Andrew Taylor

4/3/25

COMS-280-WWW01

# Banking System – Phase 3

## Link

<https://github.com/andrewrtaylor1/BankProj>

## General Linked List Design

The Linked Lists in my Banking System are generic & doubly linked, utilizing shared pointers (a smart pointer implementation in std from newer versions of C++). They are used to store Customers, Account IDs in Customers, Employees, Accounts, Transaction History in the Accounts, & Encryption keys (not yet utilized/implemented). For required functions, it has: add (put), delete (drop), update, search (find & get). On top of that, I also have a safety check (exists), and count-related functions.

To effectively use the double linked nature & allow for indexed functions (easier implementations of get, delete, etc.) the lists keep a count of the elements, and use indexed traversal. This allows for generic external nodes on the head & tail, and internal nodes that more readily work in both directions. I am using pointers for the integers to save on memory & make counting easier.

I have also elected to use templates ahead of time in my LinkedList design. Both the list and nodes make use of them for typing of the data pointer. This has saved a lot of time for implementation & bug fixing but has made me make some compromises. My search function uses the compare function in std::string, and therefore classes inside the LinkedList need a compare function as well.

## Special Class Design

Overdraft & Interest are two classes housing public static functions for handling their namesakes. The former will attempt to fix negative balance, & purchases will trigger it on a per-user basis. Similarly, interest will be checked at an interval & respect each account’s particular interest rate, compounding, etc. & handle them accordingly. This doesn’t do much yet, but the account base class will declare the interest class as a friend. Much of the current design is public; this is obviously not ideal for security reasons, encapsulation, etc. but allows for fast prototyping. This will not be kept in the final version of the project, many of the account members will be protected, including interest information that the Interest functions need.

Overdraft::OnPurchase is a function called at the end of the database’s function processing class, to attempt to correct any negative balances within a user’s accounts. It uses the baked in transfer function of the user for the transfers. The process is rather intensive, but there’s not really any other way to go about it.

Interest::payout is a function used to simplify the payout step, taking a rate & ratio derived from another function. This could also be used to do unique payouts, like a certificate of deposit account.

Interest::IndividualAccount is a function for, as the name suggests, each individual interest processing. This could be called on its own, but will most often be called by the next function. This will also call the payout function for accounts with normal interest behaviors (interval payouts, not Certificate of Deposit)

Interest::AllAccounts takes the linked lists of accounts & calls the IndividualAccount function for each of them, handling interest across the database.

## Challenges & Solutions

Finding the right smart pointer & getting used to using them was a challenge throughout implementation. I eventually settled on shared pointers for how I wanted to use them. For a standard pointer, it’s easy enough to pass it around like an ordinary variable. This does not work for, say, a naive implementation of smart pointer. You must be very careful with your destructors & constructors, at a level that it is not reasonable to implement during this class. I couldn’t use the other smart pointer types supplied by the standard library either, unique pointer couldn’t be passed around the same & the shared pointer logic is nicer than a weak pointer.

I also had a bit of trouble with basic list logic at first, but luckily that was mostly due to typos and how inheritance/derivatives work in C++. I would frequently forget that you sometimes must invoke the base class to refer to members of it. Switching to accessor functions helped me avoid these errors.

## Reflection

I was familiar with the concept of a List from other programming languages, but this is my first time (outside of the other C++ project) properly getting in and implementing a list, particularly a Linked List. It has been eye opening, seeing the “how” & “why” of design firsthand. How does one make it dynamic, what kind of bugs do you have to deal with, what are the limitations, etc. It was very rewarding seeing it work with no issues once all the major bugs were dealt with.

I particularly appreciate the way a doubly linked list can work. With a bit of an overhead by doubling the number of pointers needed (& obviously more complex code), you can make the actual traversal of the list more efficient. With each attempt to optimize, the cost could be worth less than the benefit; however, making the bank system more efficient at scale instead of upfront seems beneficial.

Special classes & their functionality are still a bit confusing to me, if just in the form of “what even *is* a special class?” However, implementing them was relatively trivial. I do not have much to say on them ultimately, but special functionality in classes is another useful tool in the programmer’s toolkit. I also do not have much need for the “friend” functionality at present, but that is a sign that too much of my codebase is public for now.

## Relevant Code

Link.h

#pragma once

#include <string>

#include <memory>

/// <summary>

/// Abstract Node class for Linked List.

/// </summary>

template <typename T>

class Node

{

private:

//these are made private because it forces the Node::set/get functions which work better

std::shared\_ptr<Node<T>> next; //next Node

std::shared\_ptr<Node<T>> previous; //previous Node

public:

Node(std::shared\_ptr<Node<T>> n, std::shared\_ptr<Node<T>> p)

//constructor; as this is abstract & never constructed directly, no need for a default. that'll be in the derived classes

{

next = n;

previous = p;

}

void setNext(std::shared\_ptr<Node<T>> n) //function to set Next

{

next = n;

}

void setPrevious(std::shared\_ptr<Node<T>> p) //function to set Previous

{

previous = p;

}

std::shared\_ptr<Node<T>> getNext() //gets Next

{

return next;

}

std::shared\_ptr<Node<T>> getPrevious() //gets Previous

{

return previous;

}

virtual ~Node() {}

//these functions have to be virtual, as only InternalNode will have the data pointer.

//Inserts the data at a specific spot. i = desired index, j = current index

virtual bool put(std::shared\_ptr<int> i, std::shared\_ptr<int> j, std::shared\_ptr<T> d) = 0;

//Grabs a pointer to the data at a given index, if it exists. Returns nullptr if it doesn't. i = desired index, j = current index

virtual std::shared\_ptr<T> get(std::shared\_ptr<int> i, std::shared\_ptr<int> j) = 0;

//moves through the list in an indexed way, returns bool if we get to the desired index & Internal Node exists. i = desired index, j = current index

virtual bool exists(std::shared\_ptr<int> i, std::shared\_ptr<int> j) = 0;

//finds a string, has to be exact. I'm not going to use C++23 just to get a Contains function I don't need, & it'd break other things

virtual int find(std::string s, std::shared\_ptr<int> j) = 0;

//deletes the node, done by simply removing all references to it

virtual bool drop(std::shared\_ptr<int> i, std::shared\_ptr<int> j) = 0;

//updates the data at a specific index

virtual bool update(std::shared\_ptr<int> i, std::shared\_ptr<int> j, std::shared\_ptr<T> d) = 0;

virtual int count(int i) = 0;

};

/// <summary>

/// Internal node, actually points to data

/// </summary>

template <typename T>

class InternalNode : public Node<T>

{

public:

std::shared\_ptr<T> data; //pointer to the data

InternalNode(std::shared\_ptr<T> d, std::shared\_ptr<Node<T>> n = std::shared\_ptr<Node<T>>(), std::shared\_ptr<Node<T>> p = std::shared\_ptr<Node<T>>()) : Node<T>(n,p)

//constructor will always have data pointer

{

data = d; //set data pointer

}

~InternalNode() {}

/// <summary>

/// Check if a specific node exists. Only internal nodes count

/// </summary>

/// <param name="i">desired index</param>

/// <param name="j">current index</param>

/// <returns>whether or not the node exists, bool</returns>

bool exists(std::shared\_ptr<int> i, std::shared\_ptr<int> j)

{

bool b = false;

if (\*i > \*j)

{

\*j = \*j + 1;

b = Node<T>::getNext()->exists(i, j);

}

if (\*i < \*j)

{

\*j = \*j - 1;

b = Node<T>::getPrevious()->exists(i, j);

}

if (\*i == \*j)

{

b = true;

}

return b;

}

/// <summary>

/// find a string in a node

/// </summary>

/// <param name="s">string to be found</param>

/// <param name="j">current index</param>

/// <returns>specific index of string, int</returns>

int find(std::string s, std::shared\_ptr<int> j)

{

if (data->compare(s) == 0) //using compare function, must add compare function to search

{

return \*j; //dereference the pointer & return it

}

if (Node<T>::getNext())

{

\*j = \*j + 1; //add one to current index

return Node<T>::getNext()->find(s, j); //get the next node's find function

}

else

{

return -1; //return negative one, error/doesn't exist

}

}

/// <summary>

/// grabs the data from the specific index

/// </summary>

/// <param name="i">desired index</param>

/// <param name="j">current index</param>

/// <returns>returns a pointer to the data</returns>

std::shared\_ptr<T> get(std::shared\_ptr<int> i, std::shared\_ptr<int> j)

{

if (\*i > \*j)

{

\*j = \*j + 1;

return Node<T>::getNext()->get(i, j);

}

if (\*i < \*j)

{

\*j = \*j - 1;

return Node<T>::getPrevious()->get(i, j);

}

if (\*i == \*j)

{

return data;

}

return std::shared\_ptr<T>();

}

/// <summary>

/// put data at a specific index

/// </summary>

/// <param name="i">desired index</param>

/// <param name="j">current index</param>

/// <param name="d">data to be placed</param>

/// <returns>was it successful? bool</returns>

bool put(std::shared\_ptr<int> i, std::shared\_ptr<int> j, std::shared\_ptr<T> d)

{

bool b = false;

if (\*i > \*j)

{

\*j = \*j + 1;

b = Node<T>::getNext()->put(i, j, d);

}

if (\*i < \*j)

{

\*j = \*j - 1;

b = Node<T>::getPrevious()->put(i, j, d);

}

if (\*i == \*j)

{

std::shared\_ptr<Node<T>> n(new InternalNode<T>(d));

Node<T>::getPrevious()->setNext(n);

Node<T>::getPrevious() = n;

}

return b;

}

/// <summary>

/// finds the node to delete; if it's this one, use the dedicated private delete function

/// </summary>

/// <param name="i">desired index</param>

/// <param name="j">current index</param>

/// <returns>was it successful? bool</returns>

bool drop(std::shared\_ptr<int> i, std::shared\_ptr<int> j)

{

bool b = false;

if (\*i > \*j)

{

\*j = \*j + 1;

b = Node<T>::getNext()->drop(i, j);

}

if (\*i < \*j)

{

\*j = \*j - 1;

b = Node<T>::getPrevious()->drop(i, j);

}

if (\*i == \*j)

{

b = drop();

}

return b;

}

/// <summary>

/// update data at a specific index

/// </summary>

/// <param name="i">desired index</param>

/// <param name="j">current index</param>

/// <param name="d">data to be placed</param>

/// <returns>was it successful? bool</returns>

bool update(std::shared\_ptr<int> i, std::shared\_ptr<int> j, std::shared\_ptr<T> d)

{

bool b = false;

if (\*i > \*j)

{

\*j = \*j + 1;

b = Node<T>::getNext()->update(i, j, d);

}

if (\*i < \*j)

{

\*j = \*j - 1;

b = Node<T>::getPrevious()->update(i, j, d);

}

if (\*i == \*j)

{

b = true;

data = d;

}

return b;

}

/// <summary>

/// simple function to count each node

/// </summary>

/// <param name="i">int of count before</param>

/// <returns>current count, int</returns>

int count(int i)

{

return Node<T>::getNext()->count(i + 1);

}

private:

/// <summary>

/// Just simply deletes the current node

/// </summary>

/// <returns>was it successful? bool</returns>

bool drop()

{

Node<T>::getPrevious()->setNext(Node<T>::getNext());

Node<T>::getNext()->setPrevious(Node<T>::getPrevious());

return true;

}

};

/// <summary>

/// external nodes; works as tail or head depending on direction

/// </summary>

template <typename T>

class ExternalNode : public Node<T>

{

public:

ExternalNode(std::shared\_ptr<Node<T>> n = std::shared\_ptr<Node<T>>(), std::shared\_ptr<Node<T>> p = std::shared\_ptr<Node<T>>()) : Node<T>(n, p) {}

~ExternalNode() {}

/// <summary>

/// Check if a specific node exists. Only internal nodes count

/// </summary>

/// <param name="i">desired index</param>

/// <param name="j">current index</param>

/// <returns>whether or not the node exists, bool</returns>

bool exists(std::shared\_ptr<int> i, std::shared\_ptr<int> j)

{

bool b = false;

if (\*i > \*j)

{

\*j = \*j + 1;

return Node<T>::getNext()->exists(i, j);

}

if (\*i < \*j)

{

\*j = \*j - 1;

return Node<T>::getPrevious()->exists(i, j);

}

return b;

}

/// <summary>

/// find a string in a node

/// </summary>

/// <param name="s">string to be found</param>

/// <param name="j">current index</param>

/// <returns>specific index of string, int</returns>

int find(std::string s, std::shared\_ptr<int> j)

{

if (Node<T>::getNext())

{

\*j = \*j + 1;

return Node<T>::getNext()-> find(s, j);

}

else

{

return -1;

}

}

/// <summary>

/// grabs the data from the specific index

/// </summary>

/// <param name="i">desired index</param>

/// <param name="j">current index</param>

/// <returns>returns a pointer to the data</returns>

std::shared\_ptr<T> get(std::shared\_ptr<int> i, std::shared\_ptr<int> j)

{

if (\*i > \*j)

{

\*j = \*j + 1;

return Node<T>::getNext()->get(i, j);

}

if (\*i < \*j)

{

\*j= \*j -1;

return Node<T>::getPrevious()->get(i, j);

}

return std::shared\_ptr<T>();

}

/// <summary>

/// put data at a specific index

/// </summary>

/// <param name="i">desired index</param>

/// <param name="j">current index</param>

/// <param name="d">data to be placed</param>

/// <returns>was it successful? bool</returns>

bool put(std::shared\_ptr<int> i, std::shared\_ptr<int> j, std::shared\_ptr<T> d)

{

bool b = false;

if (\*i > \*j)

{

\*j = \*j + 1;

if (!Node<T>::getNext())

{

b = true;

std::shared\_ptr<Node<T>> n(new InternalNode<T>(d, Node<T>::getPrevious()->getNext(), Node<T>::getPrevious()));

Node<T>::getPrevious()->setNext(n);

Node<T>::setPrevious(n);

}

else

{

b = Node<T>::getNext()->put(i, j, d);

}

}

if (\*i < \*j)

{

\*j = \*j - 1;

if (!Node<T>::getPrevious())

{

b = true;

std::shared\_ptr<Node<T>> n(new InternalNode<T>(d, Node<T>::getNext()->getPrevious(), Node<T>::getNext()));

Node<T>::getNext()->setPrevious(n);

Node<T>::setNext(n);

}

else

{

b = Node<T>::getPrevious()->put(i, j, d);

}

}

return b;

}

/// <summary>

/// deletes data at node; checking is done outside of node. For an external node, this does nothing but traversal

/// </summary>

/// <returns>was it successful? bool </returns>

bool drop(std::shared\_ptr<int> i, std::shared\_ptr<int> j)

{

bool b = false;

if (\*i > \*j)

{

\*j = \*j + 1;

return Node<T>::getNext()->drop(i, j);

}

if (\*i < \*j)

{

\*j = \*j - 1;

return Node<T>::getPrevious()->drop(i, j);

}

return b;

}

/// <summary>

/// update data at a specific index

/// </summary>

/// <param name="i">desired index</param>

/// <param name="j">current index</param>

/// <param name="d">data to be placed</param>

/// <returns>was it successful? bool</returns>

bool update(std::shared\_ptr<int> i, std::shared\_ptr<int> j, std::shared\_ptr<T> d)

{

bool b = false;

if (\*i > \*j && Node<T>::getNext())

{

\*j = \*j + 1;

b = Node<T>::getNext()->update(i, j, d);

}

if (\*i < \*j && Node<T>::getPrevious())

{

\*j = \*j - 1;

b = Node<T>::getPrevious()->update(i, j, d);

}

return b;

}

/// <summary>

/// simple function to count each node

/// </summary>

/// <param name="i">int of count before</param>

/// <returns>current count, int</returns>

int count(int i)

{

//if we have a next node (head node)

if (Node<T>::getNext())

{

//start going through nodes

return Node<T>::getNext()->count(i);

}

else

{

//if we are at the tail node, just return

return i;

}

}

};

/// <summary>

/// Doubly linked list, utilizing smart pointers & templates

/// </summary>

template <typename T>

class LinkedList

{

private:

std::shared\_ptr<Node<T>> head; //head pointer

std::shared\_ptr<Node<T>> tail; //tail pointer

int count = 0; //last count of internal nodes; updated via operations

public:

LinkedList() //constructor

{

head = std::shared\_ptr<Node<T>>(new ExternalNode<T>(std::shared\_ptr<Node<T>>(new ExternalNode<T>()))); //initialize head pointer, create tail pointer

tail = head->getNext(); //grab tail pointer

tail->setPrevious(head);

}

LinkedList(std::shared\_ptr<T> d) //constructor

{

head = std::shared\_ptr<Node<T>>(new ExternalNode<T>(std::shared\_ptr<Node<T>>(new ExternalNode<T>()))); //initialize head pointer, create tail pointer

tail = head->getNext(); //grab tail pointer

tail->setPrevious(head);

put(d);

}

/// <summary>

/// checks the existence of an internal node at specific index

/// </summary>

/// <param name="i">desired index</param>

/// <returns>does it exist, bool</returns>

bool exists(int i)

{

bool b = false;

std::shared\_ptr<int> iPointer = std::shared\_ptr<int>(new int(i));

std::shared\_ptr<int> j = std::shared\_ptr<int>(new int(0));

//switching logic; if the index is higher than half the count, we start from the back

if (i > (count / 2))

{

\*j = count;

b = tail->exists(iPointer, j);

}

else

{

b = head->exists(iPointer, j);

}

return b;

}

/// <summary>

/// finds the index for a specific string

/// </summary>

/// <param name="s">string to find</param>

/// <returns>index, -1 if not in list, int</returns>

int find(std::string s)

{

std::shared\_ptr<int> j(new int(0));

return head->find(s, j);

return -1;

}

/// <summary>

/// gets pointer to data from specific index

/// </summary>

/// <param name="i">desired index</param>

/// <returns>pointer to data</returns>

std::shared\_ptr<T> get(int i)

{

std::shared\_ptr<int> iPointer = std::shared\_ptr<int>(new int(i));

std::shared\_ptr<int> j = std::shared\_ptr<int>(new int(0));

//switching logic; if the index is higher than half the count, we start from the back

if (i > (count / 2))

{

\*j = count+1;

return tail->get(iPointer, j);

}

else

{

return head->get(iPointer, j);

}

return std::shared\_ptr<T>();

}

/// <summary>

/// get function using the string

/// </summary>

/// <param name="s">string to find</param>

/// <returns>pointer to data, null if it can't be found</returns>

std::shared\_ptr<T> get(std::string s)

{

return get(find(s));

}

/// <summary>

/// put at specific index

/// </summary>

/// <param name="d">pointer to data</param>

/// <param name="i">desired index</param>

/// <returns>was it successful, bool</returns>

bool put(std::shared\_ptr<T> d, int i)

{

bool b = false;

std::shared\_ptr<int> iPointer = std::shared\_ptr<int>(new int(i));

std::shared\_ptr<int> j = std::shared\_ptr<int>(new int(0));

//switching logic; if the index is higher than half the count, we start from the back

if (i > (count / 2))

{

\*j = count;

b = tail->put(iPointer, j, d);

}

else

{

b = head->put(iPointer, j, d);

}

if (b) count++;

return b;

}

/// <summary>

/// put, unindexed

/// </summary>

/// <param name="d">data to put, pointer</param>

/// <returns>was successful, bool</returns>

bool put(std::shared\_ptr<T> d)

{

bool b = false;

int i = count+1;

b = put(d, i);

return b;

}

/// <summary>

/// Drop (delete) an entry

/// </summary>

/// <param name="i">desired index</param>

/// <returns>success, bool</returns>

bool drop(int i)

{

bool b = false;

//just using exist code again; traversing through the list already proves whether or not the index exists

std::shared\_ptr<int> iPointer = std::shared\_ptr<int>(new int(i));

std::shared\_ptr<int> j = std::shared\_ptr<int>(new int(0));

//switching logic; if the index is higher than half the count, we start from the back

if (i > (count / 2))

{

\*j = count;

b = tail->drop(iPointer, j);

}

else

{

b = head->drop(iPointer, j);

}

return b;

}

/// <summary>

/// Drop (delete) an entry, based on string

/// </summary>

/// <param name="i">desired string</param>

/// <returns>success, bool</returns>

bool drop(std::string s)

{

int i = find(s);

return drop(i);

}

/// <summary>

/// update data at a given index

/// </summary>

/// <param name="i">desired index</param>

/// <param name="d">data to update</param>

/// <returns>successful? bool</returns>

bool update(int i, std::shared\_ptr<T> d)

{

bool b = false;

//just using exist code again; traversing through the list already proves whether or not the index exists

std::shared\_ptr<int> iPointer = std::shared\_ptr<int>(new int(i));

std::shared\_ptr<int> j = std::shared\_ptr<int>(new int(0));

//switching logic; if the index is higher than half the count, we start from the back

if (i > (count / 2))

{

\*j = count;

b = tail->update(iPointer, j, d);

}

else

{

b = head->update(iPointer, j, d);

}

return b;

}

/// <summary>

/// update data with a given string value

/// </summary>

/// <param name="s">desired string</param>

/// <param name="d">data to update</param>

/// <returns>successful? bool</returns>

bool update(std::string s, std::shared\_ptr<T> d)

{

int i = find(s);

return update(i, d);

}

/// <summary>

/// does a quick count of InternalNodes

/// </summary>

void updateCount()

{

count = head->count(0);

}

/// <summary>

/// get the count

/// </summary>

/// <returns>count, int</returns>

int getCount()

{

return count;

}

};

BankDB.h (kept to only relevant code)

/// <summary>

/// contains static functions for overdraft

/// </summary>

class Overdraft

{

public:

/// <summary>

/// Handles overdraft for a specific user

/// </summary>

/// <param name="c">Customer shared pointer</param>

/// <returns>was successful?, bool</returns>

static bool OnPurchase(std::shared\_ptr<Customer> cust, std::shared\_ptr<Database> d);

};

/// <summary>

/// contains static functions for interest

/// </summary>

class Interest

{

public:

/// <summary>

/// payout function, used to simplify code

/// </summary>

/// <param name="acc">account to do interest on</param>

static void payout(std::shared\_ptr<Account> acc, double rate, double ratio)

{

USDollar pay(rate \* ratio); //get pay from rate \* ratio

if (pay < 1) return; //if pay is 0, just stop

std::shared\_ptr<BankFunction> trans(new BankFunction(pay, "Interest payout")); //create new transaction

acc->processTransaction(trans); //send new transaction to account

acc->LastPayout = std::chrono::system\_clock::now(); //reset last payout to now

}

/// <summary>

/// handles interest per account

/// </summary>

/// <param name="acc">account to have interest updated</param>

static void IndividualAccount(std::shared\_ptr<Account> acc)

{

//if account exists & is not a null pointer

if (acc)

{

//get values for ease of use from here on

int t = acc->interestType;

int p = acc->payoutRate;

//get time values in hours

int interestTime = std::chrono::duration\_cast<std::chrono::hours>(std::chrono::system\_clock::now() - acc->LastInterest).count();

int payoutTime = std::chrono::duration\_cast<std::chrono::hours>(std::chrono::system\_clock::now() - acc->LastPayout).count();

//hours comparison value; payoutRate will change that

int payoutComparison = 8760;

if (p == 1) payoutComparison = 4320;

if (p == 2) payoutComparison = 720;

if (p == 3) payoutComparison = 24;

double rate = acc->APY;

double adjustedRate = rate;

double ratio = 1;

//what type of interest?

switch (t)

{

case 1 : case 2:

//compound interest, yearly

ratio = payoutComparison / 8760.00;

if (interestTime > 8760)

{

acc->interestSoFar = acc->interestSoFar + acc->balance.GetPercentage(adjustedRate);

acc->LastInterest = std::chrono::system\_clock::now();

}

break;

case 3:

//compound interest, monthly

adjustedRate = rate / 12;

ratio = payoutComparison / 720.00;

if (interestTime > 720)

{

acc->interestSoFar = acc->interestSoFar + acc->balance.GetPercentage(adjustedRate);

acc->LastInterest = std::chrono::system\_clock::now();

}

break;

case 4:

//compound interest, daily

adjustedRate = rate / 365;

ratio = payoutComparison / 24.00;

if (interestTime > 24)

{

acc->interestSoFar = acc->interestSoFar + acc->balance.GetPercentage(adjustedRate);

acc->LastInterest = std::chrono::system\_clock::now();

}

break;

default:

//do nothing by default, or no interest

break;

}

//if payout is greater than comparison value & not a certificate of deposit, payout

if (payoutTime > payoutComparison && acc->getType() != "Certificate of Deposit")

{

//use payout static function

payout(acc, adjustedRate, ratio);

}

}

}

/// <summary>

/// Goes through & handles interest for all accounts

/// </summary>

/// <param name="accs">list of accounts to go through</param>

static void AllAccounts(LinkedList<Account> accs)

{

for (int i = 0; i < accs.getCount(); i++)

{

IndividualAccount(accs.get(i));

}

}

};

/// <summary>

/// Database class

/// </summary>

class Database

{

public:

Database() {

//default employee

std::shared\_ptr<Employee> e(new Employee("TotallyNotAnAdmin", "ao2j4ona5rorn2"));

Customers = LinkedList<Customer>();

Employees = LinkedList<Employee>(e); //put default employee into employees

Accounts = LinkedList<Account>();

EncryptionKeys = LinkedList<std::string>();

}

~Database() {}

LinkedList<Customer> Customers; //customers, main users

LinkedList<Employee> Employees; //administrators, essentially

LinkedList<Account> Accounts; //all accounts, split from customers

LinkedList<std::string> EncryptionKeys; //encryption keys (not yet used)

/// <summary>

/// purchase request

/// </summary>

/// <param name="acc">account identifier string</param>

/// <param name="user">user identifer string</param>

/// <returns>was successful, bool</returns>

bool purchase(std::string acc, std::string user, std::string name = "Purchase", std::string origin = "Unknown")

{

std::shared\_ptr<Customer> cust = Customers.get(user);

Overdraft::OnPurchase(cust, std::shared\_ptr<Database>(this));

}

/// <summary>

/// bank processes done at a regular interval

/// </summary>

void bankProcesses()

{

Interest::AllAccounts(Accounts);

}

};

BankDB.cpp (only relevant code)

/// <summary>

/// Handles overdraft for a specific user

/// </summary>

/// <param name="c">Customer shared pointer</param>

/// <returns>was successful?, bool</returns>

bool Overdraft::OnPurchase(std::shared\_ptr<Customer> cust, std::shared\_ptr<Database> d)

{

bool success = false;

for (int i = 0; i < cust->AccountIDs.getCount(); i++)

{

//pointer to account, from account get -> dereferenced pointer to AccountID string

std::shared\_ptr<Account> a;

if (cust->AccountIDs.get(i)) a = d->Accounts.get(\*cust->AccountIDs.get(i));

//check if we actually got the account

if (a)

{

//we can assume string exists at this point

std::string aID = \*cust->AccountIDs.get(i);

//if so, continue & see if account's balance is less than 0

if (a->balance < 0)

{

//check accounts again

for (int j = 0; j < cust->AccountIDs.getCount(); j++)

{

std::shared\_ptr<Account> b;

if (cust->AccountIDs.get(j)) b = d->Accounts.get(\*cust->AccountIDs.get(j));

//make sure b exists

if (b)

{

//we can assume string exists to get to this point

std::string bID = \*cust->AccountIDs.get(j);

//extra if to skip trying to pull money from itself, Just In Case; preemptive bug fix

if (!(bID == aID))

{

//go until account balance is fixed

while (a->balance < 0 && b->available > 0)

{

//make amount to sub, starting at 5 bucks

double amtToSub = 5.00;

//scale amount to sub based on amount currently missing

if (a->balance < USDollar(20.00))

{

amtToSub = 50.00;

}

else if (a->balance < USDollar(10.00))

{

amtToSub = 20.00;

}

else if (a->balance < USDollar(5.00))

{

amtToSub = 10.00;

}

//send from account b to account A

success = cust->transfer(d, bID, aID, amtToSub)==0;

}

}

}

}

}

}

}

return success;

}