Homework 2

[2024-2] 데이터사이언스 응용을 위한 컴퓨팅 (001)

Due: 2024년 11월 3일 23:59

1. Bank Account Management System [50pts]

In this homework, you will implement a Bank Account Management System using Object-Oriented Programming principles. Implement the TODO sections in `bankaccount.hpp`.

Transaction Struct

The `Transaction` struct is used to log each operation (deposit or withdrawal) on a bank account. Every time a deposit or withdrawal is made, a new `Transaction` is added to the account's `transactionHistory`. Each `Transaction` contains:

- `deposit`: The amount of money deposited into the account (if the transaction is a deposit, otherwise 0).
- `withdraw`: The amount of money withdrawn from the account (if the transaction is a withdrawal, otherwise 0).
- `balance`: The balance of the account after the transaction.

BankAccount

The `BankAccount` class defines the core operations of a bank account. It manages the account balance (`balance`) and transaction history (`transactionHistory`), and provides fundamental functionalities such as deposit and withdrawal.

Q 1.1: BankAccount Constructor

Instruction: Implement a constructor for the `BankAccount` class that accepts an initial balance and stores it in the `balance` variable. This constructor should initialize the account with the given balance, which represents the starting amount of money in the account.

- Parameter:
    - `double initialBalance`: The initial balance of the account.
- Return: None.

Q 1.2: deposit Method

Instruction: Declare a method `deposit` in the `BankAccount` class. This method should accept an `amount` to deposit into the account and log the deposit in the `transactionHistory`. When this method is called, a `Transaction` is logged with the `deposit` field set to the `amount`, the `withdraw` field set to 0, and the `balance` updated to reflect the new balance after the deposit.

- Parameter:
    - `double amount`: The amount to deposit into the account.
- Return: None.

Q 1.3: withdraw Method

Instruction: Declare a method `withdraw` in the `BankAccount` class. This method should accept an `amount` to withdraw from the account and log the withdrawal in the `transactionHistory`. The transaction should record the withdrawn amount, 0 for the deposit, and the balance after the withdrawal. If the amount to withdraw exceeds the current balance, the withdrawal will fail. Even if the withdrawal fails, the transaction should still be logged in the `transactionHistory`. In this case, the `deposit` should be 0, the `withdraw` should be 0, and the `balance` should remain.

- Parameter:
    - `double amount`: The amount to withdraw from the account.
- Return: None.

Q 1.4: getBalance Method

Instruction: Implement the `getBalance` method, which returns the current balance of the account. This method should allow users to check the balance without modifying it.

- Parameter: None.
- Return:
    - `double`: The current balance of the account.

Q 1.5: getTransactionHistory Method

Instruction: Implement the `getTransactionHistory` method, which returns a reference to the vector containing the `transactionHistory` of the account. This method provides read-only access to all recorded deposits and withdrawals.

- Parameter: None.
- Return:

- `const std::vector<Transaction>&`: A reference to the vector of `Transaction` objects, which store the deposit, withdrawal, and balance data.

SavingsAccount

Savings accounts typically earn interest on deposits.

Q 2.1: SavingsAccount Constructor

Instruction: Implement a constructor for the `SavingsAccount` class. This constructor should initialize the `BankAccount` base class with an initial balance and store the interest rate in the `interestRate` variable.

- Parameter:
  - `double initialBalance`: The initial balance of the savings account.
  - `double rate`: The interest rate applied to the account. `rate` is a value between 0 and 0.5
- Return: None.

Q 2.2: deposit Method (Override for SavingsAccount)

Instruction: Override the `deposit` method for the `SavingsAccount` class. This method should deposit the given amount into the account and calculate and apply interest based on the `interestRate`. The total deposit consists of the sum of the `amount` and the `interest`. The interest is calculated as `amount * interestRate`. When this method is called, the `deposit` field of the `Transaction` should reflect the sum of the deposit and interest (i.e., `amount + interest`). The `withdraw` field is 0, and the `balance` is updated to the new balance after the interest is applied.

- Parameter:
  - `double amount`: The amount to deposit into the savings account.
- Return: None.

Q 2.3: Destructor for SavingsAccount

Instruction: Implement a destructor for the `SavingsAccount` class. The destructor should print a message “SavingsAccount closed”.

- Parameter: None.
- Return: None.

CheckingAccount

Checking accounts typically charge a transaction fee for each withdrawal. This fee will be deducted from the account balance whenever money is withdrawn.

Q 3.1: CheckingAccount Constructor

Instruction: Implement a constructor for the `CheckingAccount` class. This constructor should initialize the `BankAccount` base class with an initial balance and store the transaction fee in the `transactionFee` variable.

- Parameter:
    - `double initialBalance`: The initial balance of the checking account.
    - `double fee`: The transaction fee charged for each withdrawal.
- Return: None.

Q 3.2: withdraw Method (Override for CheckingAccount)

Instruction: Override the `withdraw` method for the `CheckingAccount` class. This method should withdraw the specified `amount` from the account, plus the `transactionFee`. If the balance is insufficient to cover both the amount and the fee, no withdrawal should occur. When a withdrawal is successful, a `Transaction` is logged with the `withdraw` field set to the sum of the `amount` and the `transactionFee`. The `deposit` field is 0, and the `balance` is updated to reflect the new balance after the fee is applied. Even if the withdrawal fails, the transaction should still be logged in the `transactionHistory`. The `deposit` should be 0, the `withdraw` should be 0, and the `balance` should remain unchanged.

- Parameter:
    - `double amount`: The amount to withdraw from the checking account.
- Return: None.

Q 3.3: Destructor for CheckingAccount

Instruction: Implement a destructor for the `CheckingAccount` class. The destructor should print a message “CheckingAccount closed”.

- Parameter: None.
- Return: None.

BusinessAccount

Business accounts typically offer a credit line, which allows the account to withdraw funds even if the balance is insufficient, as long as the total withdrawal amount is within the credit limit. For instance, if the balance is 1000 and the credit line is 200, you can withdraw up to 1200, leaving the balance at -200.

Q 4.1: BusinessAccount Constructor

Instruction: Implement a constructor for the `BusinessAccount` class. This constructor should initialize the `BankAccount` base class with an initial balance and store the credit line in the `creditLine` variable.

- Parameter:
  - `double initialBalance`: The initial balance of the business account.
  - `double credit`: The credit line available for the account.
- Return: None.

Q 4.2: withdraw Method (Override for BusinessAccount)

Instruction: Override the `withdraw` method for the `BusinessAccount` class. This method should allow withdrawals up to the combined total of the balance and the credit line. If the withdrawal amount exceeds the available balance plus the credit line, no withdrawal should occur. When a withdrawal is successful, a `Transaction` is logged with the `withdraw` field set to the `amount`. The `deposit` field is 0, and the `balance` is updated to reflect the new balance, taking into account the use of the credit line if necessary. Even if the withdrawal fails, the transaction should still be logged in the `transactionHistory`. The `deposit` should be 0, the `withdraw` should be 0, and the `balance` should remain unchanged.

- Parameter:
  - `double amount`: The amount to withdraw from the business account.
- Return: None.

Q 4.3: Destructor for BusinessAccount

Instruction: Implement a destructor for the `BusinessAccount` class. The destructor should print a message “BusinessAccount is closed”.

- Parameter: None.
- Return: None.

AccountManager

The `AccountManager` is responsible for managing multiple bank accounts, allowing for operations such as adding new accounts and transferring funds between them.

Q 5.1: addAccount Method

Instruction: Implement the `addAccount` method in the `AccountManager` class. This method should add a new bank account to the manager's list of accounts. The method should store the account as a `std::shared_ptr<BankAccount>`.

- Parameter:
  - `const std::shared_ptr<BankAccount>& account`: A shared pointer to the account being added.
- Return: None.

Q 5.2: transferFunds Method

Instruction: Implement the `transferFunds` method in the `AccountManager` class. This method should transfer money from one account to another. The method should first withdraw the specified amount from the account at `fromIndex` and then deposit the same amount into the account at `toIndex`. If either index is invalid, throw a `runtime_error` with a message “Invalid account index”. Make sure that `transferFunds` can only proceed if the withdrawal from the account is successfully completed.

- Parameter:
  - `int fromIndex`: The index of the account to withdraw from.
  - `int toIndex`: The index of the account to deposit into.
  - `double amount`: The amount to transfer between accounts.
- Return: None.

Q 5.3: getAccount Method

Instruction: Implement the `getAccount` method in the `AccountManager` class. The `AccountManager` keeps track of all accounts it manages. This method provides access to specific accounts by their index in the account list. This method should return a raw pointer to the bank account at the specified index. If the index is invalid, throw a `runtime_error` with a message “Invalid account index”.

- Parameter:
  - `int index`: The index of the account to retrieve.
- Return:
  - `std::shared_ptr<BankAccount>`: A pointer to the account at the given index.

2. Custom Map

This assignment involves implementing a map that is similar to std::map in the C++ standard library. A map typically consists of key-value pairs, and its construction allows for the specification of the types of keys and values through templates. Additionally, the template for the map takes std::less as the default type for Compare and std::allocator as the default type for Allocator. Figure 3 is the prototype of the actual C++ std::map.

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T>>
> class map;
```

Figure 3 Prototype of map

The map to be implemented in this assignment should be designed as a template that takes K as the data type for the key, V as the data type for the value, and Compare as a comparison class. In this assignment, you don't have to consider the Allocator type. std::less typically takes two data of the same type and compares which one is smaller. Below is an example usage of std::less. Additional explanations can be accessed through the link.

```
template <typename K, typename V, typename Compare = std::less<K>>
class map {
    private:
        // ....
        // other member variables
        // ....
        Compare comp;
        // ....
        // other member variables
        // ....
    public:
        // ....
        // other member functions
        // ....
        void example_function(K key1, K key2){
            if(comp(key1,key2)){
                cout << "Key1 is smaller than key2" << endl;
            }else if(comp(key2,key1)){
                cout << "Key2 is smaller than key1" << endl;
            }
        }
};
```

Figure 4 Example code of std::less

The objective of this assignment is to implement a CustomMap that stores data in a Binary

Search Tree (BST). In this case, the binary search tree stores data based on the key, in accordance with the BST property, and the value is stored in the same TreeNode as a member variable. The goal is to implement operations such as insert, delete, and traverse, and operator[ ] using std::less for comparing keys.

Instruction

Read all explanations above and subproblems (a) through (e) below and implement them in "custommap.hpp". To receive full credit, you must declare all member functions and member variables as instructed in the subproblems. Additionally, declaring extra member functions and variables is permissible in the completion of the assignment. **If you don't implement the BST properly, time out errors can occur. Make sure to implement BST with O(logN) time complexity. Also, if you use std::map on your implementation, you will get 0 credits. Make sure to write your code not to contain "#include <map>" Not even a comment. Unless you will get 0 credits.**

(a) [10 pts] Define the TreeNode class using a class template. The class template should take two template arguments: the type of the keys, the type of the values. TreeNode should declare private member variables **K key**, **V value**, **TreeNode* left**, and **TreeNode* right**. The constructor should take K k and V v and assign them to **key** and **value** accordingly. Additionally, there should be getter and setter functions for each member variable. Name of each getter, setter function is not fixed, so you can name it freely. Similarly, define the CustomMap class using a class template. The class template should take three template arguments: the type of the keys, the type of the values, the compare class std::less<K> for comparison inside the map data structure(refer to Figure 4). CustomMap class should have TreeNode corresponding to the root of the BST, **TreeNode* root** and **Compare object** as member variables. The CustomMap constructor should not take any arguments. CustomMap class should have **TreeNode<K,V>* get_root( )** and **Compare get_compare( )** as a getter function.

(b) [10 pts] Write a **void insert(K key, V value)** function that inserts the key-value into the BST as a TreeNode, in accordance with the properties of a binary search tree. And also, write a function **void print_map()** that prints the key:value pairs stored within CustomMap in ascending order of keys. The output from running should be like figure 5. Make sure to print the "key:value\n" form exactly, without any space or other characters. Unless you won't get any credits. Grading based on test cases will rely on the output of the print_map() function. So even if the insert() function is implemented correctly, you will get no credits if the print_map() function is not implemented correctly.

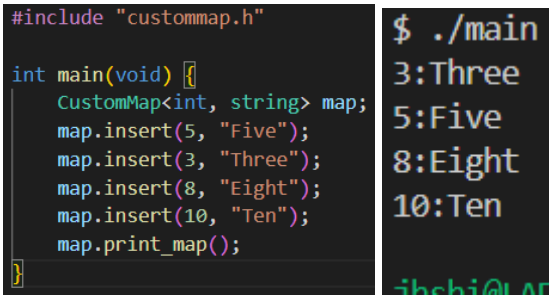

Figure 5 Result of insert and print_map

(c) [10 pts] Write a function **void deleteKey(K key)** that takes a key of type K as an argument and deletes the node corresponding to that key. If the input key does not exist, the delete function should perform nothing. Grading based on test cases will rely on the output of the print_map() function and insert function to construct the tree, so even if the deleteKey function is implemented correctly, you will get no credits if the print_map() and insert() function is not implemented correctly.

(d) [10 pts] Write a function **V get_value(K key)** that takes a key of type K as an argument and returns the corresponding value of type V. If the input key does not exist, the get_value function should return default value, V(). Grading based on test cases will rely on the insert function to construct the tree, so even if the get_value function is implemented correctly, you will get no credits if the insert() function is not implemented correctly.

(e) [10 pts] Overload the **[ ] operator** such that you can access a value using a key. When the key exists in the map, this operator returns the corresponding value. When the key does not exist, this operator inserts a new element with the key and the default value V( ). Also, this operator should allow for modifying the corresponding value directly. For example, by “map[3] = 4”, this operator has to perform “adding a pair whose key is 3 and value is 4” if there is no key 3, and has to replace its value with 4 if the key exists. Grading based on test cases will rely on the insert function to construct the tree, so even if the operator[ ] is implemented correctly, you will get no credits if the insert() function is not implemented correctly.

Submission Requirements

- Complete the implementation of all TODOs in the `bankaccount.hpp` and `custommap.hpp`.
- Be careful with formatting and typos.
- For Homework 2 Problem 1 on GradeScope, please submit `bankaccount.hpp`, and for Homework 2 Problem 2, please submit `custommap.hpp`.
- Grading will be conducted using C++11 standard. Our compilation process is as follows:
- `$ g++ main.cpp -o main -std=c++11`
- `$ ./main`
- You must not share your code with other students. Any student found to have a high level of code similarity, as determined by our similarity detection process, may face serious penalties.
- If you submit late, grace days will be automatically deducted. You have 5 grace days available for homework submissions throughout the semester. Grace days are counted in 24-hour increments from the original due time. For example, if an assignment is due on 11/3 at 11:59 PM, submitting it 30 minutes late will use one grace day, while submitting it on 11/5 at 9:00 PM will use two grace days. Late submissions will NOT be accepted once all grace days have been used.
- **Before submitting your code to GradeScope, please make sure to remove any unnecessary debugging code that uses `cout`.**