**Enhanced Requirement Implementation**

Credit Cards

When designing the program and deciding how to differentiate between chequings and savings accounts, I decided to distinguish them through a variable, type. This choice allowed for much easier addition of other type of accounts with minimal effort. Rather than creating a new class or setting up an inheritance scheme, an integer in the object would dictate how an account was displayed and handled.

To create credit cards, all that was needed was an additional spot in the accounts array in users and to add handling in main.cpp for accounts with type 3, ie credit cards. By checking the type of the account in question, different options or displays could be easily made. This also goes for functions such as deposit, which can quickly determine whether to debit or credit the amount in question.

From a usage point, I modelled credit cards off of a typical real life card: ‘deposits’ consist of payment that decrease balance on the card, as do transfers, and you are unable to withdraw cash from a credit account. This also worked well with the interest system, where interest on owed balance would accrue.

In all, this approach allowed for minimal adjustment to base features to include a new account with new functionality. Save for adding in more conditional statements to check if the account is a credit card, this feature was able to be implemented with no hassle.

Cross User Transfers

This feature was implemented by extending the base requirement for transfers. Since 3 types of accounts made it necessary to specify which particular accounts were being transferred to and from, the transfer function was extended to take the address of a separate account. This allowed non user-owned accounts to be passed into the transfer.

To distinguish this capability, displays were modified to include the new option and follow a few more steps. Once an external account was indicated as the target, the user has to specify the unique account ID that they want to transfer to. For extra security purposes, a confirmation of their PIN was added.

Using a pointer in the function allowed objects located elsewhere in memory to be modified. This new addition was simple to include as it only entailed modifying an underlying function to take a pointer and then displaying proper options on screen.

Interest Payments

Similar to real life banks, interest on account balances were added. To implement this feature, a specific rate for each account had to be added into the objects and stored, as well as a function to update balances based on these rates.

To support this, a float was included as a mandatory parameter for account constructors. This ensured each account has a rate, even if it 0. Next, the ability to update these balances had to be implemented. Rather than use a traditional timing system that would accrue interest automatically at the end of the month, this function was set as an additional option for managers. Since this program will not be ran continually over long lengths of time, it made more sense to allow a manager to call the function when they want to accrue interest in the accounts.

The function itself is simple: Iterate through the account list and update each object pointed to by the iterator to increase its balance by r%. Pointers were used so that the original object itself would be changed.

Log File Reading

I felt there wasn’t much point to maintenance staff only being able to toggle on trace execution. After looking through some of the output files it was clear that it would be useful to read them into the system to analyze.

The function was easily implemented through file io and standard library. Reading in the file was done line by line and allowed for easy formatting by inserting \n into the strings when needed.

The actual opening of the file is contained within a try-catch statement, where the catch will catch any exceptions thrown during file opening or reading. This helps keep this program running if a mistake is made or there is a bad file input.