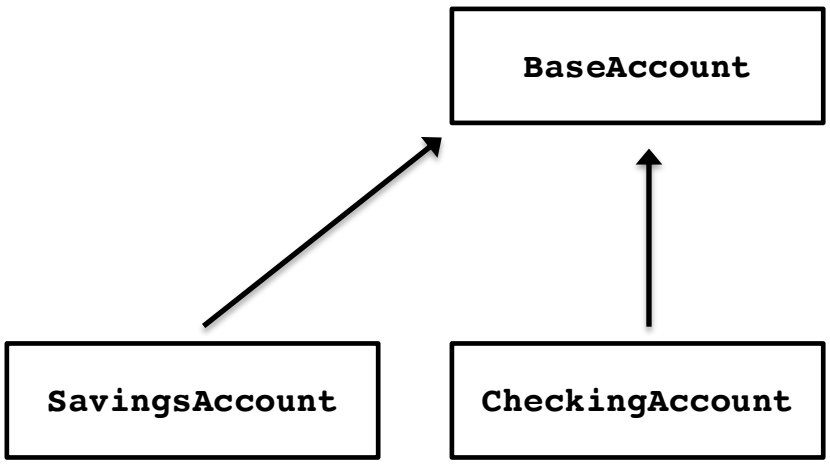
Bank Account Inheritance Hierarchy Program (10 Points)

Create an inheritance hierarchy that a bank might use to represent customers' bank accounts. All customers at this bank can deposit money into their accounts and withdraw money from their accounts. More specific types of accounts also exist. Savings accounts, for instance, earn interest on the money they hold. Checking accounts, on the other hand, charge a fee per transaction (i.e., for each deposit and each withdraw).

Create an inheritance hierarchy containing base class **BaseAccount** and derived classes **SavingsAccount** and **CheckingAccount** that inherit from class **BaseAccount** .



The base class, **BaseAccount**, should include one data member of type **long int** to represent the account balance in Pennies!! This is the internal representation of the account balance. All of the public functions will express the balance as a **double**. For example, $25.37 is Twenty Five Dollars and 37 cents, or 2537 pennies! The class should provide a constructor that receives an initial balance in dollars and cents (as a **double**) and uses it to initialize the data member. To convert that **double** to a **long int** Just multiply by 100.0, **round()** and **cast<>** the result to **long int**. The constructor should validate the initial balance to ensure that it's greater than or equal to $0.00. If not, the balance should be set to 0L and the constructor should display an error message, indicating that the initial balance was invalid. The class should provide three member functions. Member function **deposit()** should add an amount (expressed in dollars and cents as a **double**) to the current balance. Member function **withdraw()** should withdraw money (expressed in dollars and cents as a **double**) from the **BaseAccount** and ensure that the debit amount does not exceed the **BaseAccount**'s balance. If it does, the balance should be left unchanged and the function should print the message "Insufficient funds to withdraw from this account." Member function **getBalance()** should return the current balance (in dollars and cents as a **double**). 2537 pennies is $25.37, so just **cast<>** to **double** and divide by 100.0. There is Not a "setBalance()" function. All changes to the balanced are accomplished using **withdraw()** and **deposit()**.

The derived class, **SavingsAccount**, should inherit the functionality of an **BaseAccount**, but also include a data member of type double indicating the interest rate (percentage) assigned to the Account. **SavingsAccount's** constructor should receive the initial balance (as a **double**), as well as an initial value for the **SavingsAccount's** interest rate (as a **double**). **SavingsAccount** should provide a public member function **computeInterest()** that returns a double indicating the amount of interest earned by an account. Member function **computeInterest()** should determine this amount by multiplying the interest rate by the account balance. [Note: **SavingsAccount** should inherit member functions **deposit** and **withdraw** as is without redefining them.]

The derived class, **CheckingAccount**, should inherit from base class **BaseAccount** and include an additional data member of type double that represents the fee charged per transaction. **CheckingAccount**'s constructor should receive the initial balance (as a **double**), as well as a parameter indicating a fee amount (as a **double**). Class **CheckingAccount** should redefine member functions **deposit()** and **withdraw()** so that they subtract the fee from the account balance whenever either transaction is performed successfully. **CheckingAccount**'s versions of these functions should invoke the base-class **BaseAccount** version to perform the updates to an account balance.

CheckingAccount's debit function should charge a fee only if money is actually withdrawn (i.e., the debit amount does not exceed the account balance). [Hint: Define **BaseAccount**'s **withdraw()** function so that it returns a **bool** indicating whether money was withdrawn. Then use the return value to determine whether a fee should be charged.]

After defining the classes in this hierarchy, write a program that creates objects of each class and tests their member functions. Add interest to the **SavingsAccount** object by first invoking its **computeInterest()** function, then passing the returned interest amount to the object's **deposit()** function. There is no user input. No loops, if-statements, etc. Just a series of statements to create each of the 3 types of accounts and do the sequence of deposits, withdrawals, etc. The output should closely resemble the output provided.

Extra Credit: Add and use a string representing the owner of the account to the **BaseAccount**. It should be included in the constructor for the base account. In addition, provide member functions, **std::string getOnwer()** and **setOwner(std::string)**.

```
BaseAccount
+ BaseAccount ( initialBalance:double)
+ getBalance ( ) : double
+ deposit ( amount : double ): void
+ withdraw ( ) : bool
- balanceInPennies : long int

CheckingAccount
+ CheckingAccount (initialBalance:double, checkingFee:double)
+ deposit ( amount : double  ) : void
+ withdraw ( ) : bool
- checkingFee : double

SavingsAccount
+ SavingAccount (initialBalance:double, initialRate:double)
+ computeInterest ( ) : double
- interestRate : double
```

Sample Output:

```
Creating 3 Test Bank Accounts:
Account 1 (Base) balance: $50.01
Account 2 (Savings) balance: $25.02
Account 3 (Checking)balance: $80.03


Bank Account Withdrawal tests:
Now withdrawing $25.00 from account 1.
Now withdrawing $30.00 from account 2.
Insufficient funds to honor the withdrawal request.
Now withdrawing $40.00 from account 3.
$1.00 checking transaction fee charged.

Account 1 balance: $25.01
Account 2 balance: $25.02
Account 3 balance: $39.03


Bank Account Deposit tests:
Now depositing $40.23 to account 1.
Now depositing $65.99 to account 2.
Now depositing $20.01 to account 3.
$1.00 checking account transaction fee charged.

Account 1 balance: $65.24
Account 2 balance: $91.01
Account 3 balance: $58.04


Bank Account Interest test:
Adding $4.55 interest to account 2.
The new account 2 balance is: $95.56
```

Overview:

This assignments about inheritance. First create the base class, called **BaseAccount,** representing a generic bank account. The **BaseAccount** class, has one private data item, balance in pennies (***long int***). For example, $25.37, is stored internally as 2537 pennies. The **BaseAccount** class has public member functions to construct, **deposit()** and **withdraw()** to/from the account, and **getBalance()**. All public functions deal with money as a ***double***, representing dollars and cents. You will need to convert to pennies as you update the Internal balance of the **BaseAccount**, and from pennies to dollar and cents when you get the balance. The **CheckingAccount** and **SavingAccount** will inherit from the base class, **BaseAccount**. They will each add the needed data (interest rate and checking fee) and member functions calculate Interest for the savings account. The **CheckAccount** will need to redefine the **deposit()** and **withdraw()** to charge fees. Creating these classes is the bulk of the work on this assignment. There are NO ***friend*** classes or functions in this assignment. This is no setBalance() function...you change the balance using deposit and withdraw only!

The UML for the 3 classes are as follows (Sorry no way to add the relationship arrows in canvas! See the attached .pdf):

BaseAccount

```
+ BaseAccount (initialBalance:double)
+ getBalance ( ) : double
+ deposit (amount : double): void
+ withdraw (amount : double ) : bool
- balanceInPennies : long int
```

CheckingAccount

```
+ CheckingAccount (initialBalance:double, checkingFee:double)
+ deposit (amount : double) : void
+ withdraw (amount : double ) : bool
- checkingFee : double
```

SavingsAccount

```
+ SavingAccount (initialBalance:double, initialRate:double)
```

```
+ computeInterest ( ) : double

- interestRate : double
```

The **main()** has no logic, no loops, no flow control, no user input. Just statements to create the accounts, output the balance and the rest of the stuff in the sample output. This part is easy. Just simple statements using your your objects to show how they work. Use the example below as a guide. Your **main()** will have many **cout**'s along the away. Don't over think this part of the assignment.

```
Creating 3 Test Bank Accounts:

Account 1 (Base) balance: $50.01

Account 2 (Savings) balance: $25.02

Account 3 (Checking)balance: $80.03


Bank Account Withdrawal tests:

Now withdrawing $25.00 from account 1.

Now withdrawing $30.00 from account 2.

Insufficient funds to honor the withdrawal request.

Now withdrawing $40.00 from account 3.

$1.00 checking transaction fee charged.


Account 1 balance: $25.01

Account 2 balance: $25.02

Account 3 balance: $39.03


Bank Account Deposit tests:

Now depositing $40.23 to account 1.

Now depositing $65.99 to account 2.
```

```
Now depositing $20.01 to account 3.
$1.00 checking account transaction fee charged.

Account 1 balance: $65.24
Account 2 balance: $91.01
Account 3 balance: $58.04

Bank Account Interest test:
Adding $4.55 interest to account 2.
The new account 2 balance is: $95.56
```

Extra Credit:

Add a string representing the account owner to the base class. And accessor function getOwner() and setOwner(). Extend the assignment to use this new functionality.

Hints:

You MUST Follow the attached .pdf file!

Attach all 7 files (.hpp / .cpp pair for each class, and then main.cpp) PLUS the output file.