Bank Teller Project

CS 303

Project 1

Project 4 pg. 354

William Freeman

9/25/2013

**Table of Contents**

**Assumptions 2**

**UML Class Diagram 3**

**Attribute and Function Tables 4**

**Algorithms 6**

**Decisions 9**

**Appendix 14**

**Citation 40**

**ASSUMPTIONS:**

The assumptions that were made about this project include:

* The bank teller may only help one customer at a time.
* Customers are allowed to make multiple transactions, much like real life where people sometimes forget that they had another check to deposit or to withdrawal.
* Customers arrive at random intervals, relying on a designated arrival chance.
* Each customer takes a certain amount of time per transaction, some people are slower than others.
* The program is to be a self-ran simulation with little user control.
* Customers have an initial starting balance before coming to the bank.
* There is a limit to the amount that a customer is able to withdrawal and deposit per transaction.
* Only one customer will be allowed to show up at a time, multiple customers may not join the line during the same time step.
* The user will determine when the simulation ends by entering the amount of customers that will be served before terminating the simulation.
* Each transaction will be recorded and sent to a file that contains information about the transaction.
* Customer names are chosen at random by a predetermined list contained within a text file.
* The simulation constraints will be stored as private member const variables, in order to limit the user from being able to “break” the simulation into an unrealistic state.

**UML CLASS DIAGRAM:**



**ATTRIBUTE AND FUNCTION TABLES:**

**Class BankTellerSim**

|  |  |
| --- | --- |
| Data Field | Attribute |
| stack<Customer> m\_waitLine | The stack containing customers in a bank line. |
| int m\_custServed | The number of customers that have been served. |
| vector<string> m\_firstNameList | Contains a list of first names obtained from FirstNames.txt. |
| vector<string> m\_lastNameList | Contains a list of last names obtained from LastNames.txt. |
| ofstream m\_report | A .csv file that contains information pertaining to each transaction made during the simulation. |
| static const int MAX\_TRANSACTIONS | The maximum number of transactions that a customer is allowed to make. |
| static const int MAX\_STARTING\_BAL | The maximum starting balance - 1 allowed for each customer. |
| const double ARRIVAL\_PROB | The probability that a customer will join the wait-line for a given time step. |
| Function | Behavior |
| void AddCustomer() | Generates a random double between 0 – 1 and determines if a customer will arrive by comparing the random double to ARRIVAL\_PROB. |
| void OutputWaitLine() const | Outputs the total number of customers that are left in line along with the name of the next customer up. |
| void LoadNames() | Fills the vectors that contain the names of potential customers by reading them in from the text files. |
| string SelectFirstName() const | Selects the first name of a new customer from m\_firstNameList by generating a random index. |
| string SelectLastName() const | Selects the last name of a new customer from m\_lastNameList by generating a random index. |
| void ServeCustomer() | Serves the next customer in line, determines the amount of transactions that the customer will make. Also calls AddCustomer() during each time step taken place when the customer is being served. |
| void ReportTransaction(Customer& cust, double startBal, double endBal) | Creates a report of every transaction that transpired during the simulation and sends it to a .csv file. |
| BankTellerSim() | Initializes private member variables & generates a header for the .csv report. |
| void RunSim(int num) | Runs the Bank Teller Simulation, num controls the number of customers that will be served before the simulation ends. |

**Class Customer**

|  |  |
| --- | --- |
| Data Field | Attribute |
| string m\_lastName | The last name of the customer. |
| string m\_firstName | The first name of the customer. |
| double m\_curBalance | The current account balance for a customer. |
| int m\_serveTime | The amount of time a customer will take per transaction. |
| Transaction m\_transaction | The transaction that the customer will make. |
| static const int MAX\_SERVE\_TIME | The maximum amount of time allowed, for the simulation, that a customer will take per transaction. |
| Function | Behavior |
| Customer() | Sets default values for private member variables. |
| Customer(string first, string last) | Assigns first to m\_firstName, last to m\_lastName, and initializes the other private member variables to default values. |
| Customer(string first, string last, double bal) | Assigns first to m\_firstName, last to m\_lastName, bal to m\_curBalance, and generates m\_serveTime. |
| double GetCurBal() const | Returns the value of the customer’s bank account. |
| string GetName() const | Returns a concatenation of the customer’s name in a last, first format. |
| int GetServeTIme() const | Returns the value of m\_serveTime. |
| void MakeTransaction() | Runs a transaction for the customer, updates m\_curBalance based upon the transaction type. |
| Customer& operator=(const Customer& cust) | Assignment operator overload. Ensures that private member variables get assigned properly. |

**Class Transaction**

|  |  |
| --- | --- |
| Data Field | Attribute |
| bool m\_withdrawal | A bool value that determines whether the transaction is a deposit or withdrawal. |
| double m\_amount | The numerical amount of the transaction. |
| static const int MAX\_WITHDRAWAL | The maximum amount that a customer is allowed to withdrawal from their account. |
| static const int MAX\_DEPOSIT | The maximum amount that a customer is allowed to deposit in one transaction. |
| const double WITHDRAWAL\_PROB | The probability that a transaction will be a withdrawal and not a deposit. |
| Function | Behavior |
| void DetermineTransType() | Generates a random double between 0 – 1 and compares that with WITHDRAWAL\_PROB to determine the transaction type. |
| Transaction() | Sets default values for private member variables. |
| double RunTransaction() | Determines the transaction type and returns the amount to add or subtract from the customer’s account. |
| Transaction& operator=(const Transaction& trans) | Assignment operator overload. Ensures that private member variables get assigned properly. |

**ALGORITHMS:**

**void AddCustomer()**

1. randNum = random double between 0 – 1
2. if (randNum <= ARRIVAL\_PROB)
3. Create a new Customer, along with the attributes for the Customer, & add them to m\_waitLine.

**void OutputWaitLine()**

1. If (m\_waitLine is empty)
2. Notify the user that there are no customers in line.
3. Else, Output the size of m\_waitLine & the name of the top Customer in the stack.

**void LoadNames()**

1. Open “FirstNames.txt”.
2. Add names to m\_firstNameList while there are names to add.
3. Close “FirstNames.txt” & open “LastNames.txt”.
4. Add names to m\_lastNameList while there are names to add.
5. Close “LastNames.txt”.

**string SelectFirstName() & string SelectLastName()**

1. int randomIndex = rand(0 – (m\_firstNameList.size() – 1)) [or m\_lastNameList.size() – 1 for SelectLastName()].
2. Return the value at randomIndex.

**void ServeCustomer()**

1. Assign the top Customer from the stack to a variable & pop them from the stack.
2. Determine the number of transactions the customer make with a random number.
3. For each transaction:
4. Obtain the initial balance & the serve time of the customer.
5. Customer makes the transaction.
6. Obtain the end balance after the transaction & call ReportTransaction().
7. For the serve time of the customer:
   1. A new Customer might join m\_waitLine, call AddCustomer().
8. m\_custServed++

**void RunSim(int num)**

1. while (m\_custServed < num)
2. if (m\_waitLine is empty), call AddCustomer().
3. Else, serve the next customer, call ServeCustomer().
4. If (m\_custServed is a multiple of 5), call OutputWaitLine().
5. Close m\_report.

**void MakeTransaction()**

1. while (m\_curBalance + transactionAmount < 0)
2. Call RunTransaction().
3. Update m\_curBalance.

**void DetermineTransType()**

1. randDub = random double between 0 – 1
2. if (randDub <= WITHDRAWAL\_PROB), m\_withdrawal = true.
3. Else, transaction is a deposit, m\_withdrawal = false.

**double RunTransaction()**

1. Call DetermineTransType()
2. If (m\_withdrawal)
3. m\_amount = 0 – (random double between 1 – MAX\_WITHDRAWAL)
4. if (m\_amount != -MAX\_WITHDRAWAL), can add change to the withdrawal.
5. m\_amount -= random double between 0 – 1
6. return m\_amount
7. Else, the transaction is a deposit.
8. m\_amount = random double between 1 – MAX\_DEPOSIT
9. if (m\_amount != MAX\_DEPOSIT), can add change to the deposit.
10. M\_amount += random double between 0 – 1
11. Return m\_amount

**DECISIONS:**

The first design decision made was whether or not to have the customers be controlled by the user or to have the simulation be run automatically with little or no user control. I decided to go with the latter of the two, having the program simulation run with very little user control. Thus making this program a more accurate simulation instead of a user driven program and preventing user runtime error. But, the user will have control over the number of customers that the teller will be able to serve before the simulation ends. I decided to include a check of the user’s input value to make sure that the user enters at least an integer greater than 5, in order to allow the program to display the required stack size and the next customer waiting in the stack at least once.

The very next design decision was to construct the classes to use within the program. It was obvious that there needed to be a Transaction and Customer class that must contain specific values called for by the assignment. Namely, the transaction type, and amount for Transaction class, and the customer’s name and current balance for the Customer class. I determined that there needed to be one more class which will control the simulation, which I called BankTellerSim class. This would be the primary class that will control the entire simulation and contain an instance of a Customer class using composition. The Customer class would in turn contain an instance of class Transaction using composition. This would be the determined hierarchy of classes.

I started designing the classes from the bottom up, starting with Transaction and ending with BankTellerSim, which allowed for an easier time debugging and testing. The first function implemented was a function to determine if the transaction would be a deposit or withdrawal. Following the algorithm for DetermineTransType(), I followed a similar style to the example found in *Objects, Abstraction, Data Structures and Design Using C++* by Elliot B. Koffman & Paul A.T. Wolfgang on pg. 396, Listing 6.8. That is where I obtained the code:

double randNum = double(rand()) / RAND\_MAX;

This code snippet generates a decimal value between 0 - 1 which was compared to a predetermined value, WITHDRAWAL\_PROB, to determine if the transaction would be a withdrawal or a deposit. I decided to set WITHDRAWAL\_PROB to be 0.25 which should allow for a withdrawal to be made about every four transactions. It makes logical sense that customers will want to deposit money more often than making a withdrawal.

The very next major function that needed implemented was a function that would update the amount in the customer’s account when they make a transaction. This is where RunTransaction() was implemented. Since I had already decided that there would be a private member function for the Customer class that contained their current bank balance, I decided to have RunTransaction() return a double value. To determine what value to either add or subtract from the customer’s bank account first I had to determine whether or not the transaction would be a deposit or withdrawal. I used the member variable m\_amount in class Transaction to accomplish this task. If the transaction was a deposit, m\_amount would be positive, negative if the transaction is a withdrawal. I decided to limit the amount that a customer would be allowed to withdrawal and deposit to and from their account. I wanted to keep the simulation realistic and not have values that were completely absurd for a withdrawal or deposit amount. I used a pseudo-random number generator to generate an int, between 1 and the max parameter accordingly, then converted it to a double type. As seen in the algorithm for RunTransaction(), if there was room to add change to the deposit or withdrawal, I used the code snippet found in the textbook to generate a decimal value between 0 – 1 and added (or subtracted in the case of a withdrawal) it to m\_amount.

The next class to be implemented was the Customer class. Before creating the Customer class, I decided that each customer will have a specific serve time for each transaction, allowing for other customers to be able to arrive when one is being served at the teller. I stored this value as an int and once again set a limit to the amount of time a customer will take for each transaction using MAX\_SERVE\_TIME, which was set to a value of 5. There was obvious need for a few GetValue() functions which were implemented for the customer’s name, serve time, and account balance. The only major function that I decided needed to be implemented for the Customer class was MakeTransaction(). This function would essentially continue to call RunTransaction(), using the instance of class Transaction found within class Customer, until the value of the customer’s account added to the value of the transaction was not negative. The reasoning behind this was to ensure that no customer’s account would ever go negative. Once again I did not want this simulation to contain unrealistic results. After a proper value is obtained the function would simply just update the value of the customer’s bank account.

The final class to be implemented was BankTellerSim class. This class would be the class that drives the entire simulation. It would be responsible for adding new customers to the line, serving customers one at a time, and allowing for a customer’s time taking during their multiple transactions. Since the program is designed to be a self-ran simulation and since every customer needed a name, I decided to create a few text files containing both first and last names. Instead of opening and closing these text files every time a new customer would join the line, I decided to store the names in a data structure. Thus, only needing to open these text files only once during each program run. The best choice for the data structure to store the names in was the vector structure. The reason for this was the vector’s ability for random access, since the simulation would pick a name at random for each customer. Both functions, SelectFirstName() and SelectLastName() do just that, just from two different lists of names.

Once the customers were able to be named, I implemented the main function, RunSim(int num), which controls the entire simulation and would be the only function that the user would directly interact with. The parameter num would be used to control when the simulation would end once the num amount of customers have been served. A requirement of the assignment was to output the number of customers in line and the name of the next customer to be served every 5 customers have been served. I used this line of code to allow for this requirement:

if (m\_custServed % 5 == 0)

{

//Output every 5 customers served.

OutputWaitLine();

}

This snippet of code ensures that if m\_custServed is a multiple of 5 then the program will output the number of customers in line along with the name of the next customer in line with the function OutputWaitLIne().

While RunSim(int num) is the function in which the user interacts with, ServeCustomer() is the main function that does all the work of the simulation behind the scenes. ServeCustomer() is the bulk of the simulation. It not only updates the customer’s bank balance by calling MakeTransaction() for the instance of the topmost Customer from the stack, it accounts for that customer’s serve time for each transaction. During these time steps ServeCustomer() will call AddCustomer() which accounts for a customer that might arrive while a transaction is being made. ServeCustomer() will also keep tabs on the customer’s account balance before and after each transaction and then send a report of these values to ReportTransaction(). ReportTransaction() outputs information about each transaction to a .csv file which allowed me to examine the simulation results and determine if the simulation is running properly. Back in RunSim(int num) I had to ensure that ServeCustomer() was not called when m\_waitLine is empty, otherwise there would be a memory-access runtime error. Therefore, if m\_waitLine was empty RunSim(int num) would just call AddCustomer() until m\_waitLine is no longer empty.

After a few testing and debugging sessions, the simulation was up and running properly. The last step in my program design was tweaking the max parameters and probabilities for the random events to a proper value. I had to tweak the ARRIVAL\_PROB to a value where the wait-line was not always empty or filling up so quickly that there was an unrealistic number of customers joining during the simulation. Once the final value was determined the end result was a simulation that ran and provided realistic results to a real life event. I decided to keep these max value and probabilities away from the user so that the user would not have the ability to “break” the simulation producing an unrealistic simulation. In a program like this, I feel the only person who should be allowed to tweak the simulation would be the designer themselves. This allows for the program to be user friendly and easy to prevent errors from occurring.

**APPENDEX:**

**SOURCE:**

**main.cpp**

/\*William Freeman

CS303

9/26/2013

Project 1 - Bank Teller Simulation

Description: Create a program that simulates a bank teller serving customers

which are stored in a stack. The teller will serve the last customer that

arrives first. Customers are allowed to withdrawal or deposit when they are

being served. The user specifies how many customers they would like served

before the simulation ends.\*/

#include <iostream>

#include "BankTellerSim.h"

using std::cout;

using std::cin;

const int MIN\_NUM\_CUSTOMERS = 5;

//Min number of customers that the simulation must serve.

void DetermineServed(int& num);

/\*Prompts the user to enter the number of customers they would like to serve

before the simulation ends.

@Param num - The number of customers to be served.\*/

int main()

{

int custServed;

BankTellerSim bankSim;

DetermineServed(custServed);

bankSim.RunSim(custServed);

return 0;

}

void DetermineServed(int& num)

{

//Greeting.

cout << "Bank Teller Simulation:\n";

cout << "Simulates customers in line at a bank, but instead of first-come,\n";

cout << "first-serve, this bank uses the philosophy of\n";

cout << "last-in, first-served when dealing with customers.\n\n";

do

{

//Prompt user.

cout << "Enter the number of customers you would like to serve before\n";

cout << "the simulation ends (must be more than 4 customers): ";

cin >> num;

if (num < MIN\_NUM\_CUSTOMERS)

{

//Bad input.

cout << "\nInvalid number of customers to serve entered.\n\n";

}

} while (num < MIN\_NUM\_CUSTOMERS);

}

**BankTellerSim.h**

/\*William Freeman

CS303

9/26/2013

Project 1 - Bank Teller Simulation

Description: Create a program that simulates a bank teller serving customers

which are stored in a stack. The teller will serve the last customer that

arrives first. Customers are allowed to withdrawal or deposit when they are

being served. The user specifies how many customers they would like served

before the simulation ends.

--BankTellerSim.h is the header file for the BankTellerSim class--

Class Desciption: BankTellerSim controls the main portion of the simulation.\*/

#ifndef \_BANK\_TELLER\_SIM\_H

#define \_BANK\_TELLER\_SIM\_H

#include <stack>

#include <vector>

#include <fstream>

#include "Customer.h"

using std::stack;

using std::vector;

using std::ofstream;

using std::endl;

class BankTellerSim

{

private:

//Member variables

stack<Customer> m\_waitLine;

int m\_custServed;

vector<string> m\_firstNameList;

vector<string> m\_lastNameList;

ofstream m\_report;

//Static & const variables

//Max num of transactions allowed per Customer.

static const int MAX\_TRANSACTIONS = 3;

//Max starting balance allowed for a Customer + 1.

static const int MAX\_STARTING\_BAL = 10001;

const double ARRIVAL\_PROB; //Arrival rate of Customers.

//Member functions

/\*Generates a random double between 0 - 1 and determines if a Customer will

join the wait line.\*/

void AddCustomer();

/\*Outputs the total number of Customers that are left wiating in line along

with the name of the next customer to be served.\*/

void OutputWaitLine() const;

/\*Fills the vectors, m\_firstNameList & m\_lastNameList with names from the txt

files that contain these names.\*/

void LoadNames();

/\*Generates a random int between 0 - m\_firstNameList.size() - 1.

Returns a random name from m\_firstNameList.\*/

string SelectFirstName() const;

/\*Generates a random int between 0 - m\_lastNameList.size() - 1.

Returns a random name from m\_lastNameList.\*/

string SelectLastName() const;

/\*Serves the next Customer in line. Determines the amount of Transactions that

Customer will make. Takes into account the amout of time that Customer takes

per Transaction and calls AddCustomer() during each time tick.\*/

void ServeCustomer();

/\*Creates a report of each and every Transaction made in a .csv file.

Includes details about the Transaction in the format:

LAST,FIRST,START\_BAL,TRANS\_TYPE,TRANS\_AMT,CUR\_BAL.\*/

void ReportTransaction(Customer& cust, double startBal, double endBal);

public:

//Constructors

/\*Default constructor.

Initalizes member variables to default values for an instance of class

BankTellerSim. Also loads in the list of names for Customers from the files

FirstNames.txt & LastNames.txt.

Creates the header to a .csv report file.\*/

BankTellerSim():

m\_custServed(0), ARRIVAL\_PROB(0.15)

{

LoadNames();

m\_report.open("Transaction Report.csv");

m\_report << "LAST,FIRST,START\_BAL,TRANS\_TYPE,TRANS\_AMT,CUR\_BAL" << endl;

}

//Member functions

/\*Runs the Bank Teller Simulation until the specified number of customers

have been served.

@param num - the number of Customers to be served before the simulation

ends.\*/

void RunSim(int num);

}; //class BankTellerSim

#endif //\_BANK\_TELLER\_SIM\_H

**BankTellerSim.cpp**

/\*William Freeman

CS303

9/26/2013

Project 1 - Bank Teller Simulation

Description: Create a program that simulates a bank teller serving customers

which are stored in a stack. The teller will serve the last customer that

arrives first. Customers are allowed to withdrawal or deposit when they are

being served. The user specifies how many customers they would like served

before the simulation ends.

--BankTellerSim.cpp is the implementation file for the BankTellerSim class--

Class Desciption: BankTellerSim controls the main portion of the simulation.\*/

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <fstream>

#include "BankTellerSim.h"

using std::cout;

using std::endl;

using std::rand;

using std::ifstream;

/\*Generates a random double between 0 - 1 and determines if a Customer will

join the wait line.\*/

void BankTellerSim::AddCustomer()

{

double randNum = double(rand()) / RAND\_MAX;

/\*double(rand()) / RAND\_MAX;

taken from Objects, Abstraction, Data Structures and Design using

C++ by Elliot B. Koffman & Paul A.T. Wolfgang.

Pg. 396.

Generates a double number between 0 - 1.\*/

if (randNum <= ARRIVAL\_PROB) //Customer joins the line.

{

//Obtain the required info for a Customer.

string first = SelectFirstName();

string last = SelectLastName();

//0 - MAX\_STARTING\_BAL - 1

double startingBal = rand() % MAX\_STARTING\_BAL;

Customer cust(first, last, startingBal);

m\_waitLine.push(cust);

}

}

/\*Outputs the total number of Customers that are left wiating in line along

with the name of the next customer to be served.\*/

void BankTellerSim::OutputWaitLine() const

{

if (m\_waitLine.empty())

{

//No Customers in line.

cout << "---------------------------------\n"

<< "Currently the wait line is empty.\n";

}

else

{

cout << "---------------------------------\n"

<< "Number of customers in line: " << m\_waitLine.size() << endl

<< "Next customer up is: " << m\_waitLine.top().GetName() << endl;

}

}

/\*Fills the vectors, m\_firstNameList & m\_lastNameList with names from the txt

files that contain these names.\*/

void BankTellerSim::LoadNames()

{

ifstream inFile;

//Load in first names.

inFile.open("FirstNames.txt");

string name;

cout << name << endl;

while (inFile >> name)

{

m\_firstNameList.push\_back(name);

}

inFile.close();

//Load in last names.

inFile.open("LastNames.txt");

while (inFile >> name)

{

m\_lastNameList.push\_back(name);

}

inFile.close();

}

/\*Generates a random int between 0 - m\_firstNameList.size() - 1.

Returns a random name from m\_firstNameList.\*/

string BankTellerSim::SelectFirstName() const

{

int randIndex = rand() % m\_firstNameList.size();

return m\_firstNameList[randIndex];

}

/\*Generates a random int between 0 - m\_lastNameList.size() - 1.

Returns a random name from m\_lastNameList.\*/

string BankTellerSim::SelectLastName() const

{

int randIndex = rand() % m\_lastNameList.size();

return m\_lastNameList[randIndex];

}

/\*Serves the next Customer in line. Determines the amount of Transactions that

Customer will make. Takes into account the amout of time that Customer takes

per Transaction and calls AddCustomer() during each time tick.\*/

void BankTellerSim::ServeCustomer()

{

int numTransactions;

Customer cust;

cust = m\_waitLine.top();

m\_waitLine.pop();

//1 - MAX\_TRANSACTIONS

numTransactions = rand() % MAX\_TRANSACTIONS + 1;

//Customer might make multiple Transactions.

for (int trans = 0; trans < numTransactions; trans++)

{

double startBal = cust.GetCurBal(); //Obtain the initial balance.

int serveTime = cust.GetServeTime();

cust.MakeTransaction();

double endBal = cust.GetCurBal(); //Obtain balance after Transaction.

ReportTransaction(cust, startBal, endBal);

//Time the Customer takes per Transaction.

for (int time = 0; time < serveTime; time++)

{

//Customer has a chance to show up during Transaction.

AddCustomer();

}

}

m\_custServed++;

}

/\*Creates a report of each and every Transaction made in a .csv file.

Includes details about the Transaction in the format:

LAST,FIRST,START\_BAL,TRANS\_TYPE,TRANS\_AMT,CUR\_BAL.\*/

void BankTellerSim::ReportTransaction(Customer& cust, double startBal,

double endBal)

{

if (startBal > endBal)

{

//Transaction was a withdrawal.

m\_report << cust.GetName() << ",$" << startBal << ",Withdrawal,$"

<< startBal - endBal << ",$" << cust.GetCurBal() << endl;

}

else

{

//Transaction is a deposit.

m\_report << cust.GetName() << ",$" << startBal << ",Deposit,$"

<< endBal - startBal << ",$" << cust.GetCurBal() << endl;

}

}

/\*Runs the Bank Teller Simulation until the specified number of customers

have been served.

@param num - the number of Customers to be served before the simulation

ends.\*/

void BankTellerSim::RunSim(int num)

{

srand(time(NULL));

do

{

if (m\_waitLine.empty())

{

//Teller is waiting for Customers, cannot serve them.

AddCustomer();

}

else

{

ServeCustomer();

if (m\_custServed % 5 == 0)

{

//Output every 5 customers served.

OutputWaitLine();

}

}

//Stop once the requested amout of Customers have been served.

} while(m\_custServed < num);

cout << endl << "A report of all transactions have been sent to"

<< "\"Transaction Report.txt\"\n";

m\_report.close();

}

**Customer.h**

/\*William Freeman

CS303

9/26/2013

Project 1 - Bank Teller Simulation

Description: Create a program that simulates a bank teller serving customers

which are stored in a stack. The teller will serve the last customer that

arrives first. Customers are allowed to withdrawal or deposit when they are

being served. The user specifies how many customers they would like served

before the simulation ends.

--Customer.h is the header file for the Customer class--

Class Desciption: Customer contains the information about the customer

that is waiting in line. Includes the customer's last name, first name,

balance, and transaction.\*/

#ifndef \_CUSTOMER\_H

#define \_CUSTOMER\_H

#include <string>

#include <cstdlib>

#include "Transaction.h"

using std::string;

class Customer

{

private:

//Member variables

string m\_lastName;

string m\_firstName;

double m\_curBalance;

int m\_serveTime;

Transaction m\_transaction;

//Static const variables

static const int MAX\_SERVE\_TIME = 5;

//Max time a Customer will take per Transaction.

public:

//Constructors

/\*Default constructor.

Sets member variables to default values for an instance of class

Customer.\*/

Customer():

m\_lastName("Last"), m\_firstName("First"), m\_curBalance(0.0)

{

//1 - MAX\_SERVE\_TIME

m\_serveTime = rand() % MAX\_SERVE\_TIME + 1;

}

/\*Sets the member variables to the values passed as parameters,

m\_curBalance is set as a default value.

@param last - last name of Customer.

@param first - first name of Customer.\*/

Customer(string first, string last):

m\_lastName(last), m\_firstName(first), m\_curBalance(0.0)

{

//1 - MAX\_SERVE\_TIME

m\_serveTime = rand() % MAX\_SERVE\_TIME + 1;

}

/\*Sets the member variables to the values passed as parameters.

@param last - last name of Customer.

@param first - first name of Customer.

@param bal - the current balance of Customer's account.\*/

Customer(string first, string last, double bal):

m\_lastName(last), m\_firstName(first), m\_curBalance(bal)

{

//1 - MAX\_SERVE\_TIME

m\_serveTime = rand() % MAX\_SERVE\_TIME + 1;

}

//Member functions

/\*Returns the value of m\_curBalance for an instance of class

Customer.\*/

double GetCurBal() const

{

return m\_curBalance;

}

/\*Returns a concatenation of the Customer's name, in a last, first

format.\*/

string GetName() const

{

return m\_lastName + ", " + m\_firstName;

}

/\*Returns the value of m\_serveTime for an instance of class Customer.\*/

int GetServeTime() const

{

return m\_serveTime;

}

/\*Runs a Transaction for the Customer & updates the Customer's

current balance accordingly based upon the Transaction type.\*/

void MakeTransaction();

//Operator overloads

/\*Assignment operator overloading.

Ensures that the private member variables of class Customer gets assigned

properly.

Code obtained from:

Lecture notes from CS201 SM-13 class.\*/

Customer& operator=(const Customer& cust);

}; //class Customer

#endif //\_CUSTOMER\_H

**Customer.cpp**

/\*William Freeman

CS303

9/26/2013

Project 1 - Bank Teller Simulation

Description: Create a program that simulates a bank teller serving customers

which are stored in a stack. The teller will serve the last customer that

arrives first. Customers are allowed to withdrawal or deposit when they are

being served. The user specifies how many customers they would like served

before the simulation ends.

--Customer.cpp is the implementation file for the Customer class--

Class Desciption: Customer contains the information about the customer

that is waiting in line. Includes the customer's last name, first name,

balance, and transaction.\*/

#include "Customer.h"

/\*Runs a Transaction for the Customer & updates the Customer's

current balance accordingly based upon the Transaction type.\*/

void Customer::MakeTransaction()

{

double transAmount;

do

{

transAmount = m\_transaction.RunTransaction();

} while ((m\_curBalance + transAmount) < 0); //Prevent a negative balance.

m\_curBalance += transAmount;

}

/\*Assignment operator overloading.

Ensures that the private member variables of class Transaction gets assigned

properly.

Code obtained from:

Lecture notes from CS201 SM-13 class.\*/

Customer& Customer::operator=(const Customer& cust)

{

if (this == &cust)

{

//Check for self-assignment.

return \*this;

}

m\_lastName = cust.m\_lastName;

m\_firstName = cust.m\_firstName;

m\_curBalance = cust.m\_curBalance;

m\_serveTime = cust.m\_serveTime;

m\_transaction = cust.m\_transaction;

return \*this;

}

**Transaction.h**

/\*William Freeman

CS303

9/26/2013

Project 1 - Bank Teller Simulation

Description: Create a program that simulates a bank teller serving customers

which are stored in a stack. The teller will serve the last customer that

arrives first. Customers are allowed to withdrawal or deposit when they are

being served. The user specifies how many customers they would like served

before the simulation ends.

--Transaction.h is the header file for the Transaction class--

Class Desciption: Transaction contains the information pertaining to a

customer's transaction.\*/

#ifndef \_TRANSACTION\_H

#define \_TRANSACTION\_H

class Transaction

{

private:

//Member variables

bool m\_withdrawal;

double m\_amount;

//Static & const member variables

static const int MAX\_WITHDRAWAL = 500; //Max amount allowed to withdrawal.

static const int MAX\_DEPOSIT = 5000; //Max amount allowed to deposit.

const double WITHDRAWAL\_PROB; //Probability for a withdrawal.

//Member functions

/\*Determines if the Transaction is to be a withdrawl or a deposit based

upon the WITHDRAWL\_PROB.\*/

void DetermineTransType();

public:

//Constructors

/\*Default constructor.

Initalizes member variables to default values for an instance of

class Transaction.\*/

Transaction():

m\_withdrawal(false), m\_amount(0.0), WITHDRAWAL\_PROB(0.25) {}

//Member functions

/\*Calls DetermineTransType() to determine the Transaction type.

Returns the amount to add or subtract from the Customer's bank account.\*/

double RunTransaction();

//Operator overloads

/\*Assignment operator overloading.

Ensures that the private member variables of class Transaction gets

assigned properly.

Code obtained from:

Lecture notes from CS201 SM-13 class.\*/

Transaction& operator=(const Transaction& trans);

}; //class Transaction

#endif //\_TRANSACTION\_H

**Transaction.cpp**

/\*William Freeman

CS303

9/26/2013

Project 1 - Bank Teller Simulation

Description: Create a program that simulates a bank teller serving customers

which are stored in a stack. The teller will serve the last customer that

arrives first. Customers are allowed to withdrawal or deposit when they are

being served. The user specifies how many customers they would like served

before the simulation ends.

--Transaction.cpp is the implementation file for the Transaction class--

Class Desciption: Transaction contains the information pertaining to a

customer's transaction.\*/

#include <cstdlib>

#include "Transaction.h"

/\*Determines if the Transaction is to be a withdrawl or a deposit based

upon the WITHDRAWL\_PROB.\*/

void Transaction::DetermineTransType()

{

double randNum = double(rand()) / RAND\_MAX;

/\*double(rand()) / RAND\_MAX;

taken from Objects, Abstraction, Data Structures and Design using

C++ by Elliot B. Koffman & Paul A.T. Wolfgang.

Pg. 396.

Generates a double number between 0 - 1.\*/

if (randNum <= WITHDRAWAL\_PROB)

{

m\_withdrawal = true;

}

else

{

m\_withdrawal = false; //Transaction is a deposit.

}

}

/\*Calls DetermineTransType() to determine the Transaction type.

Returns the amount to add or subtract from the Customer's bank account.\*/

double Transaction::RunTransaction()

{

DetermineTransType();

if (m\_withdrawal)

{

//1 - MAX\_WITHDRAWAL

m\_amount = 0 - double(rand() % MAX\_WITHDRAWAL + 1);

if (m\_amount != -MAX\_WITHDRAWAL)

{

m\_amount -= double(rand()) / RAND\_MAX; //Add change.

/\*double(rand()) / RAND\_MAX;

taken from Objects, Abstraction, Data Structures and Design using

C++ by Elliot B. Koffman & Paul A.T. Wolfgang.

Pg. 396.

Generates a double number between 0 - 1.\*/

}

return m\_amount;

}

else

{

//1 - MAX\_DEPOSIT

m\_amount = double(rand() % MAX\_DEPOSIT + 1);

if (m\_amount != MAX\_DEPOSIT)

{

m\_amount += double(rand()) / RAND\_MAX; //Add change.

/\*double(rand()) / RAND\_MAX;

taken from Objects, Abstraction, Data Structures and Design using

C++ by Elliot B. Koffman & Paul A.T. Wolfgang.

Pg. 396.

Generates a double number between 0 - 1.\*/

}

return m\_amount;

}

}

/\*Assignment operator overloading.

Ensures that the private member variables of class Transaction gets assigned

properly.

Code obtained from:

Lecture notes from CS201 SM-13 class.\*/

Transaction& Transaction::operator=(const Transaction& trans)

{

if (this == &trans)

{

//Check for self assignment.

return \*this;

}

m\_withdrawal = trans.m\_withdrawal;

m\_amount = trans.m\_amount;

return \*this;

}

**ADDITIONAL FILES:**

**FirstNames.txt**

James

Luke

Jacob

William

Brooke

Monty

Jeanne

Jesse

Travis

Byron

Tony

Chris

Jerry

Ana

Megan

Nichole

Kevin

Tyler

Sadie

Matt

Julio

Shawn

Sean

Roddy

Vince

Lee

Scott

Victor

Tate

Robert

Jake

Thomas

Casey

Steve

Julia

Tim

Frank

**LastNames.txt**

Williams

Brewington

Dobson

Brady

Freeman

Harbin

Kelly

Stevens

Smith

Brite

Collinsworth

Davis

Boyce

Matthews

Jones

Johnson

Thompson

**Transaction Report.csv (Sample run)**

LAST,FIRST,START\_BAL,TRANS\_TYPE,TRANS\_AMT,CUR\_BAL

Freeman, Victor,$8799,Withdrawal,$427.093,$8371.91

Boyce, Ana,$8671,Withdrawal,$393.567,$8277.43

Boyce, Ana,$8277.43,Deposit,$344.969,$8622.4

Boyce, Ana,$8622.4,Deposit,$2217.48,$10839.9

Boyce, Monty,$2766,Deposit,$1015.5,$3781.5

Boyce, Monty,$3781.5,Deposit,$3984.14,$7765.64

Brewington, Monty,$6518,Withdrawal,$7.36787,$6510.63

Jones, Sean,$7423,Deposit,$1722.73,$9145.73

Jones, Sean,$9145.73,Deposit,$3957.04,$13102.8

Jones, Sean,$13102.8,Withdrawal,$199.356,$12903.4

Williams, Jacob,$4485,Withdrawal,$498.133,$3986.87

Matthews, Robert,$8206,Withdrawal,$73.8009,$8132.2

Matthews, Robert,$8132.2,Deposit,$2535.78,$10668

Matthews, Robert,$10668,Withdrawal,$495.807,$10172.2

Stevens, Brooke,$6298,Deposit,$3521.69,$9819.69

Stevens, Brooke,$9819.69,Deposit,$473.132,$10292.8

Freeman, Jesse,$631,Deposit,$275.521,$906.521

Kelly, Monty,$706,Withdrawal,$79.1756,$626.824

Kelly, Monty,$626.824,Deposit,$3005.84,$3632.67

Boyce, Monty,$3789,Deposit,$2529,$6318

Boyce, Monty,$6318,Deposit,$1741.72,$8059.72

Boyce, Monty,$8059.72,Deposit,$2315.03,$10374.7

Jones, Nichole,$8538,Deposit,$2133.64,$10671.6

Kelly, Megan,$4285,Deposit,$2539.42,$6824.42

Kelly, Megan,$6824.42,Withdrawal,$309.673,$6514.75

Kelly, Megan,$6514.75,Deposit,$2403.89,$8918.64

Harbin, Jacob,$4331,Deposit,$537.026,$4868.03

Harbin, Jacob,$4868.03,Deposit,$3223.57,$8091.59

Boyce, Steve,$9699,Deposit,$2198.35,$11897.4

Dobson, Frank,$5647,Deposit,$74.5041,$5721.5

Jones, Monty,$5463,Deposit,$596.544,$6059.54

Jones, Monty,$6059.54,Deposit,$2313.15,$8372.7

Thompson, Julia,$5220,Deposit,$911.053,$6131.05

Thompson, Julia,$6131.05,Withdrawal,$228.003,$5903.05

Thompson, Julia,$5903.05,Deposit,$2543.37,$8446.42

Boyce, Victor,$6924,Withdrawal,$482.575,$6441.42

Boyce, Victor,$6441.42,Deposit,$3167.28,$9608.7

Boyce, Victor,$9608.7,Withdrawal,$315.928,$9292.77

Davis, Tim,$2672,Deposit,$4786.67,$7458.67

Davis, Tim,$7458.67,Withdrawal,$133.864,$7324.81

Davis, Brooke,$9727,Deposit,$3553.98,$13281

Davis, Brooke,$13281,Deposit,$1574.45,$14855.4

Williams, Tim,$2274,Withdrawal,$169.479,$2104.52

Brewington, Sadie,$5249,Withdrawal,$280.814,$4968.19

Smith, Tate,$9104,Withdrawal,$215.411,$8888.59

Smith, Tate,$8888.59,Withdrawal,$338.358,$8550.23

Smith, Tate,$8550.23,Deposit,$4705.84,$13256.1

Brite, Frank,$1012,Deposit,$4815.07,$5827.07

Brite, Frank,$5827.07,Deposit,$686.523,$6513.6

Brite, Frank,$6513.6,Deposit,$333.785,$6847.38

**CITATION:**

*Objects, Abstraction, Data Structures and Design Using C++*, Elliot B. Koffman & Paul A.T. Wolfgang. Pg.369, Listing 6.8