Part II: Adding Transactions in the Bank Account

This assignment continues the bank account application from Bank Part 1. In this part, we will add the ability for accounts to create a monthly statement. The description of the bank has customers, accounts, and transactions. In part 1, we implemented a customer class and an account class. Now let's implement a transaction class.

Each event that adds or removes money from an account is a transaction. In the statement, we will display a list of the transactions for that month. A good list should have just one line per transaction. A transaction should have a date, an amount, and a description. The description can be the person or thing at the other end of the transaction, or the location of an ATM transaction. In any case, we aren't going to do anything besides print the description, so we make it a string. For the date, we don't yet have a date class to deal with the complexities of arbitrary days, so we will just use an integer to hold the day in the month (i.e. from 1 to 30). Without a currency type, we will make the amount a double, just as we did for the balance in part 1.

Step 4

Create the Transaction class. As with the other classes, use the Project -> Add Class menu option to start the two files. The Transaction class has three properties, as described above: dayOfMonth, amount, and description. The three properties are of type int, double, and string, respectively. The only use of a transaction is to record things, so change the constructor to take values for all three properties as input arguments. (We won't allow empty transactions.) Then give it two more methods: getAmount() and display(). GetAmount() is a getter for the amount. The display method should output all three values to cout on a single line. Here are three lines you can add to your main function to test the new class.

```
Transaction tr1(17, 8.99, "Burger King");
tr1.display();
cout << tr1.getAmount() << endl;
```

Here is the corresponding output:

```
17  8.99  Burger King
8.99
```

Step 5

Now let's add transaction records to the account. First, the Account class must have a property to store Transactions, and a property to keep track of how many transactions there will be.

Because we are creating an array, we must set a maximum number of transactions for the month (100) and call it MAX_TRANSATIONS. Because this number is specific to the class, we should declare it as a static const int in the class and initialize it in the declaration.

```
static const int MAX_TRANSACTIONS = 100;
```

Create an array of Transactions, called "events," using MAX_TRANSACTIONS as the size. Create an int property called "numEvents" which we will use to keep track of how many transactions have been used. Don't forget that you need an `#include "Transaction.h"`.

Now we have a problem. The Transaction class does not allow us to create empty transactions. There is only one constructor and it takes all three values. So how can we create an

array of Transactions that have not yet happened? The solution is that we will not create an array of transactions. Instead, we will create an array of references to transactions. Then, as we create each new transaction, we will add a reference to it, in the array. The reference type has the same name as the type, but has a star after the type name.

```
Transaction* events[MAX_TRANSACTIONS];
```

When an Account object is first created, it must know that there are no transactions. C++ doesn't do that by default. So in the constructor for the Account class, which was previously setting balance to 0.0, we now have to add a line to set numEvents to 0 as well.

Step 6

To add a transaction for each deposit or withdrawal, we will need to define new versions of those methods, that do the same thing and then record the transaction. Do not remove the existing methods. We will use them. The new version of the withdraw method must take all three parameters needed for a transaction. We were already passing an amount and returning a bool. Add a new withdraw method that takes amount, dayOfMonth, and and returns a bool.

In the cpp file, the implementation of the new deposit method should call the old deposit method. (The compiler distinguishes methods based on the number of input parameters, so there is no problem with using the same method name for the two different methods.) In the implementation of the new withdraw, call the old withdraw method with just the amount. If the old withdraw returns true, then we will add a transaction to the list. If it returns false, then there is no transaction and we just return false.

To be safe, we should check to make sure that we don't exceed the size of the array. So the condition for withdrawing should be as follows:

```
if ((numEvents < MAX_TRANSACTIONS) && withdraw(amount)) {
```

In the transaction, the amount withdrawn should appear as a negative number. Here is the code for adding a transaction object to the events array.

```
events[numEvents] = new Transaction(dayOfMonth, -amount, description);
numEvents++;
```

There are two things to note about this code. The first is about the use of numEvents as the index. If you have an array with 5 elements, then 5 is the number of elements and they are numbered 0 to 4. The next element, if we add one, will be element number 5. That means that if you are going to add another element, the index of the next element is the same as the number of elements currently in the list. That is why we use numEvents as the index. Then we have to increment the number of events since we just added 1.

The second thing to note is the use of the word "new". Since our list is a list of references, we have to create an object to which we can refer. We do that using the word "new" followed by the name of the class. The expression with "new" creates a new object and returns its reference.

Step 7

Do the same for the new deposit method, but with two differences. First, since the description will always be "deposit", there is no need to pass it as an input parameter. The new deposit method should take only two input parameters, the date and the amount. Second, when we create the transaction, the amount being deposited should remain positive.

```
        events[numEvents] = new Transaction(dayOfMonth, amount, "deposit");
```

Step 8

We want the user of the Account class to only use the new versions of deposit() and withdraw(). To prevent others from using the old methods, while still having them available to use in our own implementation, edit the Account.h file and move the old versions of deposit() and withdraw() to the private part of the class. In other words, move them from after to before the word "public."

Step 9

Finally, we want to add a method to display a list of the transactions. Add the new method to the .h and .cpp files and call it "displayTransactions". Like the displayHolder() method, displayTransactions() takes no input parameters and returns nothing. The displayTransactions() method will loop through the events array, printing out all of the transactions in the array (up to numEvents).

In the implementation of displayTransactions(), iterate through the list of events using a "for" loop: `for (int i = 0; i < numEvents; i++)`. Inside the loop, we can display the transaction simply by calling its display method. Since events is an array of references to transactions, and not actual transactions, we must again use the -> notation instead of a dot:

```
        events[i]->display();
```

Step 10

To test the new code, we need to modify the code in main. At least for the time being, you can keep the old version of main by renaming it to "oldMain." Then put /* before and */ after the code so it doesn't get compiled. Now create a new version of main. Use the code below for the new main, and then replace Sponge Bob with your own name.

```
int main() {
    Customer person1("Sponge Bob");
    person1.setAddress("1600 Pennsylvania Avenue");
    Account account1;
    account1.setHolder(&person1);
    cout << account1.deposit(500, 1) << endl;
    cout << account1.withdraw(1000, 3, "ATM on Elm Street") << endl;
    cout << account1.withdraw(50, 3, "ATM on Elm Street") << endl;
    cout << account1.withdraw(67.43, 7, "Sprint Wireless") << endl;
    cout << account1.withdraw(8.99, 17, "Burger King") << endl;
    account1.displayHolder();
    account1.displayTransactions();
    cout << account1.getBalance() << endl;
    return 0;
}
```

The output should appear as follows (except the Sponge Bob was replaced).

```
1
0
1
1
1
```

```
Sponge Bob
1600 Pennsylvania Avenue
1  500  deposit
3  -50  ATM on Elm Street
7  -67.43  Sprint Wireless
17  -8.99  Burger King
373.58
Press any key to continue . . .
```

What to turn in:

Now we have three classes, so there are 7 .cpp and .h files to turn in. You may also want to make a copy of the output and include that in the zip file when you turn this assignment in.