Lab rapport in C++ OOP

Hergeir Winther Lognberg
Hewi1600

1 Preamble

Assignment was to create a Bank which operated the way the lab described.

2 The Code

2.1 placement

I've decided to keep all files associated with the lab in the root of the project folder.

2.2 namespace

I chose to remove

```
1 using namespace std;
```

from all files it was previously used in. I find that this improves readability and clearly seperates std functions from self-made ones.

2.3 code

Only difference between this lab and Lab 2 is the polymorphism. Only functions from the original project that needed to be changed where:

- savetoFile and loadFrom file
  - needed to alter the functions to work with accountType and amountWithdrawals
- Account class:

  - made some functions *virtual* and moved the *private* variables to *protected* as this will be baseclass in a polymorphism.

- added a Menu object in TestApp class
  - to take the values from user for accountType
- All i then needed to do was to create the child classes
  - TransactionAccount
  - SavingsAccount
    - *LongTermSavingsAccount*

Still I have not had the need for a shared pointer.

2.4 AccountInfo

To easily manage and return account info for all the accountTypes i revised the AccountInfo struct to this:

```
1  struct AccountInfo
2  {
3     const unsigned int accountNo;
4     const std::string accountType;
5     const double balance;
6     const double credit;
7     const double interest;
8     const double available;
9     //default constructor
10    AccountInfo()
11    :accountNo(0),balance(0),credit(0),interest(0),available(0)↩
          {}
12    //constructor
13    AccountInfo(const unsigned int pAccountNo,const std::string ↩
         & pAccountType,const double pBalance, const double ↩
         pCredit, const double pInterest, const double pAvailable↩
         )
14    :accountNo(pAccountNo),accountType(pAccountType),balance(↩
         pBalance),credit(pCredit),interest(pInterest),available(↩
         pAvailable){}
15 };
```

I return this struct upp through the classes:

$$Account \rightarrow Cutsomer \rightarrow Bank$$

All of them contain a function called:

```
ClassName::getAccountInfo(const unsigned int)
```

for whenever printing account info on specific account is needed.

I initialize the struct in Account class using the virtual getFunctions for all the values. Like this:

```
// this function will follow to the other classes and give correct values for all.
const AccountInfo Account::getAccountInfo() const
{
    return AccountInfo(accountNo,getTypeText(),balance,getCredit(),getInterest(),getUsableBalance());
}
```

accountNo and balance are the only variables that are in common for all the accounts. Therefor i don't have to use get in their case. The other values return *0* (basecase definition of virtual function) unless the child class contains the variables.

2.5 enum

I used enum to easily manage the accountTypes and ensuring that the accountType only can be valid value.

3 Question

I'm pretty sure what I did was correct. I did not see a requirement to be able to set interest for any type of account. So I didn't add one. However one requirement was that the *LongTermSavingsAccount* should always have 2% higher interes rate than SavingsAccount. I wasn't really sure how I should implement this. I ended up overriding the getInterest function of the *LongTermSavingsAccount* class to return the interest + 2% like this:

```
//In SavingsAccount.hpp:
protected:
static double interest;
virtual const double getInterest() const override;

//In SavingsAccount.cpp
const double SavingsAccount::getInterest() const
{
    return interest;
```

```
}

//In LongTermSavingsAccount.hpp
protected:
virtual const double getInterest() const override;

// In LongTermSavingsAccount.cpp
double SavingsAccount::interest;
const double LongTermSavingsAccount::getInterest() const
{
   return interest+0.02;
}
```

I ended up with a *LongTermSavingsAccount* class that in every sense (except variable value) always reported having 2% higher interest rate than *SavingsAccounts*.

The advantages with this approach (that I know of)

- only one interest variable (less vasted space).
- Able to change interest of all accounts of specific type at once.
- able to implement LongTermSavingsAccount always having 2% higher interes than all SavingsAccounts

Disadvantages:

- all SavingsAccounts (and by extenstion the LongTermSavingsAccounts) can not have different interest rates.
- not being able to make sure that if we change interest of a random SavingsAccount that LongTermSavingsAccount will follow.

Was I correct in my approach?

4 Building/Compiling

Just run *make* in the Lab directory. To run the program run *make run* in same directory.

5 Enviroment

I'm programming on an Arch linux 64-bit system. I've got the gcc compiler installed and compile using it's g++ alias which links necessary libraries

automatically. To compile I use the recommended flags: "-std=c++11 -Wall -pedantic". The flags let me choose to use c++11 standard and give me useful compiling warnings and errors. For editing of code i currently use VS code with a makefile.

6 Backup

And if anything's missing you can find it on:
github: `https://github.com/Hergeirs/Cpp-Obj/tree/master/Level%202/Lab3`
Cpp-obj/Lab2

September 23, 2017