Middle East Technical University Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring 2020-2021

Programming Exam 6

Due date: 3 Jun 2021, Thursday, 23:59

1 Problem Definition

In your second programming examination for C++, you are going to implement a real estate agency simulation. There will be owners to buy and sell their properties and properties to be owned.

1.1 General Specifications

- The signatures of the functions, their explanations and specifications are given in the following section. Read them carefully.
- Make sure that your implementations comply with the function signatures.
- You may define any number of helper function(s) as you need.
- You are not allowed to import any *extra* library for this exam.
- You should NOT make any change in header files. Those given to you will not be used in evaluation process.

1.2 Quick VPL Tips (In case you've forgotten them!)

- Evaluation is fast. If evaluation seems to hang for more than a few seconds, your code is entering an infinite loop or has an abnormal algorithmic complexity. Or you've lost your connection, which is *much* less likely!
- Although the run console does not support keyboard shortcuts, it should still be possible to copy and paste using the right-click menu (Tested on latest versions of Firefox and Chrome).
- Get familiar with the environment. Press the plus shapes button on the top-left corner to see all the options. You can download/upload files, change your font and theme, switch to fullscreen etc. Useful stuff!

2 Real Estate Agency

There are two base classes in this programming exam:

- Owner
- Property

2.1 Owner

Owner Class Properties

```
class Owner
{
...
protected:
    std::string name;
    float balance;
    std::vector<Property *> properties;
};
```

Each owner has a name, a balance value and the list of properties owned. There are 2 classes derived from Owner class:

- Person
- Corporation

These two classes differ from each other buy their class properties. Person class has age property while Corporation class has address information.

Person and Corporation Class Properties

```
class Person : public Owner
{
private:
    int age;
...
};

class Corporation : public Owner
{
private:
    std::string address;
...
};
```

Person/Corporation instances may own any number of properties. They can sell a property if they own that property and they can buy a property if they can afford it.

Constructors of these derived classes are as follows:

Person and Corporation Class Constructors

```
Person(const std::string &name, float balance, int age);

Corporation(const std::string &name, float balance, std::string address);
```

There are 6 functions of these classes to be implemented:

- **void print_info():** prints name and age of the owner if it is a person, prints name and address of the owner if it is a corporation.

  ```
  If Person:
  Name: <name> Age: <age>
  If Corporation:
  Name: <name> Address: <age>
  ```

- **std::string get_name():** returns the name of the owner.

- **void add_property(Property *property):** takes a property pointer and adds it to the property list of the owner.

- **void buy(Property *property, Owner *seller):** takes a property pointer and an owner pointer describing the seller. If the seller has corresponding estate and this owner has enough money to buy this property, property will be sold to this owner. Property lists and balance values of the buyer and seller will be updated. At the beginning of the transaction, operation is printed as:

  ```
  [BUY] Property: <name of the property> Value: <value of the property> <Seller>--->< Buyer>
  ```

- **void sell(Property *property, Owner *buyer):** takes a property pointer and an owner pointer describing the buyer. If this owner has corresponding estate and the buyer has enough money to buy this property, property will be sold to buyer. Property lists and balance values of buyer and seller will be updated. At the beginning of the transaction, operation is printed as:

```
[SELL] Property: <name of the property> Value: <value of the property> <Seller>---><Buyer>
```

- **void list_properties():** prints the balance value and the list of properties.

```
Properties of <name>:
Balance: <balance>$
1. <name of the property-1>
2. <name of the property-2>
...
n. <name of the property-n>
```

2.1.1 Unsuccessful Transactions

When making a transaction (buy/sell), it is first checked whether the buyer has sufficient balance. If the buyer does not have enough money:
[ERROR] Unaffordable property
text is printed.

If the buyer has enough money, it is checked whether the seller owns the property. If the seller is not the owner of the property:
[ERROR] Transaction on unowned property
text is printed.

2.2 Property

Property Class Properties

```
class Property
{
...
protected:
    std::string property_name;
    int area;
    Owner *owner;
};
```

Each property has a name, an area value and the owner information. There are 3 classes derived from Property class:

- Villa
- Apartment
- Office

These three classes differ from each other buy their class properties. Villa Class includes the information of how many floors it has and whether it has a garden or not. The Apartment Class, on the other hand, keeps the information on which floor it is and whether there is an elevator. Finally, Office Class includes information on whether there is a provided Wi-fi and reception service.

Villa, Apartment, Office Class Properties

```
class Villa : public Property
{
private:
    int number_of_floors;
    bool having_garden;
...
};
class Apartment : public Property
{
private:
    int floor;
    bool having_elevator;
...
};
class Office : public Property
{
private:
    bool having_wifi;
    bool having_reception;
...
};
```

Constructors of these derived classes are as follows: Villa, Apartment and Office Class Constructors

```
Villa(const std::string &property_name, int area, Owner *owner, int
   number_of_floors, bool having_garden);

Apartment(const std::string &property_name, int area, Owner *owner, int
   floor, bool having_elevator);

Office(const std::string &property_name, int area, Owner *owner, bool
   having_wifi, bool having_reception);
```

Note-1: While creating instances of derived classes, you have to add that instances into property list of the owner as well if the owner is provided (not given as NULL pointer).

Note-2: A property can be owned by an owner. It can be bought or sold. If it changes hands, the owner information of this property should be updated.

There are 4 functions of these classes to be implemented:

- **std::string get_name():** returns name of the property.
- **void print_info():** prints name and area of the property and its owner in following format:

  ```
  If has owner:
  <name of the property> (<area> m2) Owner: <name of the owner>
  Else:
  <name of the property> (<area> m2) Owner: No owner
  ```

- **void set_owner(Owner *owner):** takes an owner pointer and sets the property's owner information.

- **float valuate():** calculates and returns the value of the property. Each type of property has different valuation method:
  - Villa: $area \times 10 \times number\_of\_floors$. If it has garden double the value.
  - Apartment: $area \times 10 \times (\frac{floor}{10})$. If the building has elevator multiply the value by 1.5.
  - Office: $area \times 5$. If there is a provided Wi-Fi, multiply the value by 1.3. If there is a provided reception service, multiply the value by 1.5. Multiply with both, if both of them are provided.

3 Examples

Example-Constructors

```
int main()
{
    Person per = Person("Ahmet", 5000, 35);
    Corporation corp = Corporation("ACME", 5000, "cankaya");

    Villa est1 = Villa("Villa 1", 150, &per, 2, false);
    Apartment est2 = Apartment("Apartment 1", 200, NULL, 7, true);

    per.print_info();
    corp.print_info();
    est1.print_info();
    est2.print_info();

    return 0;
}

// Output
----------------------------
Name: Ahmet Age: 35
Name: ACME Address: cankaya
Villa 1 (150 m2) Owner: Ahmet
Apartment 1 (200 m2) Owner: No owner
```

Example-Buy/Sell Operations

```
int main()
{
    Person per = Person("Ahmet", 50000, 45);
    Corporation corp = Corporation("ACME", 100000, "cankaya");

    Villa est2 = Villa("Villa 1", 150, &per, 2, false);
    Apartment est3 = Apartment("Apartment 1", 200, &corp, 7, 1);
    Apartment est4 = Apartment("Apartment 2", 200, &corp, 5, 0);

    cout << "----------------------------\n";
    per.list_properties();
    corp.list_properties();
    cout << "----------------------------\n";
    per.buy(&est3, &corp);
    cout << "----------------------------\n";
    per.list_properties();
    corp.list_properties();
    cout << "----------------------------\n";
    per.sell(&est2, &corp);
    cout << "----------------------------\n";
    per.list_properties();
    corp.list_properties();
    cout << "----------------------------\n";

    return 0;
}

// Output
----------------------------
Properties of Ahmet:
Balance: 50000$
1. Villa Veli
Properties of ACME:
Balance: 100000$
1. Apartment Ali
2. Apartment Veli
----------------------------
[BUY] Property: Apartment Ali Value: 2100$ ACME--->Ahmet
----------------------------
Properties of Ahmet:
Balance: 47900$
1. Villa Veli
2. Apartment Ali
Properties of ACME:
Balance: 102100$
1. Apartment Veli
----------------------------
[SELL] Property: Villa Veli Value: 3000$ Ahmet--->ACME
----------------------------
Properties of Ahmet:
Balance: 50900$
1. Apartment Ali
Properties of ACME:
Balance: 99100$
1. Apartment Veli
2. Villa Veli
----------------------------
```

Example-Balance Problem

```
int main()
{
    Person per = Person("Ahmet", 1000, 10);
    Corporation corp = Corporation("ACME", 1000, "cankaya");

    Villa est1 = Villa("Villa 1", 150, &per, 2, false);
    Apartment est2 = Apartment("Apartment 1", 200, &corp, 7, 1);
    Apartment est3 = Apartment("Apartment 2", 200, &corp, 5, 0);

    cout << "----------------------------\n";
    per.list_properties();
    corp.list_properties();
    cout << "----------------------------\n";

    // Ahmet cannot afford Apartment 1
    per.buy(&est2, &corp);
    // Ahmet cannot afford Apartment 1
    corp.sell(&est2, &per);
    cout << "----------------------------\n";
    per.list_properties();
    corp.list_properties();
    cout << "----------------------------\n";

    return 0;
}

// Output
----------------------------
Properties of Ahmet:
Balance: 1000$
1. Villa 1
Properties of ACME:
Balance: 1000$
1. Apartment 1
2. Apartment 2
----------------------------
[BUY] Property: Apartment 1 Value: 2100$ ACME--->Ahmet
[ERROR] Unaffordable property
[SELL] Property: Apartment 1 Value: 2100$ ACME--->Ahmet
[ERROR] Unaffordable property
----------------------------
Properties of Ahmet:
Balance: 1000$
1. Villa 1
Properties of ACME:
Balance: 1000$
1. Apartment 1
2. Apartment 2
----------------------------
```

Example-Selling not owned property or buying a property not owned by seller

```
int main()
{
    Person per = Person("Ahmet", 1000, 10);
    Corporation corp = Corporation("ACME", 1000, "cankaya");

    Villa est1 = Villa("Villa 1", 150, &per, 2, false);
    Apartment est2 = Apartment("Apartment 1", 200, &corp, 7, 1);
    Apartment est3 = Apartment("Apartment 2", 200, &corp, 5, 0);

    cout << "----------------------------\n";
    per.list_properties();
    corp.list_properties();
    cout << "----------------------------\n";
    // ACME do not own Villa 1
    per.buy(&est1, &corp);
    // ACME do not own Villa 1
    corp.sell(&est1, &per);
    cout << "----------------------------\n";
    per.list_properties();
    corp.list_properties();
    cout << "----------------------------\n";

    return 0;
}

// Output
----------------------------
Properties of Ahmet:
Balance: 5000$
1. Villa 1
Properties of ACME:
Balance: 5000$
1. Apartment 1
2. Apartment 2
----------------------------
[BUY] Property: Villa 1 Value: 3000$ ACME--->Ahmet
[ERROR] Transaction on unowned property
[SELL] Property: Villa 1 Value: 3000$ ACME--->Ahmet
[ERROR] Transaction on unowned property
----------------------------
Properties of Ahmet:
Balance: 5000$
1. Villa 1
Properties of ACME:
Balance: 5000$
1. Apartment 1
2. Apartment 2
----------------------------
```

4 Regulations

1. **Implementation and Submission:** The template files are available in the Virtual Programming Lab (VPL) activity called "PE6" on `OdtuClass`. At this point, you have two options:

   - You can download the template files, complete the implementation and test it with the given sample I/O on your local machine. Then submit the same file through this activity.
   - You can directly use the editor of VPL environment by using the auto-evaluation feature of this activity interactively. Saving the code is equivalent to submit a file.

   The second one is recommended. However, if you're more comfortable with working on your local machine, feel free to do it. Just make sure that your implementation can be compiled and tested in the VPL environment after you submit it.

   All source files are given to you so that you can work on your local machines. If you choose first option, you have to submit header files and test main file as well but they will not be included into evaluation process.

   There is no limitation on online trials or submitted files through OdtuClass. The last one you submitted will be graded.

2. **Programming Language:** You must code your program in C++. Your submission will be compiled with g++ on VPL. You are expected to make sure your code compiles successfully with g++ using the flags -ansi -pedantic.

   Makefile is provided to you. You can use following commands:

   - **make main** ⟶ compiles test main with all sources files
   - **make test test=<path of test main>** ⟶ compiles selected test with all sources
   - **make run_main** ⟶ runs compiled test main
   - **make run_test** ⟶ runs compiled test
   - **make run_test_to_file test=<name of the test>** ⟶ compiles and runs selected test and writes the output to a file with the same name
   - **make clean** ⟶ cleans execution files

3. **Cheating: We have zero tolerance policy for cheating**. People involved in cheating (any kind of code sharing and codes taken from internet included) will be punished according to the university regulations.

4. **Evaluation:** Your program will be evaluated automatically using "black-box" technique so make sure to obey the specifications. No erroneous input will be used. Therefore, you don't have to consider the invalid expressions.

   Important Note: The given sample I/O's are only to ease your debugging process and NOT official. Furthermore, it is not guaranteed that they cover all the cases of required functions. As a programmer, it is your responsibility to consider such extreme cases for the functions. Your implementations will be evaluated by the official testcases to determine your ***actual*** grade after the deadline.