Andrew Taylor

4/18/25

COMS-280-WWW01

# Banking System – Final

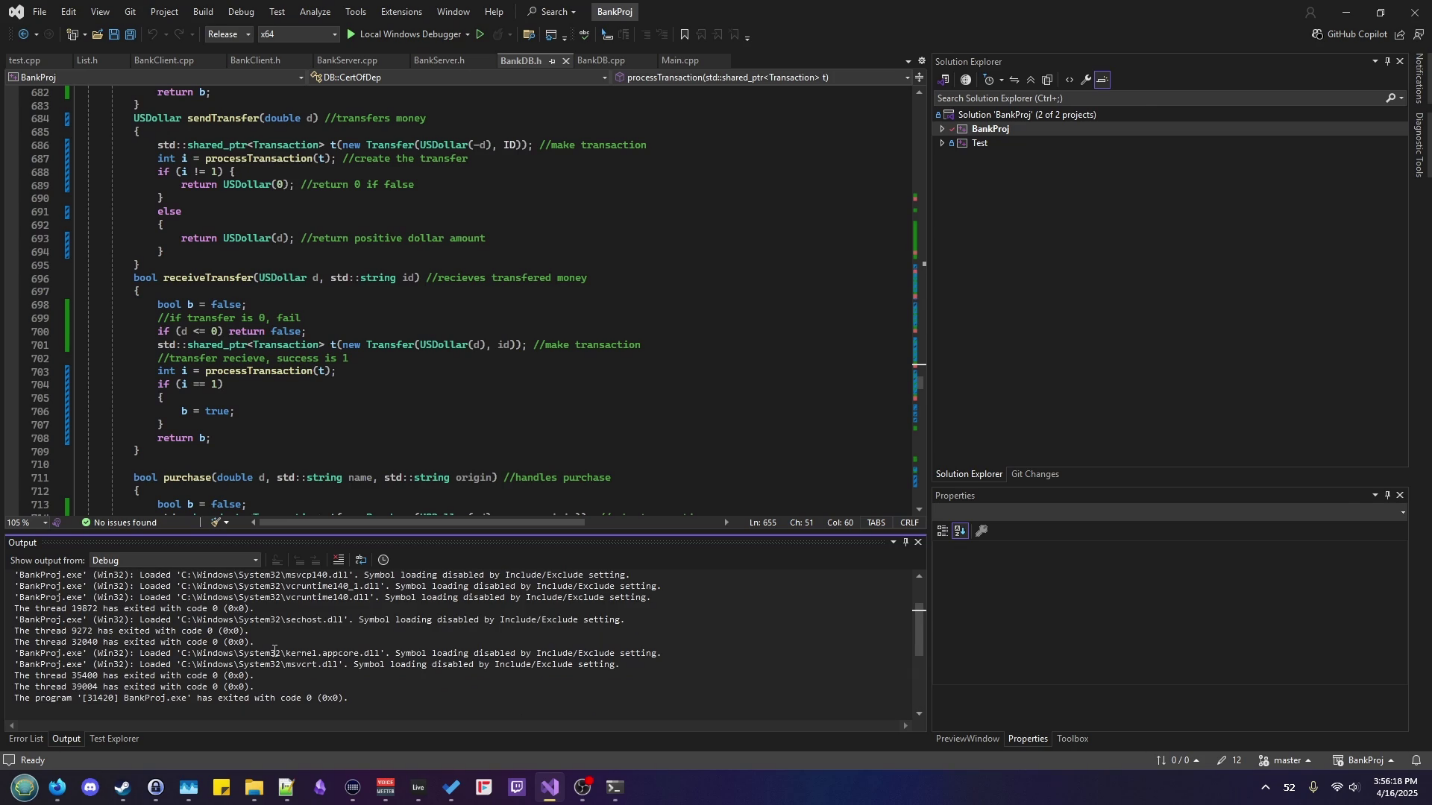
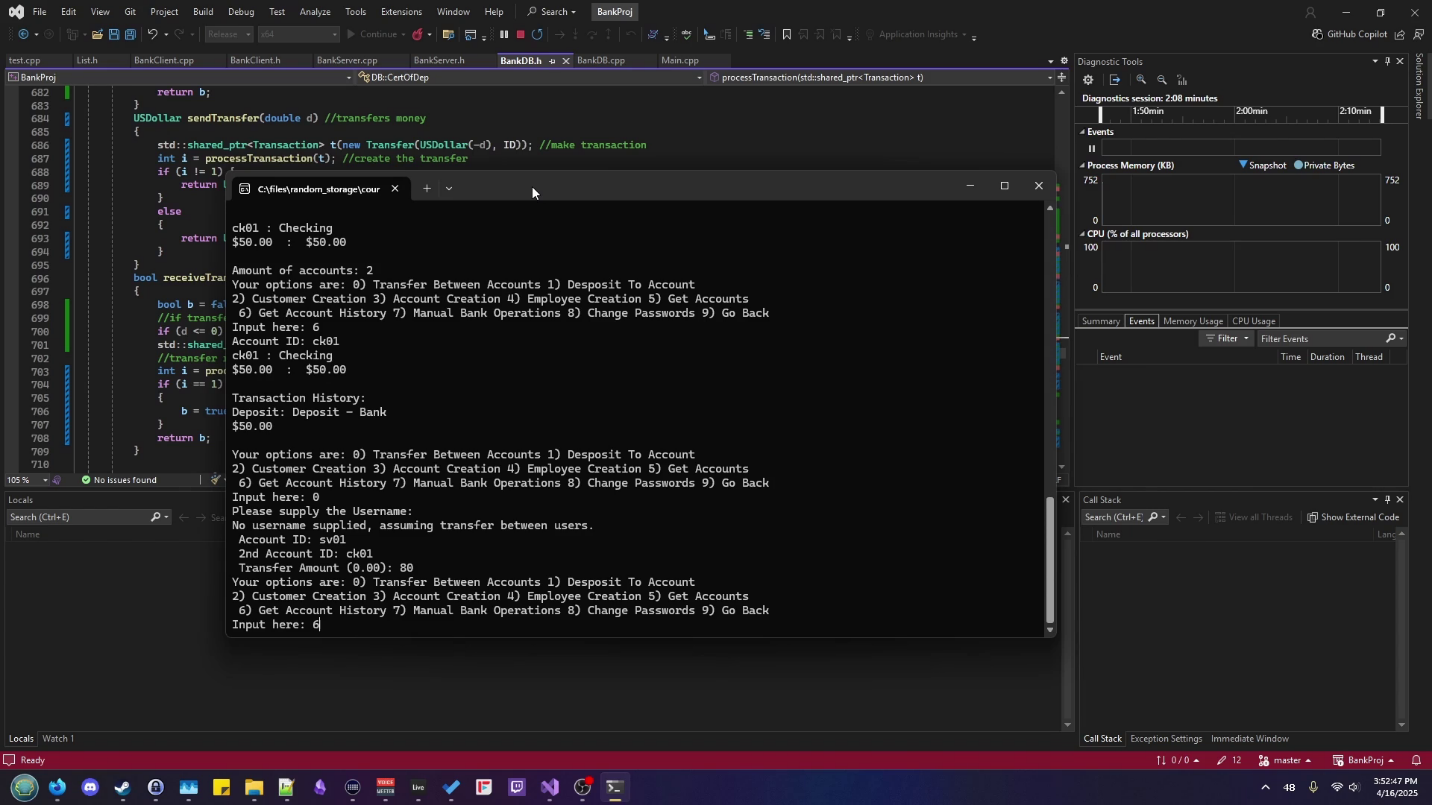
## Link

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

## Printscreens

A computer screen shot of a black screen

AI-generated content may be incorrect.



## Code

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);

d.reset();

}

/// <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 != 0)

{

\*j = count-1;

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

}

else

{

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

}

//return no matter what; false if nothing happened

iPointer.reset(); //clears the pointers

j.reset();

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 != 0)

{

\*j = count-1;

return tail->get(iPointer, j);

}

else

{

return head->get(iPointer, j);

}

iPointer.reset(); //clears the pointers

j.reset();

}

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 != 0)

{

\*j = count;

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

}

else

{

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

}

if (b) count++;

iPointer.reset(); //clears the pointers

j.reset();

}

catch (Exception& ex)

{

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

ex.printError();

}

d.reset(); //drop the data pointer cause we don't need it anymore

//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 != 0)

{

\*j = count-1;

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

}

else

{

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

}

//update the count

updateCount();

iPointer.reset(); //clears the pointers

j.reset();

}

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 != 0)

{

\*j = count-1;

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

}

else

{

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

}

iPointer.reset(); //clears the pointers

j.reset();

}

catch (Exception& ex)

{

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

ex.printError();

}

d.reset(); //drop the data pointer cause we don't need it anymore

//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;

}

};

Exception.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);

d.reset();

}

/// <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 != 0)

{

\*j = count-1;

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

}

else

{

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

}

//return no matter what; false if nothing happened

iPointer.reset(); //clears the pointers

j.reset();

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 != 0)

{

\*j = count-1;

return tail->get(iPointer, j);

}

else

{

return head->get(iPointer, j);

}

iPointer.reset(); //clears the pointers

j.reset();

}

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 != 0)

{

\*j = count;

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

}

else

{

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

}

if (b) count++;

iPointer.reset(); //clears the pointers

j.reset();

}

catch (Exception& ex)

{

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

ex.printError();

}

d.reset(); //drop the data pointer cause we don't need it anymore

//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 != 0)

{

\*j = count-1;

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

}

else

{

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

}

//update the count

updateCount();

iPointer.reset(); //clears the pointers

j.reset();

}

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 != 0)

{

\*j = count-1;

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

}

else

{

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

}

iPointer.reset(); //clears the pointers

j.reset();

}

catch (Exception& ex)

{

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

ex.printError();

}

d.reset(); //drop the data pointer cause we don't need it anymore

//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;

}

};

Encrypt.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);

d.reset();

}

/// <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 != 0)

{

\*j = count-1;

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

}

else

{

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

}

//return no matter what; false if nothing happened

iPointer.reset(); //clears the pointers

j.reset();

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 != 0)

{

\*j = count-1;

return tail->get(iPointer, j);

}

else

{

return head->get(iPointer, j);

}

iPointer.reset(); //clears the pointers

j.reset();

}

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 != 0)

{

\*j = count;

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

}

else

{

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

}

if (b) count++;

iPointer.reset(); //clears the pointers

j.reset();

}

catch (Exception& ex)

{

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

ex.printError();

}

d.reset(); //drop the data pointer cause we don't need it anymore

//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 != 0)

{

\*j = count-1;

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

}

else

{

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

}

//update the count

updateCount();

iPointer.reset(); //clears the pointers

j.reset();

}

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 != 0)

{

\*j = count-1;

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

}

else

{

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

}

iPointer.reset(); //clears the pointers

j.reset();

}

catch (Exception& ex)

{

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

ex.printError();

}

d.reset(); //drop the data pointer cause we don't need it anymore

//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;

}

};

Encrypt.cpp

#include "Encrypt.h"

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 MnuChangePassword : public Menu

{

public:

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

{

user = us;

pass = pss;

logic();

}

~MnuChangePassword() {

}

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;

};

class MnuManualBankOperations : public Menu

{

public:

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

{

user = us;

pass = pss;

logic();

}

~MnuManualBankOperations() {

}

void logic();

//strings for logic

std::string user;

std::string pass;

};

}

BankClient.cpp

#include "BankServer.h"

#include "BankClient.h"

#include <iostream>

using namespace Client;

//next 2 functions are for terminal-related security. These should be unneeded when switching to a better UI in the future

/// <summary>

/// Clears output in modern terminals

/// </summary>

void clearScreenANSI() {

std::cout << "\033[2J\033[1;1H"; //uses ANSI Escape codes to clear the console, when supported

}

/// <summary>

/// Clear terminal on specific systems. This is not ideal (system shell calls aren't great), but it'll work in our test CMD

/// </summary>

void clearScreen() {

#ifdef \_WIN32

system("cls");

#else

system("clear");

#endif

}

//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)

{

clearScreen();

clearScreenANSI();

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)

{

clearScreen();

clearScreenANSI();

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) Password Change 4) 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)

{

MnuChangePassword ch(user, pass); //change password

}

else if (i == 4)

{

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 5) Get Accounts\n 6) Get Account History 7) Manual Bank Operations 8) Change Passwords 9) 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

clearScreen();

clearScreenANSI();

}

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

clearScreen();

clearScreenANSI();

}

else if (i == 5)

{

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

}

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)

{

MnuManualBankOperations bo(user,pass); //manual operations

}

else if (i == 8)

{

MnuChangePassword ch(user, pass); //change password

}

else if (i == 9)

{

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

//get the deposit amount, convert to double

double d = 1;

try

{

d = stod(TextInput(" Deposit Amount (0.00): "));

}

catch (...)

{

std::cout << "Not a number or an error occured.";

return;

}

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

{

create = false;

}

else

{

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

}

}

}

}

/// <summary>

/// allows changing passwords

/// </summary>

void MnuChangePassword::logic()

{

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

{

std::cout << "You are changing the password as an Employee\nPlease supply the";

int type = DynamicOptions(TextInput(" Account Type (O, Customer; 1, Current Employee): "));

if (type == 0)

{

std::cout << "You are changing a customer's password. Please supply";

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

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

if (server->userPassword(userC, passwordC, 1))

{

clearScreen();

clearScreenANSI();

std::cout << "You have successfully changed a password.\n";

}

else

{

clearScreen();

clearScreenANSI();

std::cout << "The change was not successful.\n";

}

}

else if (type == 1)

{

std::cout << "You are changing your own password. Please supply";

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

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

if (passwordOld == pass)

{

if (server->userPassword(user, passwordNew, 2))

{

clearScreen();

clearScreenANSI();

std::cout << "You have successfully changed your password.\n";

}

else

{

clearScreen();

clearScreenANSI();

std::cout << "The change was not successful\n";

}

}

else

{

std::cout << "Credentials problem, exiting.";

}

}

else

{

std::cout << "Did not pick a valid option, exiting.";

}

}

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

{

std::cout << "You are changing the password as a Customer\nPlease supply the";

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

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

if (passwordOld == pass)

{

if (server->userPassword(user, passwordNew, 0))

{

clearScreen();

clearScreenANSI();

std::cout << "You have successfully changed your password.\n";

}

else

{

clearScreen();

clearScreenANSI();

std::cout << "The change was not successful\n";

}

}

else

{

std::cout << "Credentials problem, exiting.\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

if (server->accountsCount(user) < 1) create = false; //fail if we don't have any accounts

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): "));

//get the deposit amount, convert to double

double d = 1;

try

{

d = stod(TextInput(" Deposit Amount (0.00): "));

}

catch (...)

{

std::cout << "Not a number or an error occured.";

return;

}

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

if (userC == "")

{

userC = user;

std::cout << "No username supplied, assuming transfer between users.\n";

}

}

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

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

//get the transfer amount, convert to double

double d = 1;

try

{

d = stod(TextInput(" Transfer Amount (0.00): "));

}

catch (...)

{

std::cout << "Not a number or an error occured.";

return;

}

if (d < 0) d = -d; //make sure it's positive

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

//get the deposit amount, convert to double

double d = 1;

try

{

d = stod(TextInput(" Deposit Amount (0.00): "));

}

catch (...)

{

std::cout << "Not a number or an error occured.";

return;

}

if (d < 0) d = -d; //make sure it's positive

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)

{

std::cout << server->accountsDisplay(user); //display the accounts

std::cout << "Amount of accounts: " << server->accountsCount(user) << "\n"; //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);

}

}

/// <summary>

/// While task scheduling & online processing is not implemented, handle purchases & ongoing bank functions here

/// </summary>

void MnuManualBankOperations::logic()

{

if (server->userValidation(user, pass) == 1) //make sure you're an employee

{

std::cout << "Manual bank functions are available.\n";

bool running = true;

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

while (running)

{

std::cout << "Your options are: 0) Purchases 1) Recurring Tasks (Interest) 2) Exit\n";

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

if (i == 0)

{

std::string us = TextInput(" Username: "); //username

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

//get the purchase amount, convert to double

double d = 1;

try

{

d = stod(TextInput(" Purchase Amount (0.00): "));

}

catch (...)

{

std::cout << "Not a number or an error occured.";

return;

}

if (d < 0) d = -d; //make sure it's positive

std::string originAndName = "Manual Purchase";

server->purchase(us,acc,d,originAndName,originAndName); //purchase

}

else if (i == 1)

{

std::cout << "Running banking tasks.";

server->runBankProccesses(); //run said banking tasks

}

else if (i == 2)

{

running = false; //end the loop

}

else

{

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

}

}

}

}

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);

///allows users to change passwords; very important because there is a hardcoded default employee

bool userPassword(std::string user, std::string pass, int type);

//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);

//purchase

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

void runBankProccesses();

};

}

BankServer.cpp

#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);

///allows users to change passwords; very important because there is a hardcoded default employee

bool userPassword(std::string user, std::string pass, int type);

//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);

//purchase

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

void runBankProccesses();

};

}

#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::Transaction> t(new DB::Deposit(deposit));

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

t.reset(); //clear extra transaction early

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>

/// Allows users to change passwords

/// </summary>

/// <param name="user">User to change</param>

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

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

bool Server::userPassword(std::string user, std::string pass, int type)

{

bool b = false;

int i = -1;

int j = -1;

switch (type)

{

case 2:

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

if (j != -1)

{

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

if (e)

{

if (e->password != pass) //make sure to soft error if they set the same pass again

{

b = true;

e->password = pass;

}

}

}

break;

default:

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) //make sure to soft error if they set the same pass again

{

b = true;

c->password = pass;

}

}

}

break;

}

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));

a->setInterestType(4); //normal interest

break;

case 1:

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

a->setInterestType(0); //no interest

break;

case 2:

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

a->setInterestType(8); //extra good interest

break;

case 3:

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

a->setInterestType(6); //daily, okay interest

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);

}

}

else

{

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

if (e)

{

b = e->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)

{

std::string s = a->preview();

return s; //return display

}

}

}

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 "";

}

/// <summary>

/// Purchases

/// </summary>

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

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

/// <param name="val">purchase value</param>

/// <param name="name">Purchase information</param>

/// <param name="origin">Purchase origin</param>

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

bool Server::purchase(std::string user, std::string acc, double val, std::string name, std::string origin)

{

bool b = false;

int i = db->Customers.find(user); //make sure user exists

if (i != -1)

{

b = db->purchase(acc, user, val, db, name, origin); //pass purchase to DB (has its own function for overdraft)

}

return b;

}

/// <summary>

/// runs the bank processes

/// </summary>

void Server::runBankProccesses()

{

db->bankProcesses(); //uses the database bank processes function

}

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

t.reset(); //clear extra pointer

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

//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;

}

t.reset(); //clear the pointer

}

}

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)

}

t.reset(); //reset pointer

}

return s;

}

/// <summary>

/// Sets the interest information; currently hard coded

/// </summary>

/// <param name="setting">int setting code, should be 0-9</param>

void setInterestType(int setting)

{

switch (setting)

{

case 0:

APY = 0; //interest rate

interestType = 0; //0: None

payoutRate = 0; //0: None

break;

case 1:

APY = 0.5; //interest rate

interestType = 1; // 1: Simple

payoutRate = 0; //0: Yearly

break;

case 2:

APY = 0.5; //interest rate

interestType = 2; // 2: Compound Yearly

payoutRate = 0; //0: Yearly/None

break;

case 3:

APY = 0.5; //interest rate

interestType = 3; // 3: Compound Monthly

payoutRate = 1; // 1: Every 6 months

break;

case 4:

APY = 0.5; //interest rate

interestType = 3; // 3: Compound Monthly

payoutRate = 2; // 2: monthly

break;

case 5:

APY = 0.5; //interest rate

interestType = 4; // 4: Compound Daily

payoutRate = 2; // 2: monthly

break;

case 6:

APY = 0.5; //interest rate

interestType = 4; // 4: Compound Daily

payoutRate = 3; // 3: daily

break;

case 7:

APY = 2; //interest rate

interestType = 4; // 4: Compound Daily

payoutRate = 3; // 3: daily

break;

case 8:

APY = 5; //interest rate

interestType = 4; // 4: Compound Daily

payoutRate = 0; //0: Yearly

break;

case 9:

APY = 10; //interest rate

interestType = 1; // 1: Simple

payoutRate = 3; // 3: daily

break;

default:

break;

}

}

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

{

return "Account";

}

protected:

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

};

/// <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 bool 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 bool 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() {}

bool deposit(double d) //deposits money

{

bool b = false; //make return

std::shared\_ptr<Transaction> t(new Deposit(USDollar(d))); //make transaction

int i = processTransaction(t); //atempt the process

if (i == 1)

{

b = true;

}

return b;

}

USDollar sendTransfer(double d) //transfers money

{

std::shared\_ptr<Transaction> t(new Transfer(USDollar(-d), ID)); //make transaction

int i = processTransaction(t); //create the transfer

if (i != 1) {

return USDollar(0); //return 0 if false

}

else

{

return USDollar(d); //return positive dollar amount

}

}

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

{

bool b = false;

//if transfer is 0, fail

if (d <= 0) return false;

std::shared\_ptr<Transaction> t(new Transfer(USDollar(d), id)); //make transaction

//transfer recieve, success is 1

int i = processTransaction(t);

if (i == 1)

{

b = true;

}

return b;

}

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

{

bool b = false;

std::shared\_ptr<Transaction> t(new Purchase(USDollar(-d), name, origin)); //make transaction

//purchase success is 1

int i = processTransaction(t);

if (i == 1)

{

b = true;

}

return b;

}

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

//check if dollar is 0 or not

if (t->Val != 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() {}

bool deposit(double d) //deposits money

{

bool b = false; //make return

std::shared\_ptr<Transaction> t(new Deposit(USDollar(d))); //make transaction

int i = processTransaction(t); //atempt the process

if (i == 1)

{

b = true;

}

return b;

}

USDollar sendTransfer(double d) //transfers money

{

std::shared\_ptr<Transaction> t(new Transfer(USDollar(-d), ID)); //make transaction

int i = processTransaction(t); //create the transfer

if (i != 1) {

return USDollar(0); //return 0 if false

}

else

{

return USDollar(d); //return positive dollar amount

}

}

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

{

bool b = false;

//if transfer is 0, fail

if (d <= 0) return false;

std::shared\_ptr<Transaction> t(new Transfer(USDollar(d), id)); //make transaction

//transfer recieve, success is 1

int i = processTransaction(t);

if (i == 1)

{

b = true;

}

return b;

}

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

{

bool b = false;

std::shared\_ptr<Transaction> t(new Purchase(USDollar(-d), name, origin)); //make transaction

//purchase success is 1

int i = processTransaction(t);

if (i == 1)

{

b = true;

}

return b;

}

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

//check if dollar is 0 or not

if (t->Val != 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;

bool deposit(double d) //deposits money

{

bool b = false; //make return

std::shared\_ptr<Transaction> t(new Deposit(USDollar(d))); //make transaction

int i = processTransaction(t); //atempt the process

if (i == 1)

{

b = true;

}

return b;

}

USDollar sendTransfer(double d) //transfers money

{

std::shared\_ptr<Transaction> t(new Transfer(USDollar(-d), ID)); //make transaction

int i = processTransaction(t); //create the transfer

if (i != 1) {

return USDollar(0); //return 0 if false

}

else

{

return USDollar(d); //return positive dollar amount

}

}

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

{

bool b = false;

//if transfer is 0, fail

if (d <= 0) return false;

std::shared\_ptr<Transaction> t(new Transfer(USDollar(d), id)); //make transaction

//transfer recieve, success is 1

int i = processTransaction(t);

if (i == 1)

{

b = true;

}

return b;

}

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

{

bool b = false;

std::shared\_ptr<Transaction> t(new Purchase(USDollar(-d), name, origin)); //make transaction

//purchase success is 1

int i = processTransaction(t);

if (i == 1)

{

b = true;

}

return b;

}

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

//check if dollar is 0 or not

if (t->Val != 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() {}

bool deposit(double d) //deposits money

{

bool b = false; //make return

std::shared\_ptr<Transaction> t(new Deposit(USDollar(d))); //make transaction

int i = processTransaction(t); //atempt the process

if (i == 1)

{

b = true;

}

return b;

}

USDollar sendTransfer(double d) //transfers money

{

std::shared\_ptr<Transaction> t(new Transfer(USDollar(-d), ID)); //make transaction

int i = processTransaction(t); //create the transfer

if (i != 1) {

return USDollar(0); //return 0 if false

}

else

{

return USDollar(d); //return positive dollar amount

}

}

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

{

bool b = false;

//if transfer is 0, fail

if (d <= 0) return false;

std::shared\_ptr<Transaction> t(new Transfer(USDollar(d), id)); //make transaction

//transfer recieve, success is 1

int i = processTransaction(t);

if (i == 1)

{

b = true;

}

return b;

}

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

{

bool b = false;

std::shared\_ptr<Transaction> t(new Purchase(USDollar(-d), name, origin)); //make transaction

//purchase success is 1

int i = processTransaction(t);

if (i == 1)

{

b = true;

}

return b;

}

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

//check if dollar is 0 or not

if (t->Val != 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

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

//Customer deposit logic

bool 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

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

bool 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::string user , 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

trans.reset(); //clear extra shared\_ptr

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);

}

}

acc.reset(); //clears that particular shared\_ptr

}

/// <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("Admin", "defaultPassPleaseChange"));

Customers = LinkedList<Customer>();

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

e.reset(); //clear pointer

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, double val, std::shared\_ptr<Database> db, std::string name = "Purchase", std::string origin = "Unknown")

{

bool b = false;

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

if (cust) //make sure customer is real

{

int i = cust->AccountIDs.find(acc); //find account in customer's list

if (i > -1)

{

std::shared\_ptr<Account> account = Accounts.get(acc); //make sure account exists

if (account)

{

b = account->purchase(val, name, origin); //purchase in account

}

account.reset(); //clear extra shared\_ptr

}

}

Overdraft::OnPurchase(user, db); //do overdraft

return b;

}

/// <summary>

/// bank processes done at a regular interval

/// </summary>

void bankProcesses()

{

Interest::AllAccounts(Accounts);

}

};

}

BankDB.cpp

#include "BankDB.h"

using namespace DB;

bool 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

if (Account1 && Account2)

{

bool b = Account2->receiveTransfer(Account1->sendTransfer(v), acc1);

return b;

}

}

else

{

return false;

}

}

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

{

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

{

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

if (accountToDep)

{

bool b = accountToDep->deposit(v); //get deposit

return b; //return success/fail

}

}

else

{

return false;

}

}

bool 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

if (Account1 && Account2)

{

bool b = Account2->receiveTransfer(Account1->sendTransfer(v), acc1);

return b;

}

}

else

{

return false;

}

}

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

{

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

{

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

return accountToDep->deposit(v);

}

else

{

return false;

}

}

/// <summary>

/// Handles overdraft for a specific user

/// </summary>

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

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

bool Overdraft::OnPurchase(std::string user, std::shared\_ptr<Database> d)

{

bool success = false;

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

if (cust)

{

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;

std::string accID = "";

if (cust->AccountIDs.get(i))

{

accID = \*cust->AccountIDs.get(i);

}

a = d->Accounts.get(accID);

//check if we actually got the account

if (a)

{

//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;

std::string accID2 = "";

if (cust->AccountIDs.get(j))

{

accID2 = \*cust->AccountIDs.get(j);

}

b = d->Accounts.get(accID2);

//make sure b exists

if (b)

{

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

if (!(accID2 == accID))

{

//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, accID2, accID, 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

}

## Future Improvement Plan

### ASAP

Password hiding needs to be implemented. I have opted to, for now, just clear the console after logon. This is not ideal, but there is not really a clean way to do this. The methods involved are heavily platform & compiler dependent and will have to be handled accordingly.

Pick a security library of client’s choice & use that to roll hashing & encryption. Modern encryption & hashing are system-intensive functions, it is much more efficient to use an implementation that utilizes modern instruction sets to accelerate these tasks. Pre-existing libraries for security will also be heavily tested, in a way that an in-house implementation cannot match.

### Short-Term

Utilize threading & mutex/atomic values to create multiple threads, allowing the bank, server, & client to all run on their own individual loops. This would allow for daily services to be run automatically, periodic updates to displays, etc.

Deleting & updating transactions is not yet fully implemented, despite being possible under the LinkedList structure. This is largely due to the lack of proper access control, mentioned below. You don’t want a teller to mistakenly remove a whole deposit, for example.

Slowly, a large portion of the server code should be pushed over to the database side of things. Many of the operations inside the server are a collection of simple database operations strung together. As these are key banking functions, they should instead be part of the database class. I have already partially done this with deposit, transfer, & purchases, but more is needed.

Users need to be redone. More dynamic functionality should be implemented, & multiple tiers of access would be ideal. Customer, Business Customer, Teller, Trusted Bank Staff, Administrator is a more realistic set of access tiers. With proper access control & roles, you can do away with the separate lists & just have one big “Users” list.

### Long-Term

Separate client, server, & database into their own separate programs, utilizing basic networking for communication. None of the security considerations are that useful when its all run on the same hardware.

Debit handling is not at all implemented yet, and obviously in the current day that is needed.