COP 3223 Program #10: Bank Accounts (Structs)

Due Date: ***Please Consult WebCourses***

Objectives

1. To give students practice in reading input from files.
2. To give students practice utilizing structs.
3. To give students practice utilizing arrays of structs.

Problem: Bank Management

A typical bank account has both a checking account and a savings account. We can store the amount of money in an account in the following struct that was used in quiz 4:

```
struct money {
    int dollars;
    int cents;
};
```

We can use this struct to then define a struct for a bank account as follows:

```
#define MAXSIZE 20

struct bankaccount {
    char firstname[MAXSIZE+1];
    char lastname[MAXSIZE+1];
    int accountID;
    struct money checking;
    struct money savings;
};
```

For the purposes of this problem, the money stored in one's savings account can accrue interest over time, but the money in one's checking account can not.

Typical operations on a bank account include the following:

1. Depositing money into either the checking or savings account.
2. Withdrawing money from either of the two accounts.
3. Transferring money in between the two accounts (so either moving some money from checking to savings, or moving some money from savings to checking).
4. Adding interest earned to the balance of the savings acccount.
5. Printing your balance (amount of money in both of your accounts).

For this problem, you will be managing the accounts and operations on those accounts for a bank. You will read the input for the problem from a file (bankaccount.txt), and will output results to standard output. The file will contain a list of sequential operations.

It is guaranteed that there will never be more than 100 total unique bank customers.

<u>Input File Format</u>
The first line in the input file will contain a single positive integer, ***n*** (***n*** ≤ 10000), representing the number of lines of input that follow in the file. Each line of input that follows represents a single instruction.

The first item for every line will be the instruction number. This will be followed by several other items detailing that instruction.

Here is a listing of each instruction by number:

<u>Instruction #1: Creating a New Account</u>
If the instruction number is 1, this will be followed by the following pieces of information, all separated by spaces:

```
FNAME LNAME ACCTID INITCHECKING INITSAVINGS
```

where

`FNAME` indicates the first name of the person opening a new account,
`LNAME` indicates the last name of the person opening a new account,
`ACCTID` indicates the account ID assigned for this new account,
`INITCHECKING` indicates the initial amount of money in the checking account and
`INITSAVINGS` indicates the initial amount of money in the savings account.

Both `FNAME` and `LNAME` will be strings of letters with a length no more than 20.
`ACCTID` will be a 9 digit positive integer, and
both `INITCHECKING` and `INITSAVINGS` will be doubles written to exactly 2 decimal places in between 0 and 10000000.00.

Note: You won't store the money as doubles. A function will be provided that sets a struct money equal a given double that represents an amount of money.

Here is an example of this type of command:

```
1 Arup Guha 900123456 2100.46 5.00
```

This indicates that Arup Guha is opening a new account with ID number 900123456, with an initial value of $2100.46 in the checking account and $5.00 in the savings account.

It is guaranteed that any time this command is made that the account will be a new account with a unique identification number not belonging to anyone else. It's possible that two different people could have the same name but have separate account numbers. <u>This command will appear in the input a maximum of 100 times.</u>

Instruction #2: Depositing Money to an Account

If the instruction number is 2, this will be followed by the following pieces of information, all separated by spaces:

```
ACCTID ACCTTYPE AMOUNT
```

where

`ACCTID` indicates the account for the deposit,
`ACCTTYPE` is 1 to indicate the checking account or 2 to indicate the savings account
`AMOUNT` is the amount to be deposited, expressed to exactly 2 decimal places.

It is guaranteed that each of these instructions will be valid**. Namely, there will always use a valid existing account ID, the account type will always be 1 or 2, and the amount will be a positive value that won't overflow the corresponding struct money.**

Here is an example of an instruction with this format:

```
2 900123456 2 100.00
```

This indicates adding $100.00 to the savings account of the bank account with the account ID equal to 900123456. (If we executed this right after the example statement for Instruction #1, then Arup Guha's savings account would be updated to $105.00.)

Instruction #3: Withdrawing Money from an Account

If the instruction number is 3, this will be followed by the following pieces of information, all separated by spaces:

```
ACCTID ACCTTYPE AMOUNT
```

where

`ACCTID` indicates the account for the deposit,
`ACCTTYPE` is 1 to indicate the checking account or 2 to indicate the savings account
`AMOUNT` is the amount to be deposited, expressed to exactly 2 decimal places.

It is guaranteed that the account ID will exist, the account type will always be 1 or 2, and that the amount given will be positive and less than 10000000.00.

However, it's possible that the amount listed might be more than the amount available in the desired account. In these cases, the transaction requested will be denied and no change will be made to the account.

In all other cases, the withdrawal should be made.

For example,

```
3 900123456 1 1000.00
```

indicates to withdraw $1000.00 from the checking account of the bank account with the account ID equal to 900123456. If we were to do this instruction immediately following the two previous sample instructions, this instruction would be executed and then Arup Guha's checking account would have $1100.46 remaining in it.

If, we subsequently tried this instruction:

```
3 900123456 2 1000.00
```

it would NOT be executed because Arup Guha has less than $200.00 in his savings account! (Eeek!!!)

<u>Instruction #4: Transferring Money Between Accounts in the Same Bank Account</u>
If the instruction number is 4, this will be followed by the following pieces of information, all separated by spaces:

```
ACCTID ACCTTYPEFROM AMOUNT
```

where

`ACCTID` indicates the account for the deposit,
`ACCTTYPEFROM` is 1 to indicate a transfer from checking to savings, and 2 to indicate a transfer from savings to checking.
`AMOUNT` is the amount to be transferred, expressed to exactly 2 decimal places.

<u>It is guaranteed that the account ID will exist, the account type will always be 1 or 2, and that the amount given will be positive and less than 10000000.00.</u>

<u>However, it's possible that the amount listed might be more than the amount available in the desired account.</u> In these cases, the transaction requested will be denied and no change will be made to the account.

In all other cases, the transfer should be made.

For example, the instruction

```
4 900123456 1 500.00
```

were attempted right after the previous sample instructions, the instruction would be executed because Arup Guha has at least $500.00 in his checking account. After the instruction, his new account balances would be $600.46 in checking and $605.00 in savings.

A subsequent instruction of

```
4 900123456 2 2000.00
```

would not be processed because Arup Guha doesn't have $2000.00 in his savings account.

Instruction #5: Adding Interest to all Savings Accounts
If the instruction number is 5, this will be followed by one more piece of information, separated by a space:

```
PERC
```

representing the percentage increase for all savings accounts. This value will be a real number written to exactly 1 decimal place, in between 0.1 and 5.0. It will be guaranteed that none of these operations will ever cause an overflow error. **(In particular, no individual account will ever exceed 10 million dollars.)**

This update will occur to ALL active back accounts.

To properly execute this instruction, do the following:

1. Calculate the total cents in the relevant struct money.
2. Multiply this value by 1 + perc/100, where perc is the percentage increase for all accounts.
3. Add EPS (0.000000001) to the value from step 2.
4. Cast the value from step 3 to an integer.
5. The value in step 4 is the total number of cents for the updated struct money. Use integer division by 100 and mod by 100 to update the struct money.

For example, the instruction

```
5 1.0
```

would increase the value of each savings account by 1%. (For example, Arup Guha's savings account would change from $605.00 to $611.05.)

Instruction #6: Printing an Account Balance

If the instruction number is 6, this will be followed by one more piece of information, separated by a space:

```
ACCTID
```

representing the account ID for which the request is being made.

For example, the instruction

```
6 900123456
```

represents a request for the account information for Arup Guha.

Each of these requests will only be made on valid accounts.

Output Format

The only instruction for which output will be provided is Instruction #6. For these instructions, output a single line with the following format:

```
FNAME LNAME ACCTID $CHECKING $SAVINGS
```

where each item indicates the same thing it does as described for the first input instruction.

For example, for the last sample instruction given in the input, the corresponding output would be

```
Arup Guha 900123456 $600.46 $611.05
```

Sample Input/Output

Will be provided on the course web page.

Implementation Requirements
Use the following constants:

```
#define MAXSIZE 20
#define MAXCUSTOMERS 100
#define EPS 0.000000001
```

The first is the maximum length of any of the input strings, the second is the maximum number of customers and the third is used for accurate truncation.

You must use the following function to store an amount of money represented in a double in a struct money variable:

```
void setMoney(struct money* ptrMoney, double amt) {

    // Add epsilon to counteract rounding errors.
    int totalCents = (int)(100*amt + EPS);

    // Now assign both components.
    ptrMoney->dollars = totalCents/100;
    ptrMoney->cents = totalCents%100;

}
```

Use this function to print a struct money:

```
void printMoney(const struct money* ptrMoney) {
    printf("$%d.%02d", ptrMoney->dollars, ptrMoney->cents);
}
```

You will be required to have a function that takes in an integer account number, an array of bank accounts, and the number of active accounts and returns the index at which the account with the given account number is stored. Its function prototype should look like this:

```
int getAccountIndex(struct bankaccount allAccounts[], int
numAccounts, int id);
```

if the listed account is not in the array in between index 0 and numAccounts-1, the function should return -1. Otherwise, as previously stated, the function should return the index into the array at which the account with the account number accountID is stored.

Grading Details

Your grade will mostly be based on correctness and following the required implementation requirements since this is the last assignment and the graders have less time to grade it. There will still be 10 style points though.

Deliverables

Please submit a single .c file called, banking.c. Please make sure to fully comment your code, including both a header comment and internal comments.