Andrew Taylor

4/11/25

COMS-280-WWW01

# Banking System – Phase 4

## Link

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

## Templates Overview

Templates are a wonderful tool for the purpose of code reuse & reliability. You can reduce multiple separate implementations down to just one reusable template, supply the type when you need it, and then the compiler will handle the rest. Beyond having to type less code, you are also funneling everything through this same code; you need only to bugfix one implementation for multiple use cases.

As mentioned in the previous phase, I have made extensive use of Templates in the implementation of my LinkedList. They have been used for the Node, InternalNode, ExternalNode, & LinkedList class. They each take one parameter, type T. Also, I make heavy use of the standard library’s shared pointer class, which is a template as well.

The LinkedList class is the point of entry into the templated classes, you need only supply the type to LinkedList & the Nodes will be using that type as well. Inside of the classes, the supplied type (T) is used to create & use smart pointers generically. It allows for a single LinkedList implementation to support any data that can be pointed to with a pointer & supplied to a template.

## Exception Handling Overview

Exceptions are an easy, programmer supplied way to take note of errors at runtime. There are many things a compiler cannot anticipate, and just as many things C++ allows that could cause major problems. If a bug or problematic behavior cannot be caught at compile time, it could cause major issues down the line. You need to anticipate ways in which your code may behave in unintended ways. It is up to the programmer to throw exceptions, and use try/catch blocks to catch them. These allow both logging of errors and utilizing the catch block to deploy code to help your program fail gracefully.

For ease of use & consistency, I made an exception base class and derived classes for specific exceptions. The primary exception class is housed in a header, other exceptions can be derived in other classes or added to the main exception header, based on reusability. When an exception prints its error, it will also print a string for where the exception was created. This will be a user supplied std::string, as macros/magic functions that turn function identifiers into strings are often compiler specific (often without ways to find out if they are defined) & I’d rather my code be as general as possible.

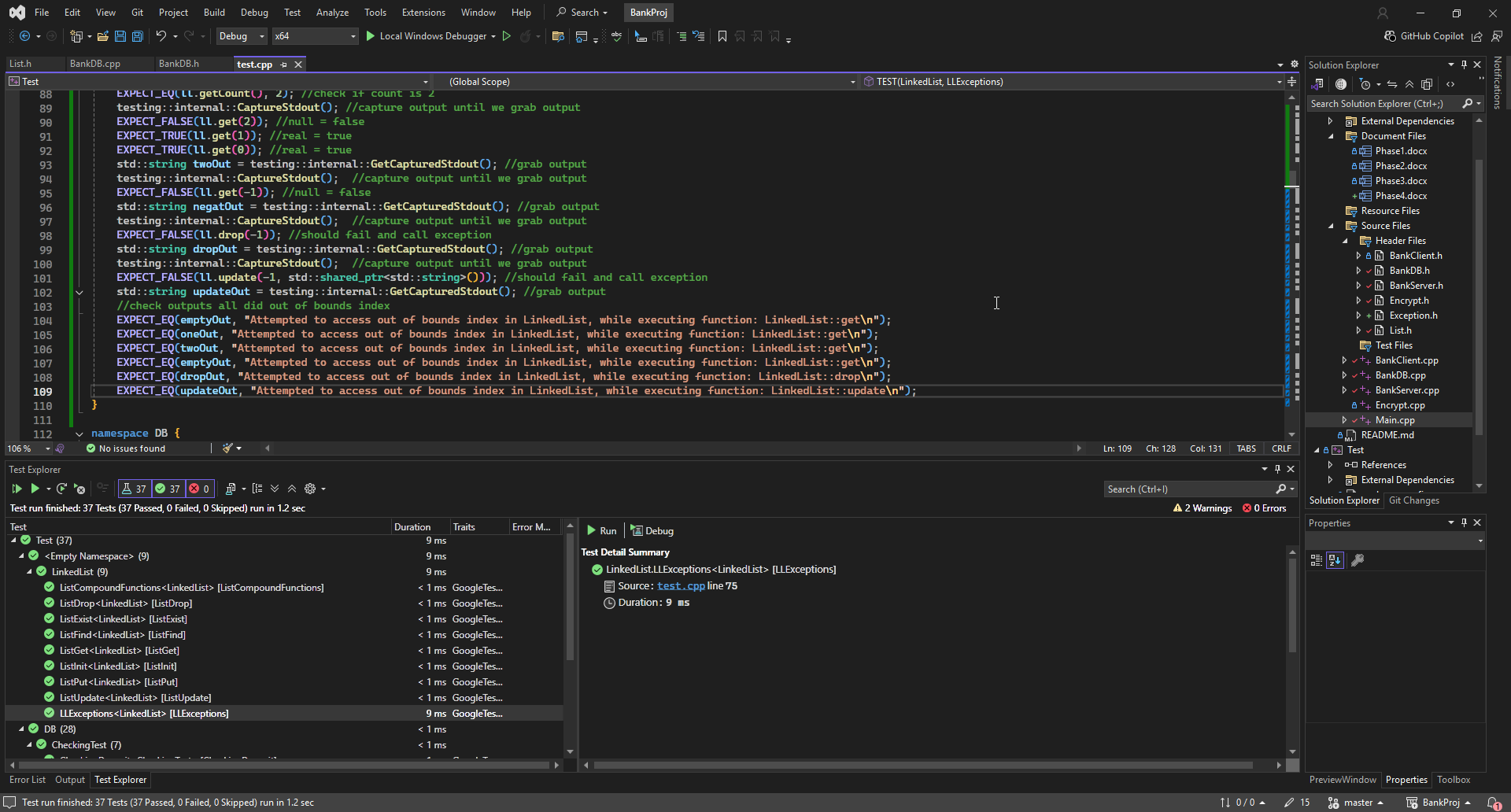
### Exceptions:

Out of Bounds Exception (Linked List, ExLLOoB): Called if the LinkedList is, at any point, asked for an integer outside of its bounds (amount of nodes). This is defined within the LinkedList header, due to it being specific to the LinkedList. LinkedList is used throughout the code.

## Template & Exception Validation

Both my templates & my exceptions are primarily in the LinkedList implementation. As such, I have set up unit tests to primarily just test the functionality of the entire linked list, as the template & exceptions are embedded deeply into that. I also already have my account unit tests to help confirm that my functionality is working, as they make heavy use of the LinkedList<Transaction> embedded in them.

Testing suites are designed around code that does not catch their own exceptions, so testing my linked list functions for that part of the functionality proved a bit challenging. What I have done is captured the error printout during my unit tests.



While this set of tests is for testing LinkedList as a whole, it also proves that the template system is working as intended. The types in the LinkedList are interacted with properly.

## Reflection & Improvements

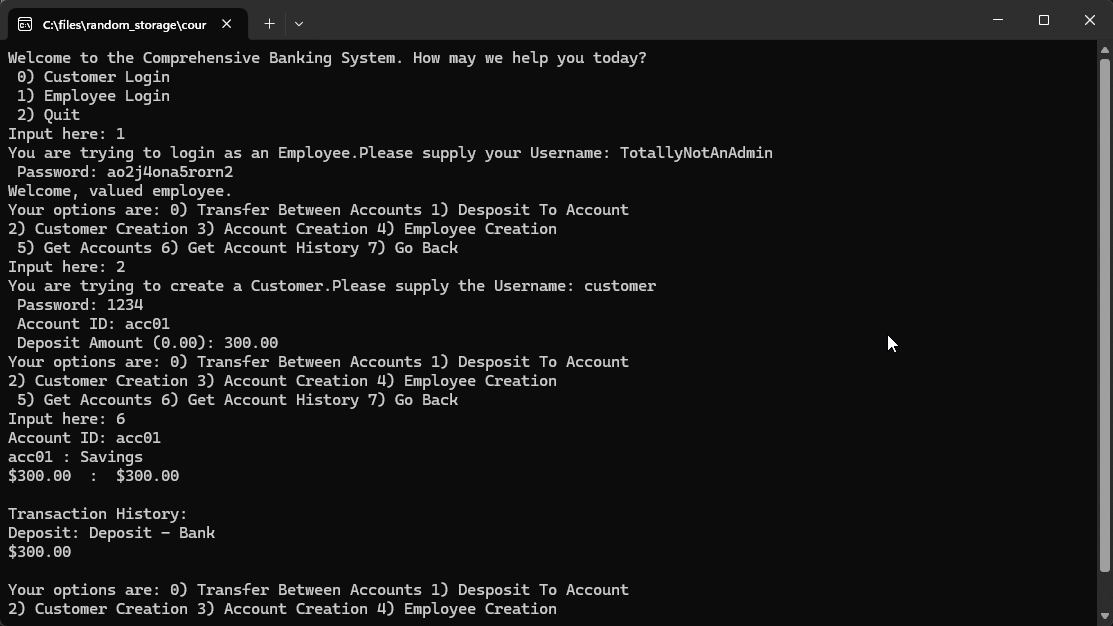
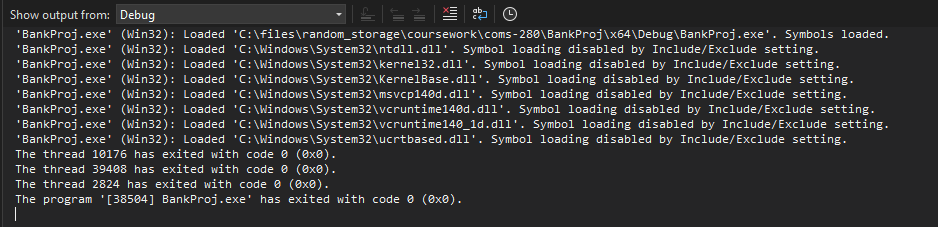
Overall, I had a much better time with templates & exceptions than anticipated at the start of the project. Working together, templates & exceptions make for a strong base of functionality, while handling any issues or errors you might encounter. You filter the code through one implementation with proper exception throwing whenever there’s an issue, and if you understand exactly what is needed inside a template class, you can use it for just about anything.

However, I did something you are not supposed to do and paid the price for about 2 days. A major refactor to change functionality came alongside my implementation of the Out of Bound exception in the LinkedList. When I first designed the list, I had the range as 1,Count. To make the bounds easier to understand, I wanted to change it to 0,Count minus 1. Unsurprisingly, making code without entirely thinking about what you’re doing can make some nasty bugs. I wasn’t paying attention & copy-pasted something that reversed the node order. Luckily, everything worked great after that bugfix, but it had me confused for quite a while. Proper debugging tools are a lifesaver.

The template functionality has one major flaw at the moment. Each class supplied to the LinkedList class needs a “compare” function like std::string to use the “find” function & its compound functions. The “contains” function in std::string provided in preview C++ versions would be a better fit for certain complex classes. However, full usefulness of the template would require a more dynamic solution, with the added benefit of working on a minimum C++ version that is heavily tested.

More robust exception handling is one of the largest future improvements that can be made. This may require major refactoring, and may need to happen after each main element of the system is made into its own discrete program (client, server, database). One of the reasons for this is that prior to exceptions being included, I had already designed most of my code to fail gracefully as part of the functionality. The other reason for this is better logging. Currently, I am logging directly to cout, while also still using that as the UI. This is far from ideal. Most errors need not be user facing, and each module should have their own specific logs.

## Compiled Code Output



## Code

Test.ccp

#include "pch.h"

#include "../Src/header/List.h"

#include "../Src/header/BankDB.h"

//LinkedList initialization

TEST(LinkedList, ListInit)

{

LinkedList < std::string> ll; //make an empty list

EXPECT\_EQ(ll.getCount(),0); //check if count is 0

LinkedList<std::string> ll2(std::shared\_ptr<std::string>(new std::string("string"))); //make a list with 1 entry

EXPECT\_EQ(ll2.getCount(), 1); //check if count is 1

}

//LinkedList put function

TEST(LinkedList, ListPut)

{

LinkedList < std::string> ll; //make an empty list

EXPECT\_EQ(ll.getCount(), 0); //check if count is 0

EXPECT\_TRUE(ll.put(std::shared\_ptr<std::string>(new std::string("string")))); //put a new string in, check if it worked

EXPECT\_EQ(ll.getCount(), 1); //check if count is 1

}

//LinkedList basic get function

TEST(LinkedList, ListGet)

{

LinkedList<std::string> ll(std::shared\_ptr<std::string>(new std::string("string"))); //make a list with 1 entry

EXPECT\_EQ(\*ll.get(0), "string"); //dereference pointer to get string back, check if string is equivalent

}

//LinkedList find function

TEST(LinkedList, ListFind)

{

LinkedList<std::string> ll(std::shared\_ptr<std::string>(new std::string("string"))); //make a list with 1 entry

EXPECT\_EQ(ll.find("string"), 0); //try to find entry for string, should equal 0

}

//LinkedList exist function

TEST(LinkedList, ListExist)

{

LinkedList<std::string> ll(std::shared\_ptr<std::string>(new std::string("string"))); //make a list with 1 entry

EXPECT\_TRUE(ll.exists(0)); //try to find if entry exists

}

//LinkedList drop function

TEST(LinkedList, ListDrop)

{

LinkedList<std::string> ll(std::shared\_ptr<std::string>(new std::string("string"))); //make a list with 1 entry

EXPECT\_TRUE(ll.drop(0)); //drop should work

EXPECT\_EQ(ll.getCount(), 0); //drop should lead to 0 count

}

//LinkedList update function

TEST(LinkedList, ListUpdate)

{

LinkedList<std::string> ll(std::shared\_ptr<std::string>(new std::string("string"))); //make a list with 1 entry

EXPECT\_TRUE(ll.update(0, std::shared\_ptr<std::string>(new std::string("string2")))); //update function, should be true

EXPECT\_EQ(\*ll.get(0), "string2"); //dereference pointer to get string back, see if theupdate worked

}

//functions making use of other functions, like get with string input (find leads to get)

TEST(LinkedList, ListCompoundFunctions)

{

LinkedList<std::string> ll(std::shared\_ptr<std::string>(new std::string("string"))); //make a list with 1 entry

//next functions will seem a bit silly Usefulness-wise, but we do also use LinkedLists for more complex classes

EXPECT\_EQ(\*ll.get("string"), "string"); //check get works

EXPECT\_TRUE(ll.update("string", std::shared\_ptr<std::string>(new std::string("string2")))); //check update works

EXPECT\_TRUE(ll.drop("string2")); //check that drop works

}

//testing for proper exception handling

TEST(LinkedList, LLExceptions)

{

LinkedList < std::string> ll; //make an empty list

EXPECT\_EQ(ll.getCount(), 0); //check if count is 0

testing::internal::CaptureStdout(); //capture output until we grab output

EXPECT\_FALSE(ll.get(0)); //null = false

std::string emptyOut = testing::internal::GetCapturedStdout(); //grab output

EXPECT\_TRUE(ll.put(std::shared\_ptr<std::string>(new std::string("string")))); //put a new string in, check if it worked

EXPECT\_EQ(ll.getCount(), 1); //check if count is 1

testing::internal::CaptureStdout(); //capture output until we grab output

EXPECT\_FALSE(ll.get(1)); //null = false

EXPECT\_TRUE(ll.get(0)); //real = true

std::string oneOut = testing::internal::GetCapturedStdout(); //grab output

EXPECT\_TRUE(ll.put(std::shared\_ptr<std::string>(new std::string("string2")))); //put a new string in, check if it worked

EXPECT\_EQ(ll.getCount(), 2); //check if count is 2

testing::internal::CaptureStdout(); //capture output until we grab output

EXPECT\_FALSE(ll.get(2)); //null = false

EXPECT\_TRUE(ll.get(1)); //real = true

EXPECT\_TRUE(ll.get(0)); //real = true

std::string twoOut = testing::internal::GetCapturedStdout(); //grab output

testing::internal::CaptureStdout(); //capture output until we grab output

EXPECT\_FALSE(ll.get(-1)); //null = false

std::string negatOut = testing::internal::GetCapturedStdout(); //grab output

testing::internal::CaptureStdout(); //capture output until we grab output

EXPECT\_FALSE(ll.drop(-1)); //should fail and call exception

std::string dropOut = testing::internal::GetCapturedStdout(); //grab output

testing::internal::CaptureStdout(); //capture output until we grab output

EXPECT\_FALSE(ll.update(-1, std::shared\_ptr<std::string>())); //should fail and call exception

std::string updateOut = testing::internal::GetCapturedStdout(); //grab output

//check outputs all did out of bounds index

EXPECT\_EQ(emptyOut, "Attempted to access out of bounds index in LinkedList, while executing function: LinkedList::get\n");

EXPECT\_EQ(oneOut, "Attempted to access out of bounds index in LinkedList, while executing function: LinkedList::get\n");

EXPECT\_EQ(twoOut, "Attempted to access out of bounds index in LinkedList, while executing function: LinkedList::get\n");

EXPECT\_EQ(emptyOut, "Attempted to access out of bounds index in LinkedList, while executing function: LinkedList::get\n");

EXPECT\_EQ(dropOut, "Attempted to access out of bounds index in LinkedList, while executing function: LinkedList::drop\n");

EXPECT\_EQ(updateOut, "Attempted to access out of bounds index in LinkedList, while executing function: LinkedList::update\n");

}

namespace DB {

//initialization test

TEST(SavingsTest, SavingsInit) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1)));

Saving s(t, "s0001"); //initialize

EXPECT\_EQ(s.Transactions.get(0)->Val, 1); //test that transaction list is working; 1 cent in initialization, 1 cent in outcome

}

//test just polymorphic call

TEST(SavingsTest, SavingsPolymorph)

{

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1)));

std::shared\_ptr<Account> a(new Saving(t, "s0001")); //initialize;pointer of type "account" to test polymorphism

EXPECT\_EQ(a->getType(), "Savings"); //type information is accessed polymorphically, should return a string "Savings"

}

//transaction

TEST(SavingsTest, SavingsTransaction) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1)));

std::shared\_ptr<Account> a(new Saving(t, "s0001")); //initialize

std::shared\_ptr<Transaction> t1(new BankFunction(USDollar(10))); //new transaction

a->processTransaction(t1); //feed the transaction in

EXPECT\_EQ(a->Transactions.get(1)->Val, 10); //new entry in transaction list, on top of the first one; should give 10 cents

EXPECT\_EQ(a->balance, 11); //balance should successfully add the entries together

EXPECT\_EQ(a->available, 11); //ditto with above, available should match balance

}

//deposit

TEST(SavingsTest, SavingsDeposit) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1)));

std::shared\_ptr<Account> a(new Saving(t, "s0001")); //initialize

EXPECT\_TRUE(a->deposit(10.01)); //deposit 10 bucks & 1 cent, should return true

EXPECT\_EQ(a->Transactions.get(1)->Val, 1001); //check for 2nd transaction as above

EXPECT\_EQ(a->balance, 1002); //balance should be 1 cent + 10.01, 10.02

EXPECT\_EQ(a->available, 1002); //available should be 1 cent + 10.01, 10.02

}

//purchase

TEST(SavingsTest, SavingsPurchase) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(100000))); //1000 dollars

std::shared\_ptr<Account> a(new Saving(t, "s0001")); //initialize

EXPECT\_TRUE(a->purchase(-10.01, "Test Purchase", "Test Company")); //buy something that's 10 bucks & 1 cent

EXPECT\_EQ(a->Transactions.get(1)->Val, -1001); //get the subtractive transaction, -$10.01

EXPECT\_EQ(a->balance, 98999); //$989.99 remaining in balance

EXPECT\_EQ(a->available, 98999); //$989.99 remaining in available

}

//sending transfer

TEST(SavingsTest, SavingsTransferFrom) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(100000))); //1000 dollars

std::shared\_ptr<Account> a(new Saving(t, "s0001")); //initialize

EXPECT\_EQ(a->sendTransfer(10.01), USDollar(1001)); //Transfer 10.01 dollars

EXPECT\_EQ(a->Transactions.get(1)->Val, -1001); //get the subtractive transaction, -$10.01

EXPECT\_EQ(a->balance, 98999); //$989.99 remaining in balance

EXPECT\_EQ(a->available, 98999); //$989.99 remaining in available

}

//sending & recieving transfer

TEST(SavingsTest, SavingsTransferTo) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1))); //1 cent

std::shared\_ptr<Transaction> t1(new Deposit(USDollar(100000))); //1000 dollars

std::shared\_ptr<Account> a(new Saving(t, "s0001")); //init 1

std::shared\_ptr<Account> a1(new Saving(t1, "s0002")); //init 2

EXPECT\_TRUE(a->receiveTransfer(a1->sendTransfer(10.01),a1->ID)); //send & recieve transfer

EXPECT\_EQ(a->Transactions.get(1)->Val, 1001); //get the additive transaction, $10.01

EXPECT\_EQ(a1->Transactions.get(1)->Val, -1001); //get the subtractive transaction, -$10.01

EXPECT\_EQ(a->balance, 1002); //balance should be 1 cent + $10.01, $10.02

EXPECT\_EQ(a->available, 1002); //available should be 1 cent + $10.01, $10.02

EXPECT\_EQ(a1->balance, 98999); //$989.99 remaining in balance

EXPECT\_EQ(a1->available, 98999); //$989.99 remaining in available

}

//initialization test

TEST(CheckingTest, CheckingInit) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1)));

Checking c(t, "0001"); //initialize

EXPECT\_EQ(c.Transactions.get(0)->Val, 1); //test that transaction list is working; 1 cent in initialization, 1 cent in outcome

}

//test just polymorphic call

TEST(CheckingTest, CheckingPolymorph)

{

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1)));

std::shared\_ptr<Account> a(new Checking(t, "c0001")); //initialize;pointer of type "account" to test polymorphism

EXPECT\_EQ(a->getType(), "Checking"); //type information is accessed polymorphically, should return a string "Savings"

}

//transaction

TEST(CheckingTest, CheckingTransaction) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1)));

std::shared\_ptr<Account> a(new Checking(t, "c0001")); //initialize

std::shared\_ptr<Transaction> t1(new BankFunction(USDollar(10))); //new transaction

a->processTransaction(t1); //feed that transaction in

EXPECT\_EQ(a->Transactions.get(1)->Val, 10); //new entry in transaction list, on top of the first one; should give 10 cents

EXPECT\_EQ(a->balance, 11); //balance should successfully add the entries together

EXPECT\_EQ(a->available, 11); //ditto with above, available should match balance

}

//deposit

TEST(CheckingTest, CheckingDeposit) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1)));

std::shared\_ptr<Account> a(new Checking(t, "c0001")); //initialize

EXPECT\_TRUE(a->deposit(10.01)); //deposit 10 bucks & 1 cent, should return true

EXPECT\_EQ(a->Transactions.get(1)->Val, 1001); //check for 2nd transaction as above

EXPECT\_EQ(a->balance, 1002); //balance should be 1 cent + 10.01, 10.02

EXPECT\_EQ(a->available, 1002); //available should be 1 cent + 10.01, 10.02

}

//purchase

TEST(CheckingTest, CheckingPurchase) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(100000))); //1000 dollars

std::shared\_ptr<Account> a(new Checking(t, "c0001")); //initialize

EXPECT\_TRUE(a->purchase(-10.01, "Test Purchase", "Test Company")); //buy something that's 10 bucks & 1 cent

EXPECT\_EQ(a->Transactions.get(1)->Val, -1001); //get the subtractive transaction, -$10.01

EXPECT\_EQ(a->balance, 98999); //$989.99 remaining in balance

EXPECT\_EQ(a->available, 98999); //$989.99 remaining in available

}

//sending transfer

TEST(CheckingTest, CheckingTransferFrom) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(100000))); //1000 dollars

std::shared\_ptr<Account> a(new Checking(t, "c0001")); //initialize

EXPECT\_EQ(a->sendTransfer(10.01), USDollar(1001)); //Transfer 10.01 dollars

EXPECT\_EQ(a->Transactions.get(1)->Val, -1001); //get the subtractive transaction, -$10.01

EXPECT\_EQ(a->balance, 98999); //$989.99 remaining in balance

EXPECT\_EQ(a->available, 98999); //$989.99 remaining in available

}

//sending & recieving transfer

TEST(CheckingTest, CheckingTransferTo) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1))); //1 cent

std::shared\_ptr<Transaction> t1(new Deposit(USDollar(100000))); //1000 dollars

std::shared\_ptr<Account> a(new Checking(t, "c0001")); //init 1

std::shared\_ptr<Account> a1(new Checking(t1, "c0002")); //init 2

EXPECT\_TRUE(a->receiveTransfer(a1->sendTransfer(10.01), a1->ID)); //send & recieve transfer

EXPECT\_EQ(a->Transactions.get(1)->Val, 1001); //get the additive transaction, $10.01

EXPECT\_EQ(a1->Transactions.get(1)->Val, -1001); //get the subtractive transaction, -$10.01

EXPECT\_EQ(a->balance, 1002); //balance should be 1 cent + $10.01, $10.02

EXPECT\_EQ(a->available, 1002); //available should be 1 cent + $10.01, $10.02

EXPECT\_EQ(a1->balance, 98999); //$989.99 remaining in balance

EXPECT\_EQ(a1->available, 98999); //$989.99 remaining in available

}

//initialization test

TEST(CoDTest, CoDInit) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1)));

CertOfDep cd(t, "cd0001"); //initialize

EXPECT\_EQ(cd.Transactions.get(0)->Val, 1); //test that transaction list is working; 1 cent in initialization, 1 cent in outcome

}

//test just polymorphic call

TEST(CoDTest, CoDPolymorph)

{

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1)));

std::shared\_ptr<Account> a(new CertOfDep(t, "cd0001")); //initialize;pointer of type "account" to test polymorphism

EXPECT\_EQ(a->getType(), "Certificate of Deposit"); //type information is accessed polymorphically, should return a string "Savings"

}

//transaction

TEST(CoDTest, CoDTransaction) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1)));

std::shared\_ptr<Account> a(new CertOfDep(t, "cd0001")); //initialize

std::shared\_ptr<Transaction> t1(new BankFunction(USDollar(10))); //new transaction

a->processTransaction(t1); //feed thet transaction in

EXPECT\_EQ(a->Transactions.get(1)->Val, 10); //new entry in transaction list, on top of the first one; should give 10 cents

EXPECT\_EQ(a->balance, 11); //balance should successfully add the entries together

EXPECT\_EQ(a->available, 11); //ditto with above, available should match balance

}

//deposit

TEST(CoDTest, CoDDeposit) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1)));

std::shared\_ptr<Account> a(new CertOfDep(t, "cd0001")); //initialize

EXPECT\_TRUE(a->deposit(10.01)); //deposit 10 bucks & 1 cent, should return true

EXPECT\_EQ(a->Transactions.get(1)->Val, 1001); //check for 2nd transaction as above

EXPECT\_EQ(a->balance, 1002); //balance should be 1 cent + 10.01, 10.02

EXPECT\_EQ(a->available, 1002); //available should be 1 cent + 10.01, 10.02

}

//purchase

TEST(CoDTest, CoDPurchase) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(100000))); //1000 dollars

std::shared\_ptr<Account> a(new CertOfDep(t, "cd0001")); //initialize

EXPECT\_TRUE(a->purchase(-10.01, "Test Purchase", "Test Company")); //buy something that's 10 bucks & 1 cent

EXPECT\_EQ(a->Transactions.get(1)->Val, -1001); //get the subtractive transaction, -$10.01

EXPECT\_EQ(a->balance, 98999); //$989.99 remaining in balance

EXPECT\_EQ(a->available, 98999); //$989.99 remaining in available

}

//sending transfer

TEST(CoDTest, CoDTransferFrom) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(100000))); //1000 dollars

std::shared\_ptr<Account> a(new CertOfDep(t, "cd0001")); //initialize

EXPECT\_EQ(a->sendTransfer(10.01), USDollar(1001)); //Transfer 10.01 dollars

EXPECT\_EQ(a->Transactions.get(1)->Val, -1001); //get the subtractive transaction, -$10.01

EXPECT\_EQ(a->balance, 98999); //$989.99 remaining in balance

EXPECT\_EQ(a->available, 98999); //$989.99 remaining in available

}

//sending & recieving transfer

TEST(CoDTest, CoDTransferTo) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1))); //1 cent

std::shared\_ptr<Transaction> t1(new Deposit(USDollar(100000))); //1000 dollars

std::shared\_ptr<Account> a(new CertOfDep(t, "cd0001")); //init 1

std::shared\_ptr<Account> a1(new CertOfDep(t1, "cd0002")); //init 2

EXPECT\_TRUE(a->receiveTransfer(a1->sendTransfer(10.01), a1->ID)); //send & recieve transfer

EXPECT\_EQ(a->Transactions.get(1)->Val, 1001); //get the additive transaction, $10.01

EXPECT\_EQ(a1->Transactions.get(1)->Val, -1001); //get the subtractive transaction, -$10.01

EXPECT\_EQ(a->balance, 1002); //balance should be 1 cent + $10.01, $10.02

EXPECT\_EQ(a->available, 1002); //available should be 1 cent + $10.01, $10.02

EXPECT\_EQ(a1->balance, 98999); //$989.99 remaining in balance

EXPECT\_EQ(a1->available, 98999); //$989.99 remaining in available

}

//initialization test

TEST(MMTest, MMInit) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1)));

MoneyMarket mm(t, "mm0001"); //initialize

EXPECT\_EQ(mm.Transactions.get(0).get()->Val, 1); //test that transaction list is working; 1 cent in initialization, 1 cent in outcome

}

//test just polymorphic call

TEST(MMTest, MMPolymorph)

{

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1)));

std::shared\_ptr<Account> a(new MoneyMarket(t, "mm0001")); //initialize;pointer of type "account" to test polymorphism

EXPECT\_EQ(a->getType(), "Money Market"); //type information is accessed polymorphically, should return a string "Savings"

}

//transaction

TEST(MMTest, MMTransaction) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1)));

std::shared\_ptr<Account> a(new MoneyMarket(t, "mm0001")); //initialize

std::shared\_ptr<Transaction> t1(new BankFunction(USDollar(10))); //new transaction

a->processTransaction(t1); //feed the transaction in

EXPECT\_EQ(a->Transactions.get(1)->Val, 10); //new entry in transaction list, on top of the first one; should give 10 cents

EXPECT\_EQ(a->balance, 11); //balance should successfully add the entries together

EXPECT\_EQ(a->available, 11); //ditto with above, available should match balance

}

//deposit

TEST(MMTest, MMDeposit) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1)));

std::shared\_ptr<Account> a(new MoneyMarket(t, "mm0001")); //initialize

EXPECT\_TRUE(a->deposit(10.01)); //deposit 10 bucks & 1 cent, should return true

EXPECT\_EQ(a->Transactions.get(1)->Val, 1001); //check for 2nd transaction as above

EXPECT\_EQ(a->balance, 1002); //balance should be 1 cent + 10.01, 10.02

EXPECT\_EQ(a->available, 1002); //available should be 1 cent + 10.01, 10.02

}

//purchase

TEST(MMTest, MMPurchase) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(100000))); //1000 dollars

std::shared\_ptr<Account> a(new MoneyMarket(t, "mm0001")); //initialize

EXPECT\_TRUE(a->purchase(-10.01, "Test Purchase", "Test Company")); //buy something that's 10 bucks & 1 cent

EXPECT\_EQ(a->Transactions.get(1)->Val, -1001); //get the subtractive transaction, -$10.01

EXPECT\_EQ(a->balance, 98999); //$989.99 remaining in balance

EXPECT\_EQ(a->available, 98999); //$989.99 remaining in available

}

//sending transfer

TEST(MMTest, MMTransferFrom) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(100000))); //1000 dollars

std::shared\_ptr<Account> a(new MoneyMarket(t, "mm0001")); //initialize

EXPECT\_EQ(a->sendTransfer(10.01), USDollar(1001)); //Transfer 10.01 dollars

EXPECT\_EQ(a->Transactions.get(1)->Val, -1001); //get the subtractive transaction, -$10.01

EXPECT\_EQ(a->balance, 98999); //$989.99 remaining in balance

EXPECT\_EQ(a->available, 98999); //$989.99 remaining in available

}

//sending & recieving transfer

TEST(MMTest, MMTransferTo) {

std::shared\_ptr<Transaction> t(new Deposit(USDollar(1))); //1 cent

std::shared\_ptr<Transaction> t1(new Deposit(USDollar(100000))); //1000 dollars

std::shared\_ptr<Account> a(new MoneyMarket(t, "mm0001")); //init 1

std::shared\_ptr<Account> a1(new MoneyMarket(t1, "mm0002")); //init 2

EXPECT\_TRUE(a->receiveTransfer(a1->sendTransfer(10.01), a1->ID)); //send & recieve transfer

EXPECT\_EQ(a->Transactions.get(1)->Val, 1001); //get the additive transaction, $10.01

EXPECT\_EQ(a1->Transactions.get(1)->Val, -1001); //get the subtractive transaction, -$10.01

EXPECT\_EQ(a->balance, 1002); //balance should be 1 cent + $10.01, $10.02

EXPECT\_EQ(a->available, 1002); //available should be 1 cent + $10.01, $10.02

EXPECT\_EQ(a1->balance, 98999); //$989.99 remaining in balance

EXPECT\_EQ(a1->available, 98999); //$989.99 remaining in available

}

}

List.h

#pragma once

#include "Exception.h"

#include <string>

#include <memory>

class ExLLOoB : public Exception

{

public:

/// <summary>

/// Constructor

/// </summary>

/// <param name="s">throwing function</param>

ExLLOoB(std::string s) : Exception(s) {}

/// <summary>

/// print the error to cout

/// </summary>

void printError()

{

std::cout << "Attempted to access out of bounds index in LinkedList, while executing function: " << throwingFunc << "\n";

}

};

/// <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 = true;

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>

void drop()

{

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

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

}

};

/// <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 && Node<T>::getNext())

{

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

}

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

{

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

}

if (\*i == \*j)

{

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

{

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

}

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

{

return Node<T>::getNext()->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)

{

//out of bounds detection

if (\*j < 0)

{

throw ExLLOoB("ExternalNode::find");

return -1;

}

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

{

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)

{

//out of bounds detection

if (\*i < 0)

{

throw ExLLOoB("ExternalNode::get");

}

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

{

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

}

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

{

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

}

if (\*i == \*j)

{

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

{

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

}

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

{

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

//out of bounds detection

if (\*i < 0)

{

throw ExLLOoB("ExternalNode::put");

return b;

}

if (\*i > \*j)

{

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)

{

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);

}

}

if (\*i == \*j)

{

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);

}

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

{

b = true;

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

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

Node<T>::setNext(n);

}

}

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;

//out of bounds detection

if (\*i < 0)

{

throw ExLLOoB("ExternalNode::drop");

return b;

}

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

{

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

}

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

{

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

}

if (\*i == \*j)

{

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

{

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

}

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

{

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

//out of bounds detection

if (\*i < 0)

{

throw ExLLOoB("ExternalNode::update");

return b;

}

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

{

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

}

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

{

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

}

if (\*i == \*j)

{

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

{

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

}

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

{

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

//we don't need to check for out of bounds because you Should be able to send any number to exists

//make pointers to values

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) && i != \*j)

{

\*j = count;

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

}

else

{

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

}

//return no matter what; false if nothing happened

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)

{

//try/catch for exceptions

try

{

//out of bounds detection

if (i < 0 || i >= count)

{

throw ExLLOoB("LinkedList::get");

}

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) && i != \*j)

{

\*j = count;

return tail->get(iPointer, j);

}

else

{

return head->get(iPointer, j);

}

}

catch (Exception& ex)

{

//print error, access by reference means it'll get the right printError

ex.printError();

}

//return null if we didn't get anything

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;

try

{

//out of bounds detection; no upper bound for use at the end

if (i < 0)

{

throw ExLLOoB("LinkedList::put");

return b;

}

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) && i != \*j)

{

\*j = count;

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

}

else

{

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

}

if (b) count++;

}

catch (Exception& ex)

{

//print error, access by reference means it'll get the right printError

ex.printError();

}

//return no matter what; false if nothing happened

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)

{

int i = count; //get count

return put(d, i); //feed it to numbered put function

}

/// <summary>

/// Drop (delete) an entry

/// </summary>

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

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

bool drop(int i)

{

bool b = false;

//try/catch for exceptions

try

{

//out of bounds detection

if (i < 0 || i > count)

{

throw ExLLOoB("LinkedList::drop");

return b;

}

//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) && i != \*j)

{

\*j = count;

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

}

else

{

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

}

//update the count

updateCount();

}

catch (Exception& ex)

{

//print error, access by reference means it'll get the right printError

ex.printError();

}

//return no matter what; false if nothing happened

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;

//try/catch for exceptions

try

{

//out of bounds detection

if (i < 0 || i > count)

{

throw ExLLOoB("LinkedList::update");

return b;

}

//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) && i != \*j)

{

\*j = count;

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

}

else

{

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

}

}

catch (Exception& ex)

{

//print error, access by reference means it'll get the right printError

ex.printError();

}

//return no matter what; false if nothing happened

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;

}

};

BankClient.h

#pragma once

#include <string>

#include <memory>

namespace Client

{

//base Menu class, will display and handle logic for each action

class Menu

{

public:

Menu(){}

virtual ~Menu(){}

virtual void logic() = 0;

//function declarations (generic input functions)

bool YesNo(bool def = false);

int DynamicOptions(std::string s);

std::string TextInput(std::string s = "Input here: ");

};

//specific menu classes

class MnuStart : public Menu

{

public:

MnuStart() { logic(); }

~MnuStart() {

}

void logic();

};

class MnuCustomerLogin : public Menu

{

public:

MnuCustomerLogin() { logic(); }

~MnuCustomerLogin() {

}

void logic();

};

class MnuEmployeeLogin : public Menu

{

public:

MnuEmployeeLogin() { logic(); }

~MnuEmployeeLogin() {

}

void logic();

std::string login;

};

class MnuCustomerStart : public Menu

{

public:

MnuCustomerStart(std::string us, std::string pss)

{

user = us;

pass = pss;

logic();

}

~MnuCustomerStart() {

}

void logic();

std::string user;

std::string pass;

};

class MnuEmployeeStart : public Menu

{

public:

MnuEmployeeStart(std::string us, std::string pss)

{

user = us;

pass = pss;

logic();

}

~MnuEmployeeStart() {

}

void logic();

std::string user;

std::string pass;

};

class MnuEmployeeCreation : public Menu

{

public:

MnuEmployeeCreation(std::string us, std::string pss)

{

user = us;

pass = pss;

logic();

}

~MnuEmployeeCreation() {

}

void logic();

std::string user;

std::string pass;

};

class MnuCustomerCreation : public Menu

{

public:

MnuCustomerCreation(std::string us, std::string pss)

{

user = us;

pass = pss;

logic();

}

~MnuCustomerCreation() {

}

void logic();

std::string user;

std::string pass;

};

class MnuAccountCreation : public Menu

{

public:

MnuAccountCreation(std::string us, std::string pss)

{

user = us;

pass = pss;

logic();

}

~MnuAccountCreation() {

}

void logic();

std::string user;

std::string pass;

};

class MnuTransferBetweenAccounts : public Menu

{

public:

MnuTransferBetweenAccounts(std::string us, std::string pss)

{

user = us;

pass = pss;

logic();

}

~MnuTransferBetweenAccounts()

{

}

void logic();

//strings for logic

std::string user;

std::string pass;

};

class MnuDeposit : public Menu

{

public:

MnuDeposit(std::string us, std::string pss)

{

user = us;

pass = pss;

logic();

}

~MnuDeposit() {

}

void logic();

//strings for logic

std::string user;

std::string pass;

};

class MnuGetAccounts : public Menu

{

public:

MnuGetAccounts(std::string us, std::string pss)

{

user = us;

pass = pss;

logic();

}

~MnuGetAccounts() {

}

void logic();

//strings for logic

std::string user;

std::string pass;

};

class MnuGetAccountHistory : public Menu

{

public:

MnuGetAccountHistory(std::string us, std::string pss, std::string acc)

{

user = us;

pass = pss;

account = acc;

logic();

}

~MnuGetAccountHistory() {

}

void logic();

//strings for logic

std::string user;

std::string pass;

std::string account;

};

}

BankClient.cpp

#include "BankServer.h"

#include "BankClient.h"

#include <iostream>

using namespace Client;

//pointer to new server

std::shared\_ptr<Serv::Server> server = std::shared\_ptr<Serv::Server>(new Serv::Server);

//function implementations

/// <summary>

/// boolean inputs

/// </summary>

/// <param name="def">boolean for default option, defaults to no(default=false)</param>

/// <returns>Yes or no (true, false)</returns>

bool Menu::YesNo(bool def)

{

bool b = def; //grab the default

std::string s = ""; //make empty string

std::cout << "(Y/y for yes, N/n for no; default: " << (def ? "yes" : "no") << ")"; //display the request

std::getline(std::cin, s); //get the input

if (s.compare("Y") == 0 || s.compare("y") == 0) b = true; //if yes, true

if (s.compare("N") == 0 || s.compare("n") == 0) b = false; //if no, false

//if anything else, just have the default

return b; //return value

}

/// <summary>

/// dynamic inputs; takes input & then iterates through a string to find return code. -1 for no valid input

/// </summary>

/// <param name="s">string of characters</param>

/// <returns>integer of picked option, -1 if none</returns>

int Menu::DynamicOptions(std::string s)

{

int r = -1; //default value

//number inputs give first 10, probably all we need

if (s.compare("0") == 0) r = 0;

if (s.compare("1") == 0) r = 1;

if (s.compare("2") == 0) r = 2;

if (s.compare("3") == 0) r = 3;

if (s.compare("4") == 0) r = 4;

if (s.compare("5") == 0) r = 5;

if (s.compare("6") == 0) r = 6;

if (s.compare("7") == 0) r = 7;

if (s.compare("8") == 0) r = 8;

if (s.compare("9") == 0) r = 9;

return r;

}

/// <summary>

/// string input, returns empty string if nothing is recieved

/// </summary>

/// <param name="s">The input ask text</param>

/// <returns>string from user</returns>

std::string Menu::TextInput(std::string s)

{

std::string sin = ""; //empty string

std::cout << s; //ask for input

std::getline(std::cin, sin); //just grab the line

return sin; //return whatever we've got

}

/// <summary>

/// landing page

/// </summary>

void MnuStart::logic()

{

std::cout << "Welcome to the Comprehensive Banking System. How may we help you today?\n 0) Customer Login\n 1) Employee Login\n 2) Quit\n"; //give quick menu landing

bool running = true; //we're starting to grab input

int i = -1; //default is negative 1; declared here so it isn't redeclared every loop

while (running) //go until asked to stop

{

i = DynamicOptions(TextInput()); //ask for input, convert to int with option function

if (i == 0)

{

MnuCustomerLogin cl; //start the customer login

//fake a full reset, we just need to continue the loop

std::cout << "Reset in progress.\nWelcome to the Comprehensive Banking System. How may we help you today?\n 0) Customer Login\n 1) Employee Login\n 2) Quit\n";

}

else if (i == 1)

{

MnuEmployeeLogin el; //start the employee login

//fake a full reset, we just need to continue the loop

std::cout << "Reset in progress.\nWelcome to the Comprehensive Banking System. How may we help you today?\n 0) Customer Login\n 1) Employee Login\n 2) Quit\n";

}

else if (i == 2)

{

running = false; //close the loop, shutting the program down

}

else

{

std::cout << "I'm sorry, that's not one of the options, try again.\nYour options are: \n 0) Customer Login\n 1) Employee Login\n"; //redisplay options

}

}

}

/// <summary>

/// login for customers

/// </summary>

void MnuCustomerLogin::logic()

{

std::cout << "You are trying to login as a Customer.";

bool login = true; //we are running the login functionality

while (login) //run until we get success

{

std::cout << "Please supply your";

std::string user = TextInput(" Username: "); //get the username

std::string password = TextInput(" Password: "); //get the password

if (server->userValidation(user, password) == 0)

{

MnuCustomerStart cs(user, password);

login = false;

}

else

{

std::cout << "Username or password incorrect. Please try again.";

}

}

}

/// <summary>

/// login for employees

/// </summary>

void MnuEmployeeLogin::logic()

{

std::cout << "You are trying to login as an Employee.";

bool login = true; //we are running the login functionality

while (login) //run until we get success

{

std::cout << "Please supply your";

std::string user = TextInput(" Username: "); //get the username

std::string password = TextInput(" Password: "); //get the password

if (server->userValidation(user, password) == 1)

{

MnuEmployeeStart es(user, password);

login = false;

}

else

{

std::cout << "Username or password incorrect. Please try again.";

}

}

}

/// <summary>

/// customer overview

/// </summary>

void MnuCustomerStart::logic()

{

if (server->userValidation(user, pass) == 0)

{

std::cout << "Welcome, valued customer\n"; //give quick menu landing

bool running = true; //we're starting to grab input

int i = -1; //default is negative 1; declared here so it isn't redeclared every loop

while (running) //go until asked to stop

{

std::cout << "Your options are 0) Account Summaries 1) Transfer Between Accounts 2) Get Account History 3) Go Back\n";

i = DynamicOptions(TextInput()); //ask for input, convert to int with option function

if (i == 0)

{

MnuGetAccounts ga(user, pass); //get accounts summary

}

else if (i == 1)

{

MnuTransferBetweenAccounts trnsfr(user, pass); //transfer between accounts

}

else if (i == 2)

{

std::string id = TextInput("Account ID: "); //grab account ID

MnuGetAccountHistory ah(user, pass, id); //show account history

}

else if (i == 3)

{

running = false; //end the loop

}

else

{

std::cout << "I'm sorry, that's not one of the options, try again.\n"; //redisplay options

}

}

}

}

/// <summary>

/// employee overview

/// </summary>

void MnuEmployeeStart::logic()

{

if (server->userValidation(user, pass) == 1)

{

std::cout << "Welcome, valued employee.\n"; //give quick menu landing

bool running = true; //we're starting to grab input

int i = -1; //default is negative 1; declared here so it isn't redeclared every loop

while (running) //go until asked to stop

{

std::cout << "Your options are: 0) Transfer Between Accounts 1) Desposit To Account\n2) Customer Creation 3) Account Creation 4) Employee Creation\n 5) Get Accounts 6) Get Account History 7) Go Back\n";

i = DynamicOptions(TextInput()); //ask for input, convert to int with option function

if (i == 0)

{

MnuTransferBetweenAccounts trnsfr(user, pass); //start the transfer menu

}

else if (i == 1)

{

MnuDeposit dep(user, pass); //start the deposit menu

}

else if (i == 2)

{

MnuCustomerCreation cc(user, pass); //start customer creation

}

else if (i == 3)

{

MnuAccountCreation ac(user, pass); //start making a new account

}

else if (i == 4)

{

MnuEmployeeCreation ec(user, pass); //start making anew employee entry

}

else if (i == 5)

{

MnuGetAccounts ga(user, pass);

}

else if (i == 6)

{

std::string id = TextInput("Account ID: "); //grab account ID

MnuGetAccountHistory ah(user, pass, id); //show account history

}

else if (i == 7)

{

running = false; //end the loop

}

else

{

std::cout << "I'm sorry, that's not one of the options, try again.\n"; //redisplay options

}

}

}

}

/// <summary>

/// page to make a new employee login

/// </summary>

void MnuEmployeeCreation::logic()

{

if (server->userValidation(user, pass) == 1)

{

std::cout << "You are trying to create an Employee.";

bool create = true; //we are running the login functionality

while (create) //run until we get success

{

std::cout << "Please supply the";

std::string userC = TextInput(" Username: "); //get the username

std::string passwordC = TextInput(" Password: "); //get the password

if (server->employeeCreation(userC,passwordC)) //make employee

{

create = false;

}

else

{

std::cout << "Error Occured, try again\n";

}

}

}

}

/// <summary>

/// page to make a new customer login & their first account

/// </summary>

void MnuCustomerCreation::logic()

{

if (server->userValidation(user, pass) == 1)

{

std::cout << "You are trying to create a Customer.";

bool create = true; //we are running the login functionality

while (create) //run until we get success

{

std::cout << "Please supply the";

std::string userC = TextInput(" Username: "); //get the username

std::string passwordC = TextInput(" Password: "); //get the password

std::string accC = TextInput(" Account ID: "); //get the account ID

double d = stod(TextInput(" Deposit Amount (0.00): ")); //get the deposit amount, convert to double

if (server->userCreation(userC, passwordC, accC, d)) //make user

{

create = false;

}

else

{

std::cout << "Error Occured, try again\n";

}

}

}

}

/// <summary>

/// making a new account

/// </summary>

void MnuAccountCreation::logic()

{

if (server->userValidation(user, pass) == 1)

{

bool create = true; //we are running the login functionality

while (create) //run until we get success

{

std::cout << "Please supply the";

std::string userC = TextInput(" Username: "); //get the username

std::string accC = TextInput(" Account ID: "); //get the account ID

int i = DynamicOptions(TextInput(" Account Type\n(0 - Savings, 1 - Checking, 2 - Certificate of Deposit, 3 - Money Market): "));

double d = stod(TextInput(" Deposit Amount (0.00): ")); //get the deposit amount, convert to double

if (server->accountCreation(userC, accC, i, d)) //make account

{

create = false;

}

else

{

std::cout << "Error Occured, try again\n";

}

}

}

}

/// <summary>

/// move money between accounts

/// </summary>

void MnuTransferBetweenAccounts::logic()

{

int i = server->userValidation(user, pass);

if (i >= 0)

{

bool create = true; //we are running the login functionality

std::string userC = user;

while (create) //run until we get success

{

std::cout << "Please supply the";

if (i == 1)

{

userC = TextInput(" Username: "); //get the username

}

std::string accC = TextInput(" Account ID: "); //get the account ID

std::string acc2C = TextInput(" 2nd Account ID: "); //get the account ID again

double d = stod(TextInput(" Deposit Amount (0.00): ")); //get the deposit amount, convert to double

if (server->accountsTransfer(userC, accC, acc2C, d)) //make transfers

{

create = false;

}

else

{

std::cout << "Error Occured, try again\n";

}

}

}

}

/// <summary>

/// employee despositing cash; no self service/ATM yet

/// </summary>

void MnuDeposit::logic()

{

if (server->userValidation(user, pass) == 1)

{

bool create = true; //we are running the login functionality

while (create) //run until we get success

{

std::cout << "Please supply the";

std::string userC = TextInput(" Username: "); //get the username

std::string accC = TextInput(" Account ID: "); //get the account ID

double d = stod(TextInput(" Deposit Amount (0.00): ")); //get the deposit amount, convert to double

if (server->accountDeposit(userC, accC, d)) //make deposit

{

create = false;

}

else

{

std::cout << "Error Occured, try again\n";

}

}

}

}

/// <summary>

/// Get the summary of every account for a user

/// </summary>

void MnuGetAccounts::logic()

{

int access = server->userValidation(user, pass); //validate & get privlege

if (access >= 0)

{

server->accountsDisplay(user); //display the accounts

std::cout << "Amount of accounts: " << server->accountsCount(user); //also get count of accounts

}

}

/// <summary>

/// view the transaction history

/// </summary>

void MnuGetAccountHistory::logic()

{

if (server->userValidation(user, pass) >= 0)

{

std::cout << server->accountDisplay(user, account);

std::cout << server->accountTransactions(user, account);

}

}

BankServer.cpp

#include "BankDB.h"

#include "BankServer.h"

using namespace Serv;

//pointer to a new Database

std::shared\_ptr<DB::Database> db = std::shared\_ptr<DB::Database>(new DB::Database);

/// <summary>

/// validates user & gives their access level

/// </summary>

/// <param name="user">username</param>

/// <param name="pass">password</param>

/// <returns>int code for validation; -1 is default, 0-1 are access levels (user,employee)</returns>

int Server::userValidation(std::string user, std::string pass)

{

int result = -1;

int i = db->Customers.find(user);

if (i != -1)

{

std::shared\_ptr<DB::Customer> c = db->Customers.get(i); //grab customer

if (c)

{

if(c->password == pass) result = 0; //will be replaced with password hash comparison later

}

}

else

{

int j = db->Employees.find(user);

if (j != -1)

{

std::shared\_ptr<DB::Employee> e = db->Employees.get(j);

if (e)

{

if(e->password == pass) result = 1; //will be replaced with password hash comparison later

}

}

}

return result;

}

/// <summary>

/// creating a user

/// </summary>

/// <param name="user">username</param>

/// <param name="pass">pass</param>

/// <param name="acc">first account name</param>

/// <param name="deposit">initial deposit</param>

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

bool Server::userCreation(std::string user, std::string pass, std::string acc, double deposit)

{

bool b = false;

if (db->Customers.find(user) == -1 && db->Accounts.find(acc) == -1) //make sure user & acc don't already exist

{

std::shared\_ptr<DB::Customer> u = std::shared\_ptr<DB::Customer>(new DB::Customer(user, pass));

std::shared\_ptr<DB::Account> a = std::shared\_ptr<DB::Account>(new DB::Saving(std::shared\_ptr<DB::Transaction>(new DB::Deposit(deposit)), acc));

std::shared\_ptr<std::string> s = std::shared\_ptr<std::string>(new std::string(acc));

b = (u->AccountIDs.put(s) && db->Accounts.put(a) && db->Customers.put(u));

}

return b;

}

/// <summary>

/// Creates an employee account. just needs user & pass

/// </summary>

/// <param name="user">username</param>

/// <param name="pass">password</param>

/// <returns>were we successful? bool </returns>

bool Server::employeeCreation(std::string user, std::string pass)

{

bool b = false;

if (db->Customers.find(user) == -1)

{

std::shared\_ptr<DB::Employee> u = std::shared\_ptr<DB::Employee>(new DB::Employee(user, pass));

b = db->Employees.put(u);

}

return b;

}

/// <summary>

/// creating a new account for an existing user

/// </summary>

/// <param name="user">existing user</param>

/// <param name="acc">account name to make</param>

/// <param name="t">type code</param>

/// <param name="deposit">deposit amount</param>

/// <returns>were we successful, bool</returns>

bool Server::accountCreation(std::string user, std::string acc, int t, double deposit)

{

bool b = false;

int i = db->Customers.find(user);

if (i != -1)

{

std::shared\_ptr<DB::Customer> c = db->Customers.get(i);

if (c)

{

std::shared\_ptr<DB::Transaction> tr(new DB::Deposit(deposit));

if (c->AccountIDs.find(acc) == -1 && db->Accounts.find(acc) == -1)

{

std::shared\_ptr<DB::Account> a; //make empty pointer

switch (t) //switch based on #

{

case 0:

a = std::shared\_ptr<DB::Account>(new DB::Saving(tr, acc));

break;

case 1:

a = std::shared\_ptr<DB::Account>(new DB::Checking(tr, acc));

break;

case 2:

a = std::shared\_ptr<DB::Account>(new DB::CertOfDep(tr, acc));

break;

case 3:

a = std::shared\_ptr<DB::Account>(new DB::MoneyMarket(tr, acc));

break;

default:

break;

}

if (a) //make sure our pointer isn't empty

{

std::shared\_ptr<std::string> id(new std::string(a->ID)); //make ID

b = c->AccountIDs.put(id) && db->Accounts.put(a); //put ID & account

}

}

}

}

return b;

}

/// <summary>

/// transfer between accounts

/// </summary>

/// <param name="user">user to transfer for</param>

/// <param name="acc">account 1</param>

/// <param name="acc2">account 2</param>

/// <param name="amnt">amount to transfer</param>

/// <returns>were we successful, bool</returns>

bool Server::accountsTransfer(std::string user, std::string acc, std::string acc2, double amnt)

{

bool b = false;

int i = db->Customers.find(user);

if (i != -1)

{

std::shared\_ptr<DB::Customer> c = db->Customers.get(i);

if (c)

{

b = c->transfer(db, acc, acc2, amnt);

}

}

return b;

}

/// <summary>

/// Depositing into account

/// </summary>

/// <param name="user">user to deposit for</param>

/// <param name="acc">account to deposit in</param>

/// <param name="deposit">amount to deposit</param>

/// <returns>were we successful? bool</returns>

bool Server::accountDeposit(std::string user, std::string acc, double deposit)

{

bool b = false;

int i = db->Customers.find(user);

if (i != -1)

{

std::shared\_ptr<DB::Customer> c = db->Customers.get(i);

if (c)

{

b = c->deposit(db, acc, deposit);

}

}

return b;

}

/// <summary>

/// get the count of accounts. For customers, their accounts. for Employees, all accounts.

/// </summary>

/// <param name="user">user to get</param>

/// <returns>count of accounts, int</returns>

int Server::accountsCount(std::string user)

{

int i = db->Customers.find(user);

if (i != -1)

{

return db->Customers.get(i)->AccountIDs.getCount();

}

else

{

int j = db->Employees.find(user);

if (j != -1)

{

return db->Accounts.getCount();

}

}

return 0;

}

/// <summary>

/// account display, displays a summary of 1 account

/// </summary>

/// <param name="user">user to check account against</param>

/// <param name="acc">account to find</param>

/// <returns>text for account display, string</returns>

std::string Server::accountDisplay(std::string user, std::string acc)

{

int i = db->Customers.find(user); //get user

if (i != -1)

{

std::shared\_ptr<DB::Customer> c = db->Customers.get(i);

if (c && c->AccountIDs.find(acc) >= 0)

{

int z = db->Accounts.find(acc);

std::shared\_ptr<DB::Account> a = db->Accounts.get(z);

if (a) return a->preview();

}

}

else

{

int j = db->Employees.find(user);

if (j != -1)

{

int k = db->Accounts.find(acc);

if (k > -1) return db->Accounts.get(k)->preview();

}

}

return "";

}

/// <summary>

/// displays all accounts; all of customers, or all of them for employees

/// </summary>

/// <param name="user">user to check against</param>

/// <returns>a combined string of all the accounts</returns>

std::string Server::accountsDisplay(std::string user)

{

std::string s = "";

int i = db->Customers.find(user); //get user

if (i != -1)

{

std::shared\_ptr<DB::Customer> c = db->Customers.get(i);

if (c)

{

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

{

s += accountDisplay(user, \*c->AccountIDs.get(j));

}

}

}

else

{

int j = db->Employees.find(user);

if (j != -1)

{

for (int j = 0; j < db->Accounts.getCount(); j++)

{

s += accountDisplay(user, db->Accounts.get(j)->ID);

}

}

}

return s;

}

/// <summary>

/// list of account transactions

/// </summary>

/// <param name="user">user to check against</param>

/// <param name="acc">account to get</param>

/// <returns>list of all transactions, string</returns>

std::string Server::accountTransactions(std::string user, std::string acc)

{

int i = db->Customers.find(user); //get user

if (i != -1)

{

std::shared\_ptr<DB::Customer> c = db->Customers.get(i);

if (c && c->AccountIDs.find(acc) >= 0)

{

int z = db->Accounts.find(acc);

std::shared\_ptr<DB::Account> a = db->Accounts.get(z);

if (a) return a->transactionHistory();

}

}

else

{

int j = db->Employees.find(user);

if (j != -1)

{

int k = db->Accounts.find(acc);

if (k != -1) return db->Accounts.get(k)->transactionHistory();

}

}

return "";

}

BankServer.h

#pragma once

#include "Encrypt.h"

#include "List.h"

namespace Serv

{

/// <summary>

/// Connection layer between Client & DB; will Encrypt & Decrypt everything once encryption is implemented.

/// </summary>

class Server

{

public:

Server(){}

~Server(){}

//validates the user, customer or employee

int userValidation(std::string user, std::string pass);

//creates the user

bool userCreation(std::string user, std::string pass, std::string acc, double deposit = 10.00);

//creates the employee

bool employeeCreation(std::string user, std::string pass);

//adds account to user

bool accountCreation(std::string user, std::string acc, int t, double deposit=10.00);

//transfers between accounts

bool accountsTransfer(std::string user, std::string acc, std::string acc2, double amnt);

//deposits into account

bool accountDeposit(std::string user, std::string acc, double deposit);

//counts accounts available

int accountsCount(std::string user);

//displays summary of accounts

std::string accountDisplay(std::string user, std::string acc);

//displays summaries

std::string accountsDisplay(std::string user);

//gets transactions

std::string accountTransactions(std::string user, std::string acc);

};

}

BankDB.h

#pragma once

#include "List.h"

#include <chrono>

#include <string>

namespace DB

{

//Forward declarations

class Database;

/// <summary>

/// Currency base class

/// </summary>

class Currency

{

public:

Currency(int i = 0) //default/int constructor; this isn't human facing so we just need to pass the raw value

{

value = i;

}

virtual ~Currency(){} //destructor

virtual const std::string getName() = 0; //currency Name

virtual const std::string getSymbol() = 0;//currency symbol, like $

virtual const std::string formattedValue() = 0; //formatted currency, like $0.00

//allows string comparison function

int compare(std::string s)

{

return formattedValue().compare(s); //compares formatted value string

}

protected:

int value; //value is stored as int to innately handle

};

/// <summary>

/// Currency used by the bank, United States Dollars

/// </summary>

class USDollar : public Currency

{

public:

USDollar(int i = 1) : Currency(i) {} //default/int constructor; this isn't human facing so we just need to pass the raw value

USDollar(double d) : Currency((int)(d \* 100)) {} //double constructor; human-facing systems & interest calcs will make doubles

~USDollar() {} //destructor

//returns the name of the currency. Could be used to ID currency types

const std::string getName()

{

return "US Dollar";

}

//gets the symbol; might not need this but could be useful later

const std::string getSymbol()

{

return "$";

}

//returns the value with the proper decimal position & dollar sign, as a string

const std::string formattedValue()

{

std::string s = std::to\_string(value); //converts value to a string

//is it negative

if (s.front() == '-')

{

s.erase(0, 1); //get rid of negative

if (s.length() < 3) s = "00" + s; //adds 2 0's to keep formatting right

s = "$" + s; //add dollar sign

s.insert(s.length() - 2, "."); //add decimal point

s = "-" + s; //add negative back

}

else

{

if (s.length() < 3) s = "00" + s; //adds 2 0's to keep formatting right

s = "$" + s; //add dollar sign

s.insert(s.length() - 2, "."); //add decimal point

}

return s; //return string

}

//operators & certain other functions do need to be implemented in the derived class

//Gets Percentage; as the stored int is real value \* 100 already, we do have to cast to int

USDollar GetPercentage(double d)

{

return USDollar(int(value \* (d / 100)));

}

//positive operator

USDollar operator+() const

{

return USDollar(value);

}

//negative operator

USDollar operator-() const

{

return USDollar(-value);

}

//allows addition

USDollar operator+(const USDollar& add) const

{

return USDollar(value + add.value);

}

//allows subtraction

USDollar operator-(const USDollar& sub) const

{

return USDollar(value - sub.value);

}

//comparison operators

//equals operator overload

bool operator==(const USDollar& comp) const

{

return value == comp.value;

}

//not equal operator overload

bool operator!=(const USDollar& comp) const

{

return value != comp.value;

}

//greater than operator overload

bool operator>(const USDollar& comp) const

{

return value > comp.value;

}

//less than operator overload

bool operator<(const USDollar& comp) const

{

return value < comp.value;

}

//greater than or equal overload

bool operator>=(const USDollar& comp) const

{

return value >= comp.value;

}

//less than or equal overload

bool operator<=(const USDollar& comp) const

{

return value <= comp.value;

}

};

/// <summary>

/// individual transactions

/// </summary>

class Transaction

{

public:

Transaction(USDollar c) //constructor; don't want a default constructor cause we always want a dollar amount

{

Val = c;

};

virtual ~Transaction(){} //destructor

const std::chrono::system\_clock::time\_point Timestamp = std::chrono::system\_clock::now(); //time, to resolve conflicts + sorting

USDollar Val; //the actual value of the transaction

std::string Name = "Transaction"; //default transaction name is Transaction; should be changed

std::string Origin = "Bank"; //default Origin is Bank; will need to be changed

bool Pending = false; //transaction isn't finalized; initalized as false

bool Suspicious = false; //marks transaction as suspicious

virtual std::string TransactionType() = 0;

int compare(std::string s) //lets Compare work on this class; uses all strings available

{

return (Name + Origin + TransactionType() + Val.formattedValue()).compare(s);

}

};

/// <summary>

/// purchases (like debit)

/// </summary>

class Purchase : public Transaction

{

public:

Purchase(USDollar c, std::string n, std::string o) : Transaction(c) {

Name = n;

Origin = o;

}

~Purchase() {}

std::string TransactionType() { return "Purchase"; }

};

/// <summary>

/// transfer between accounts transaction

/// </summary>

class Transfer : public Transaction

{

public:

Transfer(USDollar c, std::string n) : Transaction(c) {

Name = n;

}

~Transfer() {}

std::string TransactionType() { return "Transfer"; }

};

/// <summary>

/// transaction for deposits

/// </summary>

class Deposit : public Transaction

{

public:

Deposit(USDollar c, std::string o = "Bank") : Transaction(c) {

Name = "Deposit";

Origin = o;

}

~Deposit() {}

std::string TransactionType() { return "Deposit"; }

};

/// <summary>

/// transaction for overdraft/interest/other

/// </summary>

class BankFunction : public Transaction

{

public:

BankFunction(USDollar c, std::string n = "Bank Function") : Transaction(c) {

Name = n;

}

~BankFunction() {}

std::string TransactionType() { return "Bank Function"; }

};

/// <summary>

/// Bank Accounts base class

/// </summary>

class Account

{

friend class Interest; //forward declaration of friendship

public:

Account(std::shared\_ptr<Transaction> t, std::string id) {

Transactions = LinkedList<Transaction>(t); //construct Transaction list

updateBalance(); //get the first balance

ID = id; //gets the name; we always want a unique name, 0000 would be an error/placeholder

}

virtual ~Account(){}

LinkedList<Transaction> Transactions; //transaction history!

std::string ID = "0000"; //identifier

USDollar balance; //total balance; updated when transactions gets changed

USDollar available; //total available; in theory, it is total balance - account minimum & certain charges

double APY = 0; //interest rate (can always be expressed as APY, it's just that simple doesn't compound each year)

int interestType = 0; //0: None, 1: Simple, 2: Compound Yearly, 3: Compound Monthly, 4: Compound Daily

int payoutRate = 0; //0: Yearly/None, 1: Every 6 months, 2: monthly, 3: daily

USDollar interestSoFar; //interest accrued so far. This is needed both for compounding & also compound that doesn't pay out at the compound rate

//time members; will just go unused when interest is disabled

//last time paid out; compared against for current payout. default is now(), whenever it is initialized.

std::chrono::system\_clock::time\_point LastPayout = std::chrono::system\_clock::now();

//last time interest came in; compared against for interest. default is now(), whenever it is initialized.

std::chrono::system\_clock::time\_point LastInterest = std::chrono::system\_clock::now();

void updateBalance() //updates balance & available

{

USDollar b(0); //balance

USDollar a(0); //available

//for each member of the transactions list

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

{

//get the transaction as a pointer

std::shared\_ptr<Transaction> t = Transactions.get(i);

//make sure i actually grabbed a value & not a null ptr

if (t)

{

//check if transaction isn't pending, add to available

if (!(t->Pending)) a = a + t->Val;

b = b + t->Val; //add to balance

//fill the values

balance = b;

available = a;

}

}

}

int compare(std::string s) const //lets Compare work on this class; gets the ID

{

return ID.compare(s);

}

//function for displaying the account at a glance

std::string preview()

{

std::string s = "";

s.append(ID + " : " + this->getType()+ "\n");

s.append(available.formattedValue() + " : " + balance.formattedValue() + "\n\n");

return s;

}

//displays transaction history

std::string transactionHistory()

{

std::string s = "Transaction History:\n";

for (int i = Transactions.getCount()-1; i >= 0; i--)

{

//grabs transaction as pointer (shared ptr lets me make & destroy as many as I want)

std::shared\_ptr<Transaction> t = Transactions.get(i);

if (t) //check if it exists

{

s.append(t->TransactionType() + ": " + t->Name + " - " + t->Origin + "\n"); //type, name, and origin

s.append(t->Val.formattedValue() + "\n\n"); //value display (money gained/lost)

}

}

return s;

}

virtual bool deposit(double d) = 0; //deposit dollar amount

virtual USDollar sendTransfer(double d) = 0; //creates the transfer dollar amount

virtual bool receiveTransfer(USDollar d, std::string id) = 0; //receive transfer amount

virtual bool purchase(double d, std::string name, std::string origin) = 0; //handles purchases

virtual int processTransaction(std::shared\_ptr<Transaction> t) = 0; //receives a new transaction

virtual void specialFunctions() = 0; //will run certain special functions in derived classes

virtual std::string getType() //returns account type

{

return "Account";

}

};

/// <summary>

/// User base class, takes a name & password

/// </summary>

class User

{

public:

User(std::string s, std::string pass)

{

name = s;

password = pass;

}

virtual ~User() {}

std::string name;

std::string password;

int compare(std::string s) //comparison function

{

return name.compare(s);

}

virtual int transfer(std::shared\_ptr<Database> d, std::string acc1, std::string acc2, double v) = 0; //Transfer by accounts for user; int for return code

virtual int deposit(std::shared\_ptr<Database> d, std::string acc, double v) = 0; //Deposit into account; int for return code

};

/// <summary>

/// Savings account

/// </summary>

class Saving : public Account

{

public:

Saving(std::shared\_ptr<Transaction> t, std::string id) : Account(t, id) {}

~Saving() {}

void specialFunctions() //handles special functions ()

{

}

bool deposit(double d) //deposits money

{

//pretty verbose line here, let's work backwards

//I make a new dollar amount with the double, pass that to the new Transaction, which gets put into a smart pointer, which is then processed (success is 1)

return processTransaction(std::shared\_ptr<Transaction>(new Deposit(USDollar(d))))==1;

}

USDollar sendTransfer(double d) //transfers money

{

//simple for right now, logic for special functionality related to transfers will be here

USDollar dollar = USDollar(d); //makes dollar amount

processTransaction(std::shared\_ptr<Transaction>(new Transfer(-dollar, ID))); //create the transfer

return dollar;

}

bool receiveTransfer(USDollar d, std::string id) //recieves transfered money

{

//transfer recieve, success is 1

return processTransaction(std::shared\_ptr<Transaction>(new Transfer(USDollar(d), id)))==1;

}

bool purchase(double d, std::string name, std::string origin) //handles purchase

{

//purchase success is 1

return processTransaction(std::shared\_ptr<Transaction>(new Purchase(USDollar(d), name, origin)))==1;

}

int processTransaction(std::shared\_ptr<Transaction> t) //underlying method for processing transactions (int return code for what happened to the transaction)

{

//very simple for right now

int i = 0; //failure code is 0

if (Transactions.put(t))

{

i = 1; //success code is 1

updateBalance(); //update the balance/available

}

return i; //return code

}

std::string getType() //returns account type

{

return "Savings";

}

};

/// <summary>

/// Checking account

/// </summary>

class Checking : public Account

{

public:

Checking(std::shared\_ptr<Transaction> t, std::string id) : Account(t, id) {}

~Checking() {}

void specialFunctions() //handles special functions ()

{

}

bool deposit(double d) //deposits money

{

//pretty verbose line here, let's work backwards

//I make a new dollar amount with the double, pass that to the new Transaction, which gets put into a smart pointer, which is then processed (success is 1)

return processTransaction(std::shared\_ptr<Transaction>(new Deposit(USDollar(d)))) == 1;

}

USDollar sendTransfer(double d) //transfers money

{

//simple for right now, logic for special functionality related to transfers will be here

USDollar dollar = USDollar(d); //makes dollar amount

processTransaction(std::shared\_ptr<Transaction>(new Transfer(-dollar, ID))); //create the transfer

return dollar;

}

bool receiveTransfer(USDollar d, std::string id) //recieves transfered money

{

//transfer recieve, success is 1

return processTransaction(std::shared\_ptr<Transaction>(new Transfer(USDollar(d), id))) == 1;

}

bool purchase(double d, std::string name, std::string origin) //handles purchase

{

//purchase success is 1

return processTransaction(std::shared\_ptr<Transaction>(new Purchase(USDollar(d), name, origin))) == 1;

}

int processTransaction(std::shared\_ptr<Transaction> t) //underlying method for processing transactions (int return code for what happened to the transaction)

{

//very simple for right now

int i = 0; //failure code is 0

if (Transactions.put(t))

{

i = 1; //success code is 1

updateBalance(); //update the balance/available

}

return i; //return code

}

std::string getType() //returns account type

{

return "Checking";

}

};

/// <summary>

/// Certificates of Deposit account

/// </summary>

class CertOfDep : public Account

{

public:

CertOfDep(std::shared\_ptr<Transaction> t, std::string id) : Account(t, id) {}

~CertOfDep() {}

std::chrono::system\_clock::time\_point EndOfTerm;

void specialFunctions() //handles special functions (Savings + ensuring term)

{

}

bool deposit(double d) //deposits money

{

//pretty verbose line here, let's work backwards

//I make a new dollar amount with the double, pass that to the new Transaction, which gets put into a smart pointer, which is then processed (success is 1)

return processTransaction(std::shared\_ptr<Transaction>(new Deposit(USDollar(d)))) == 1;

}

USDollar sendTransfer(double d) //transfers money

{

//simple for right now, logic for special functionality related to transfers will be here

USDollar dollar = USDollar(d); //makes dollar amount

processTransaction(std::shared\_ptr<Transaction>(new Transfer(-dollar, ID))); //create the transfer

return dollar;

}

bool receiveTransfer(USDollar d, std::string id) //recieves transfered money

{

//transfer recieve, success is 1

return processTransaction(std::shared\_ptr<Transaction>(new Transfer(USDollar(d), id))) == 1;

}

bool purchase(double d, std::string name, std::string origin) //handles purchase

{

//purchase success is 1

return processTransaction(std::shared\_ptr<Transaction>(new Purchase(USDollar(d), name, origin))) == 1;

}

int processTransaction(std::shared\_ptr<Transaction> t) //underlying method for processing transactions (int return code for what happened to the transaction)

{

//very simple for right now

int i = 0; //failure code is 0

if (Transactions.put(t))

{

i = 1; //success code is 1

updateBalance(); //update the balance/available

}

return i; //return code

}

std::string getType() //returns account type

{

return "Certificate of Deposit";

}

};

/// <summary>

/// Money Market account

/// </summary>

class MoneyMarket : public Account

{

public:

MoneyMarket(std::shared\_ptr<Transaction> t, std::string id) : Account(t, id) {}

~MoneyMarket() {}

void specialFunctions() //handles special functions (has needs of both checkings & savings, and its own needs)

{

}

bool deposit(double d) //deposits money

{

//pretty verbose line here, let's work backwards

//I make a new dollar amount with the double, pass that to the new Transaction, which gets put into a smart pointer, which is then processed (success is 1)

return processTransaction(std::shared\_ptr<Transaction>(new Deposit(USDollar(d)))) == 1;

}

USDollar sendTransfer(double d) //transfers money

{

//simple for right now, logic for special functionality related to transfers will be here

USDollar dollar = USDollar(d); //makes dollar amount

processTransaction(std::shared\_ptr<Transaction>(new Transfer(-dollar, ID))); //create the transfer

return dollar;

}

bool receiveTransfer(USDollar d, std::string id) //recieves transfered money

{

//transfer recieve, success is 1

return processTransaction(std::shared\_ptr<Transaction>(new Transfer(USDollar(d), id))) == 1;

}

bool purchase(double d, std::string name, std::string origin) //handles purchase

{

//purchase success is 1

return processTransaction(std::shared\_ptr<Transaction>(new Purchase(USDollar(d), name, origin))) == 1;

}

int processTransaction(std::shared\_ptr<Transaction> t) //underlying method for processing transactions (int return code for what happened to the transaction)

{

//very simple for right now

int i = 0; //failure code is 0

if (Transactions.put(t))

{

i = 1; //success code is 1

updateBalance(); //update the balance/available

}

return i; //return code

}

std::string getType() //returns account type

{

return "Money Market";

}

};

/// <summary>

/// Customer User; has accounts associated via IDs in Linked List (primary as first id)

/// </summary>

class Customer : public User

{

public:

Customer(std::string s, std::string pass) : User(s,pass) {}

~Customer() {}

LinkedList<std::string> AccountIDs; //accounts owned/accessible by user

//Transfer between accounts; int for return code. Customers need to own/have access to account

int transfer(std::shared\_ptr<Database> d, std::string acc1, std::string acc2, double v);

//Customer deposit logic

int deposit(std::shared\_ptr<Database> d, std::string acc, double v);

};

/// <summary>

/// Employee User

/// </summary>

class Employee : public User

{

public:

Employee(std::string s, std::string pass) : User(s,pass) {}

~Employee() {}

//Transfer between accounts; int for return code. Employees don't care about account ownership

int transfer(std::shared\_ptr<Database> d, std::string acc1, std::string acc2, double v);

int deposit(std::shared\_ptr<Database> d, std::string acc, double v);

};

/// <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

#include "BankDB.h"

using namespace DB;

int Customer::transfer(std::shared\_ptr<Database> d, std::string acc1, std::string acc2, double v)

//Transfer between accounts; int for return code. Customers need to own/have access to account

{

if (AccountIDs.find(acc1) >= 0 && AccountIDs.find(acc2) >= 0) //if the customer has access to both accounts (find returns -1 if it can't find anything)

{

std::shared\_ptr<Account> Account1 = d->Accounts.get(acc1); //grab account 1

std::shared\_ptr<Account> Account2 = d->Accounts.get(acc2); //grab account 2

//bool can be converted to int, so i can return it. SendTransfer creates the dollar amount & puts a negative transaction in Account 1,

// passing dollar amount to Account 2 which will also confirm the transaction completed successfully when done

return Account2->receiveTransfer(Account1->sendTransfer(v), acc1);

}

else

{

return 0;

}

}

int Customer::deposit(std::shared\_ptr<Database> d, std::string acc, double v)

{

if (AccountIDs.find(acc) >= 0)

{

return 1;

}

else

{

return 0;

}

}

int Employee::transfer(std::shared\_ptr<Database> d, std::string acc1, std::string acc2, double v)

//Transfer between accounts; int for return code. Employees don't care about account ownership

{

if (d->Accounts.find(acc1) >= 0 && d->Accounts.find(acc2) >= 0)

//check if both accounts exist (find returns -1 if it can't find anything)

{

std::shared\_ptr<Account> Account1 = d->Accounts.get(acc1); //grab account 1

std::shared\_ptr<Account> Account2 = d->Accounts.get(acc2); //grab account 2

//bool can be converted to int, so i can return it. SendTransfer creates the dollar amount & puts a negative transaction in Account 1,

// passing dollar amount to Account 2 which will also confirm the transaction completed successfully when done

return Account2->receiveTransfer(Account1->sendTransfer(v), acc1);

}

else

{

return 0;

}

}

int Employee::deposit(std::shared\_ptr<Database> d, std::string acc, double v)

{

if (d->Accounts.find(acc) >= 0)

{

return 1;

}

else

{

return 0;

}

}

/// <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;

}

Main.cpp

#include "BankDB.h"

#include "BankServer.h"

#include "BankClient.h"

#include <string>

#include <iostream>

int main()

{

Client::MnuStart(); //start

return 0; //end the program

}