friend
Bakr Ali, Phone: 567-890-1234, 333 Cedar St, Eastside, ES 45678
```

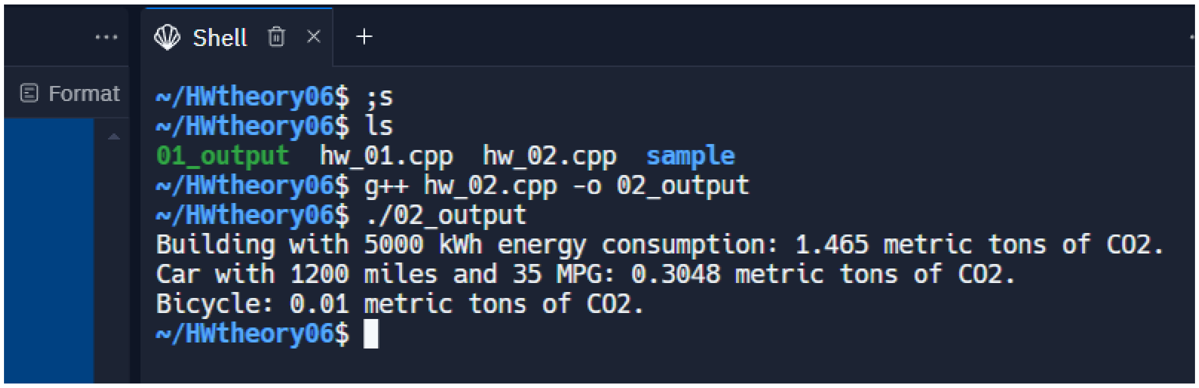
2. **Using an abstract class with only pure virtual functions, you can specify similar behaviors for possibly disparate classes. Governments and companies worldwide are becoming increasingly concerned with carbon footprints (annual releases of carbon dioxide into the atmosphere) from buildings burning various types of fuels for heat, vehicles burning fuels for power, and the like. Many scientists blame these greenhouse gases for the phenomenon called global warming. Create three small classes unrelated by inheritance -- classes *Building*, *Car* and *Bicycle*. Give each class some unique appropriate attributes and behaviors that it does not have in common**

with other classes. Write an abstract class *CarbonFootprint* with only a pure virtual *getCarbonFootprint* method. Have each of your classes inherit from that abstract class and implement the *getCarbonFootprint* method to calculate an appropriate carbon footprint for that class *(check out a few websites that explain how to calculate carbon footprints, such as https://www.youtube.com/watch?v=wWeI0c1m14Y)*. Write an application that creates objects of each of the three classes, places pointers to those objects in a vector of *CarbonFootprint* pointers, then iterates through the vector, polymorphically invoking each object's *getCarbonFootprint* method. For each object, print some identifying information and the object's carbon footprint.

CODE:

```
#include <iostream>
#include <vector>
#include <memory>

using namespace std;

// Abstract class
class CarbonFootprint {
public:
    virtual double getCarbonFootprint() const = 0;  // Pure virtual function
    virtual void print() const = 0;                 // Pure virtual print function for polymorphic behavior
    virtual ~CarbonFootprint() {}
};

// Building class
class Building : public CarbonFootprint {
private:
    double energyConsumed; // in kWh
public:
    Building(double energy) : energyConsumed(energy) {}
    double getCarbonFootprint() const override {
        // Example calculation: energy consumed * 0.000293 metric tons of CO2 per kWh
        return energyConsumed * 0.000293;
    }
    void print() const override {
        cout << "Building with " << energyConsumed << " kWh energy consumption: ";
    }
};

// Car class
class Car : public CarbonFootprint {
private:
    double milesDriven;
    double fuelEfficiency; // miles per gallon
public:
    Car(double miles, double efficiency) : milesDriven(miles), fuelEfficiency(efficiency) {}
    double getCarbonFootprint() const override {
        // Example calculation: miles driven / fuel efficiency * 8.89 kg CO2 per gallon of gasoline
        return (milesDriven / fuelEfficiency) * 8.89 / 1000; // Convert to metric tons
    }
    void print() const override {
        cout << "Car with " << milesDriven << " miles and " << fuelEfficiency << " MPG: ";
    }
};

// Bicycle class
class Bicycle : public CarbonFootprint {
public:
    double getCarbonFootprint() const override {
        // Assuming minimal carbon footprint
        return 0.01; // Arbitrary small number
    }
    void print() const override {
        cout << "Bicycle: ";
    }
};

int main() {
    // Vector of pointers to CarbonFootprint objects
    vector<shared_ptr<CarbonFootprint>> items;
    items.push_back(make_shared<Building>(5000));
    items.push_back(make_shared<Car>(1200, 35));
    items.push_back(make_shared<Bicycle>());

    // Polymorphically invoking each object's getCarbonFootprint and print methods
    for (const auto& item : items) {
        item->print();
        cout << item->getCarbonFootprint() << " metric tons of CO2." << endl;
    }

    return 0;
}
```

OUTPUT:



```
~/HWtheory06$ ;s
~/HWtheory06$ ls
01_output  hw_01.cpp  hw_02.cpp  sample
~/HWtheory06$ g++ hw_02.cpp -o 02_output
~/HWtheory06$ ./02_output
Building with 5000 kWh energy consumption: 1.465 metric tons of CO2.
Car with 1200 miles and 35 MPG: 0.3048 metric tons of CO2.
Bicycle: 0.01 metric tons of CO2.
~/HWtheory06$ 
```