University of Washington Bothell

CSS 342: Data Structures, Algorithms, and Discrete Mathematics I

Program 5: The Jolly Banker

Purpose

This lab will serve a few purposes. First, it will provide hands-on experience using both queues and binary search trees. It will also provide an opportunity for further program/class design as the project does not specifically delineate class structure or design.

There are aspects of the spec below which allow interpretation—please read, design and ask questions early to clarify any ambiguity.

A design will be required to be turned in before you begin coding. *This design will be turned into canvas and graded.* Your final delivery can differ from the design you initially make.

Problem Overview:

You will build a banking application which processes transactions. This banking application consists of three phases.

1) The program will read in a string of transactions from a file into an in-memory queue. These transactions can open accounts, withdraw funds, deposit funds, transfer funds, or ask for the transactional history to be printed.

2) The program will next read from the queue and process the transactions in order.

3) When the queue has been depleted the program will print out all open accounts and balances in those accounts.

Details:

Input:

To test your program a file will be passed *in by argument to the program on the command line*. The file will contain a list of transactions that need to be executed. Transactions of the format described below (see section on transactions) will be contained in this file. There will be one transaction per line.

Assume that the input is well-formed in the file—that is, there are no syntax errors. That said, there may be errors in the transactions themselves. For instance, a transaction may try to withdraw more money than there is a fund or try to withdraw for a non-existent fund. See the section below entitled errors for details.

Client Accounts and Funds:

Each client account contains assets held in up to ten funds. A client account is represented by a first and last name (two strings) and a unique id (integer).

The ten funds that each account has are as follows:

```
0: Money Market           5: Value Fund
1: Prime Money Market     6: Growth Equity Fund
2: Long-Term Bond         7: Growth Index Fund
3: Short-Term Bond        8: Crypto ETF
4: S+P Index Fund         9. Precious Metals ETF
```

All client accounts are opened with an open transaction, O (see below).

Transactions:

There are five types of transactions, and each are identified by a character beginning the line.

O: Open a client account and instantiate the ten funds. All funds start with $0 except for one chosen fund which gets seeded with $250

D: Deposit assets into a fund

W: Withdraw assets from a fund

T: Transfer assets between funds (can be funds owned by a single client or transfers between clients)

H: Display the history of all transactions for a client account

F: Display the history of all transactions of a given fund

The format for the commands consists of a transaction type (above), followed by an account id, fund id, and amount. For transfers the ***from*** account and fund comes before ***to*** account and fund.

Examples:

D 1234 1 100	Deposit $100 into the prime money market account of client ID 1234
W 1234 0 500	Withdraw $500 from the money market of client ID 1234.
T 1234 0 1234 1 1000	Transfer $1000 from client 1234's money market to the prime money market.
T 1234 0 5678 0 1000	Transfer $1000 from 1234's money market to 5678's money market.
H 1234	Display the history of all transactions of all accounts for client 1234.
F 1234 4	Display the history for all transactions on the S+P Index Fund for client 1234
O 6537 1 Bowden C.	Open an account for client C. Bowden. Use account id 6537. Fund 1 gets the $250 starting bonus

Errors:

As noted above, assume the format (number and types of input items) are correct. However, the program should deal with all other types of semantic errors. For instance, there could be a bad account number (for instance, one already used) or an amount which is too much to withdraw.

Assume that each line will begin with an appropriate letter: O, W, D, T, H, and F.

Examples of errors which may occur:

W 6543 6 10000 (when the Gowth Equity fund of client 6543 has only $20)
D 7654 76 1000

A transaction that would cause a balance to become negative should not occur and is an error (a withdrawal or transfer of $0 is fine). **There is one exception to this rule:** if one is withdrawing or transferring assets from a money market fund with insufficient dollars, and it can be covered with dollars from the other money market fund owned by the client account, the desired amount (only the partial amount needed to cover the withdrawal) is moved to that money market account. The two Bond accounts are handled similarly. No other types of accounts are handled in this manner.

If one is transferring between two linked funds (say, funds 0 and 1) then the transaction should only succeed if the fund being withdrawn from has enough funds.

Appropriate error messages should be printed out to cerr. No other messages should be printed out during phase 1 or phase 2.

Output:

In Phase 3, each client account will be printed out with the amount held in each account. Please make sure to create an intuitive and readable output for this aspect of the program. Ideally it will look similarly to the format that I provide in the example output.

Data Structures:

One key aspect of this Lab exercise is the right class design for handling the problem. There are no pre-defined classes or signature structure for the classes required. However, it should follow best OO practices. The suggestion is to keep the balances as **ints** (**doubles** or **floats** are not required). This is up to you to define. However, there are two data structures which are required.

First, you need to use a queue to read the transactions during phase 1. All transactions should be read before processing starts. The queue can be the STL queue.

Second, the client accounts should be stored in a binary search tree (BSTree).

Here is a possible header file for the BSTree. I will review the code in the BSTree but not test it individually. All testing will be done on the entire Bank. I expect there to be a delete operation in the BSTree even though one is not required for the program to work correctly.

```
class BSTree
{
public:
	BSTree();
	BSTree(const BSTree& tree);
	~BSTree();

	bool Insert(Account *account);

	// retrieve object, first parameter is the ID of the account
	// second parameter holds pointer to found object, NULL if not found
	bool Retrieve(const int& account_id, Account* & account) const;

	// Delete object, first parameter is the ID of the account
	// second parameter holds pointer to found object, NULL if not found
	bool Delete(const int& account_id, Account* & account);

	BSTree& operator=(const BSTree& tree);


	// displays the contents of a tree to cout;  you could also overload
operator<<
	void Display() const;
	int Size() const;

private:
	struct Node
	{
		Account* p_acct;
		Node* right;
		Node* left;
	};
	Node* root_;
};
```

What to bring to Checkpoint peer design review

We will use class time to review and discuss your design before you turn it in. The class will break into groups of three or four and discuss the program design with each other. The more you have prepared for the discussion the more you will benefit. To prepare for this

discussion, I would suggest you come with as much of the design as possible. Note that the intent is to have this discussion inform your design. Your design will likely change after the discussion.

This initial design you turn in should clearly show the different Classes/Data Structures you will use and how they interact. You can do this with some class diagrams, .h files, and a page or two description. Visuals are very beneficial to help in your explanation and should be included.

Sample Input/Output will be placed on Canvas soon.

See files: bank_trans_in.txt and bank_trans_out.txt.

What to turn in final project

As everyone will have different factoring (Classes), please make sure to follow very closely what needs to be turned in. This is critical to get the correct grade for your project.

Here is what should be turned in:

- All .cpp and .h files required to make your project work
- An output file called: bank_trans_out.txt. This is Output of your program running the sample input found on Canvas. Note that the output from your program will not look exactly like the output from my program but hopefully it will be close.
- A set of test inputs which you created to test your implementation.
- The grader and I will build your code and run against new test cases copying the .h and .cpp files on the LINUX lab
    - g++ *.cpp -o jolly_banker
    - jolly_banker input_testfile.txt