Author: Michael LaPan
Assignment: Program 9

Table of contents:

amazonTrans.h

```
#pragma once
#include <string>
#include <iostream>
#include <iomanip>
#include "person.h"
using namespace std;

/*
*this is a class to hold information about an
*amazon transction. it extends person so it
*also holds the info for the person that made
*this transction
*/
class amazonTrans : public person
{

private:
	int custID;
	double transactionAmount;
	string date;
public:
	amazonTrans();
	void setCustomerID(int toSet);
	int getCustomerID();
	void setTransactionAmount(double toSet);
	double getTransactionAmount();
	void setDate(string toSet);
	//overloading of <
	bool operator< (const amazonTrans& toTest);
	//overloading of ==
	bool operator== (const amazonTrans& isEql);
	string getDate();

	//overloading for the << so i can cout my obj
	friend ostream &operator<<(ostream &output, amazonTrans &D)
	{
		output.clear();
		output << D.fName << " " << D.lName << "(Customer " << D.custID << ")" << "\n"
			<< D.date << "\n"
			<< "$" << setprecision(2) << fixed << D.transactionAmount << "(Tax $" <<
D.transactionAmount*.06 << ")";

		return output;
	}
};
```

binTreeType.h

```
// Specification file for the BinTreeType class
// PRECONDITION for use of this class:
//    Data type defining tree node "info" must have operators
//   '<', 'cout', and ==', or they must be overloaded


//Michael changed stuff here..
//added a few features for this .
//for searching.
#ifndef BINARYTREE_H
#define BINARYTREE_H

#include "person.h"
#include "amazonTrans.h"
#include <iostream>
using namespace std;

template <class ItemType>
class BinTreeType
{
  private:
   struct TreeNode
   {
      ItemType info;
      TreeNode *left;
      TreeNode *right;
   };

   TreeNode *root;

   // Overloaded functions for recursive actions
   void insert(TreeNode *&, TreeNode *&);
   void deleteIt(ItemType, TreeNode *&);
   void makeDeletion(TreeNode *&);
   void destroySubTree(TreeNode *);
   void getSucccessor( TreeNode* aNode, ItemType& data);
   void copyTree(TreeNode*& copy, const TreeNode* origTree);

   // Overloaded traversal functions for recursive actions
   void displayInOrder(TreeNode *);
   void displayPreOrder(TreeNode *);
   void displayPostOrder(TreeNode *);

   // Recursive functions for various utility operations
   int countNodes(TreeNode* tree);
   int getDepth(TreeNode* tree);
            bool found = false;
public:
   BinTreeType();                       // Constructor
   BinTreeType(BinTreeType& origTree);      // Copy constructor
   void operator= (BinTreeType& origTree);   // Overloaded assignment operator
   ~BinTreeType();                      // Destructor

   // Tree data insertion, deletion, and searching
   void insertNode(ItemType);
   bool searchNode(ItemType);
   void deleteNode(ItemType);

   // Tree traversal
   void displayInOrder();
   void displayPreOrder();
   void displayPostOrder();

   // Utilities for tree operations
   int numberOfNodes();               // Count nodes in tree
   int treeDepth();
```

```
//added the functions below here
        person get(ItemType);
        void findme(ItemType);
        void findme(TreeNode *nodePtr, ItemType);



};

//main calling function for the recursive find me function
//if the recursive function does not find it, found will stay false
//once the functions are done. if found is false is will give error
template <class ItemType>
void BinTreeType<ItemType>::findme(ItemType item)
{
        findme(root, item);
        if (found == false)
        {
                cout << "could not find that dat/Cust ID, try again. please check and try again" << endl;
        }
}

// Recursive function fo searching for a cust id and date

template <class ItemType>
void BinTreeType<ItemType>::findme(TreeNode *nodePtr, ItemType item)
{
        //trans the user is looking for
        amazonTrans temp = item;
        if (nodePtr != NULL)
        {
                //temp trans for this current node
                amazonTrans temp2 = nodePtr->info;
                //compare
                if ( temp.getDate() == temp2.getDate()
                        && temp.getCustomerID() == temp2.getCustomerID())
                {
                        //if found cout the trans and set found to true
                        cout << temp2 << endl;
                        found = true;
                }
                else
                {
                        //else recurse
                        findme(nodePtr->left, temp);
                        findme(nodePtr->right, temp);
                }
        }

}

//************************************************************
//************************************************************
// Implementation file for the BinTreeType class
//************************************************************
//************************************************************

// Constructor
template <class ItemType>
BinTreeType<ItemType>::BinTreeType()
{
  root = NULL;
}

template <class ItemType>

//will return the person info.
//this will scan through the tree looking
//for a person by their id and then return them
```

```
person BinTreeType<ItemType>::get(ItemType item)
{
	TreeNode *nodePtr = root;
	person temp;
	while (nodePtr != NULL)
	{
		if (nodePtr->info == item)
		{
			//temp = nodePtr->info;
			return nodePtr->info;
		}

		else if (item < nodePtr->info)
			nodePtr = nodePtr->left;
		else
			nodePtr = nodePtr->right;
	}
}


//***************************************************************
// Copy constructor - Utilizes recursive utility function
// copyTree to actually replicate original tree
//***************************************************************

template <class ItemType>
BinTreeType<ItemType>::BinTreeType(BinTreeType<ItemType>& origTree)
{
  copyTree(root, origTree.root);
}

//***************************************************************
// Overloaded assignment operator - Utilizes recursive utility function
// copyTree to actually replicate original tree
//***************************************************************

template <class ItemType>
void BinTreeType<ItemType>::operator= (BinTreeType<ItemType>& origTree)
{
  destroySubTree(root);        // Eliminate any existing nodes in target
  copyTree(root, origTree.root); // Copy source to target as part of assignment
}

//***************************************************************
// Destructor
//***************************************************************

template <class ItemType>
BinTreeType<ItemType>::~BinTreeType()
{
  destroySubTree(root);
}

//***************************************************************
// insert accepts a TreeNode pointer and a pointer to a node. *
// The function inserts the node into the tree pointed to by  *
// the TreeNode pointer. This function is called recursively. *
//***************************************************************

template <class ItemType>
void BinTreeType<ItemType>::insert(TreeNode *&nodePtr, TreeNode *&newNode)
{
   if (nodePtr == NULL)
      nodePtr = newNode;                  // Insert the node.
   else if (newNode->info < nodePtr->info)
      insert(nodePtr->left, newNode);       // Search the left branch
   else
      insert(nodePtr->right, newNode);      // Search the right branch
}
```

```
//********************************************************
// insertNode creates a new node to hold num as its value, *
// and passes it to the insert function.                   *
//********************************************************

template <class ItemType>
void BinTreeType<ItemType>::insertNode(ItemType num)
{
   TreeNode *newNode;       // Pointer to a new node.

   // Create a new node and store num in it.
   newNode = new TreeNode;
   newNode->info = num;
   newNode->left = newNode->right = NULL;
   // Insert the node.
   insert(root, newNode);
}

//*****************************************************
// destroySubTree is called by the destructor. It   *
// deletes all nodes in the tree.                   *
//*****************************************************

template <class ItemType>
void BinTreeType<ItemType>::destroySubTree(TreeNode *nodePtr)
{
   if (nodePtr != NULL)
   {
      if (nodePtr->left != NULL)
         destroySubTree(nodePtr->left);
      if (nodePtr->right != NULL)
         destroySubTree(nodePtr->right);
      delete nodePtr;
   }
}

//*****************************************************
// searchNode determines if a value is present in   *
// the tree. If so, the function returns true.      *
// Otherwise, it returns false.                     *
//*****************************************************

template <class ItemType>
bool BinTreeType<ItemType>::searchNode(ItemType item)
{
   TreeNode *nodePtr = root;

   while (nodePtr != NULL)
   {
      if (nodePtr->info == item)
         return true;
      else if (item < nodePtr->info)
         nodePtr = nodePtr->left;
      else
         nodePtr = nodePtr->right;
   }
   return false;
}

//*********************************************
// Function deleteNode triggers the chain of   *
// recursive  calls to search for and delete   *
// target node.                                *
//*********************************************

template <class ItemType>
void BinTreeType<ItemType>::deleteNode(ItemType item)
{
```

```
   deleteIt(item, root);
}


//***********************************************
// Function deleteIt recursively searches for   *
// the item to delete and calls function        *
// makeDeletion to perform the actual deletion. *
//***********************************************

template <class ItemType>
void BinTreeType<ItemType>::deleteIt(ItemType item, TreeNode *&nodePtr)
{
   if (item < nodePtr->info)
      deleteIt(item, nodePtr->left);
   else if (item > nodePtr->info)
      deleteIt(item, nodePtr->right);
   else
      makeDeletion(nodePtr);
}

//*********************************************************
// makeDeletion takes a reference to a pointer to the node  *
// that is to be deleted. The node is removed and the       *
// branches of the tree below the node are reattached.      *
//*********************************************************

template <class ItemType>
void BinTreeType<ItemType>::makeDeletion(TreeNode *&nodePtr)
{
   TreeNode *tempNodePtr;   // Temporary pointer, used for deletion
   ItemType data;

   if (nodePtr->right == NULL)      // If no right child exists
   {
      tempNodePtr = nodePtr;
      nodePtr = nodePtr->left;     // Then reattach the left child
      delete tempNodePtr;
   }
   else if (nodePtr->left == NULL)   // If no left child exists
   {
      tempNodePtr = nodePtr;
      nodePtr = nodePtr->right;    // Then reattach the right child
      delete tempNodePtr;
   }
   else                         // If the node has two children
   {
      // Get data for immediate successor (largest node in right subtree)
      getSucccessor(nodePtr,data);

      // Move information from successor node to target node
      nodePtr->info = data;
      deleteIt(data, nodePtr->right);    // And delete successor node
   }
}

//*************************************************************
// This function scans for the succeeding node in order within  *
// a binary tree.  It moves the the right child, and then moves  *
// down the chain of left children until NULL is reached.  It    *
// returns the data at the predecessor node by reference.       *
//*************************************************************

template <class ItemType>
void BinTreeType<ItemType>::getSucccessor( TreeNode* aNode, ItemType& data)
{
  aNode = aNode->right;
  while (aNode->left != NULL)
    aNode = aNode->left;
```

```
  data = aNode->info;

}

//***************************************************************
// The displayInOrder member function displays the values       *
// in the subtree pointed to by nodePtr, via inorder traversal.  *
//***************************************************************

template <class ItemType>
void BinTreeType<ItemType>::displayInOrder()
{
  displayInOrder(root);
}

// Recursive function performing traversal
template <class ItemType>
void BinTreeType<ItemType>::displayInOrder(TreeNode *nodePtr)
{
   if (nodePtr != NULL)
   {

      displayInOrder(nodePtr->left);
      cout << nodePtr->info << " ";
      displayInOrder(nodePtr->right);
   }
}

//***************************************************************
// The displayPreOrder member function displays the values      *
// in the subtree pointed to by nodePtr, via preorder traversal. *
//***************************************************************

template <class ItemType>
void BinTreeType<ItemType>::displayPreOrder()
{
  displayPreOrder(root);
}

// Recursive function performing traversal
template <class ItemType>
void BinTreeType<ItemType>::displayPreOrder(TreeNode *nodePtr)
{
   if (nodePtr != NULL)
   {
      cout << nodePtr->info << " ";
      displayPreOrder(nodePtr->left);
      displayPreOrder(nodePtr->right);
   }
}

//***************************************************************
// The displayPostOrder member function displays the values     *
// in the subtree pointed to by nodePtr, via postorder traversal.*
//***************************************************************

template <class ItemType>
void BinTreeType<ItemType>::displayPostOrder()
{
  displayPostOrder(root);
}

// Recursive function performing traversal
template <class ItemType>
void BinTreeType<ItemType>::displayPostOrder(TreeNode *nodePtr)
{
   if (nodePtr != NULL)
   {
      displayPostOrder(nodePtr->left);
```

```
      displayPostOrder(nodePtr->right);
      cout << nodePtr->info << " ";
   }
}

//***************************************************************
// This function recursively traverses the tree and increments   *
// a counter at each node "visit" to count the total number of   *
// data nodes in the tree.                                       *
//***************************************************************

template<class ItemType>
int BinTreeType<ItemType>::numberOfNodes()
{
  return countNodes(root);
}

// Private function performing recursive count
template<class ItemType>
int BinTreeType<ItemType>::countNodes(TreeNode* tree)
{
  if (tree == NULL)
    return 0;
  else
    return countNodes(tree->left)  +
        countNodes(tree->right) + 1;
}

//***************************************************************
// This function replicates a tree as part of the copy constructor *
// and overloaded assignment operations.                         *
//***************************************************************

template<class ItemType>
void BinTreeType<ItemType>::copyTree(TreeNode*& copy, const TreeNode* origTree)
{
  if (origTree == NULL)   // Handle case of empty tree
    copy = NULL;
  else
  {
    copy = new TreeNode;
    copy->info = origTree->info;
    copyTree(copy->left, origTree->left);
    copyTree(copy->right, origTree->right);
  }
}

//***************************************************************
// Function checking maximum depth below current node
//***************************************************************

// Public function initiating count and returning total to main
// function call
template<class ItemType>
int BinTreeType<ItemType>::treeDepth()
{
   int    depth = getDepth(root) - 1;
   return depth;
}

template<class ItemType>
int BinTreeType<ItemType>::getDepth(TreeNode* tree)
{
   if (tree == NULL)
      return 0;
   else
   {
      // Get depths below current node
      int leftDepth  = getDepth(tree->left);
```

```
      int rightDepth = getDepth(tree->right);

      // Return max depth of subtrees plus one for "this" node
      if ( leftDepth > rightDepth)
         return leftDepth + 1;
      else
         return rightDepth + 1;
   }
}


#endif
```

Person.h

```
#ifndef PERSON_H
#define PERSON_H
#include <string>
#include <iostream>

using namespace std;

/*
*this is a basic person class
*holds info about a person
*/
class person
{
protected:
	string fName;
	string lName;
	int transID;
public:
	person();
	void setFName(string toSet);
	string getFName();
	void setLName(string toSet);
	string getLName();
	void setTransID(int toSet);
	//overloading of <
	bool operator< (const person& toTest);
	//overloading of ==
	bool operator== (const person& isEql);
	int getTransID();

	//overloading for the << to cout the obj
	friend ostream &operator<<(ostream &output, person &D)
	{
		output.clear();
		output << "    " << D.fName << "     " << D.lName << "  " << D.transID;
		return output;
	}
};

#endif
```

Wrap.h

```
#ifndef WRAP_H
#define WRAP_H


#include <fstream>
#include <iostream>
#include <string>
#include "binTreeType.h"
#include "amazonTrans.h"
#include "person.h"
using namespace std;

/*
*this is a wrapper class to push all the nasty
*stuff for binary search tree. this way the user
*just has to call the find function to get the info
*they want
*/
class wrap
{
private:
	//two BST to hold all data
	BinTreeType<amazonTrans> transTree;
	BinTreeType<person> personTree;

	//one call to make both BST they have to be
	//built in a certin order
	void makeBST();

	//populate the people BST
	void popPepsBST();

	//populate the tranzation BST
	void popTransBST();

public:

	wrap();

	//call this to find something in the transction BST
	void find(int toFind, string findIt);
};

#endif
```

Amazontrans.cpp

```
#include "amazonTrans.h"


amazonTrans::amazonTrans()
{
	custID = 0;
	transactionAmount = 0.00;
}


void amazonTrans::setCustomerID(int toSet)
{
```

```
	custID = toSet;
}


int amazonTrans::getCustomerID()
{
	return custID;
}


void amazonTrans::setTransactionAmount(double toSet)
{
	transactionAmount = toSet;
}


double amazonTrans::getTransactionAmount()
{
	return transactionAmount;
}


void amazonTrans::setDate(string toSet)
{
	date = toSet;
}


string amazonTrans::getDate()
{
	return date;
}


//overloading of ==
//if obj passed in equals this return true

bool amazonTrans::operator== (const amazonTrans& isEql)
{
	//everything must equal this
	cout << isEql.custID << " " << custID << " " << isEql.date << " " << date << " " << endl;
	if (isEql.custID == custID)
	{
			return true;


	}
	return false;
}

//overloading of the < operator
//if criticality of obj getting passed in is
//>then this criticality then return true

bool amazonTrans::operator< (const amazonTrans& toTest)
{
	if (toTest.custID == custID)
	{
		if (toTest.date > date)
		{
			return true;
		}
		else
		{
			return false;
		}
	}
	else if (toTest.custID > custID)
	{
```

```
            return true;
}
else
{
            return false;
}

}
```

Person.cpp

```
#include "person.h"


person::person()
{
	transID = 0;
}

//tests to see if lesser. lesser people have
//a lesser trans id

bool person::operator< (const person& toTest)
{
	if (toTest.transID < transID)
	{
		return true;
	}
	return false;
}

//comparing person to person
//the same person has the same trans id

bool person::operator== (const person& isEql)
{
	if (isEql.transID == transID)
	{
		return true;
	}
	return false;
}

void person::setFName(string toSet)
{
	fName = toSet;
}


string person::getFName()
{
	return fName;
}


void person::setLName(string toSet)
{
	lName = toSet;
}


string person::getLName()
{
	return lName;
}


void person::setTransID(int toSet)
{
	transID = toSet;
}


int person::getTransID()
{
```

```
        return transID;
}
```

Source.cpp

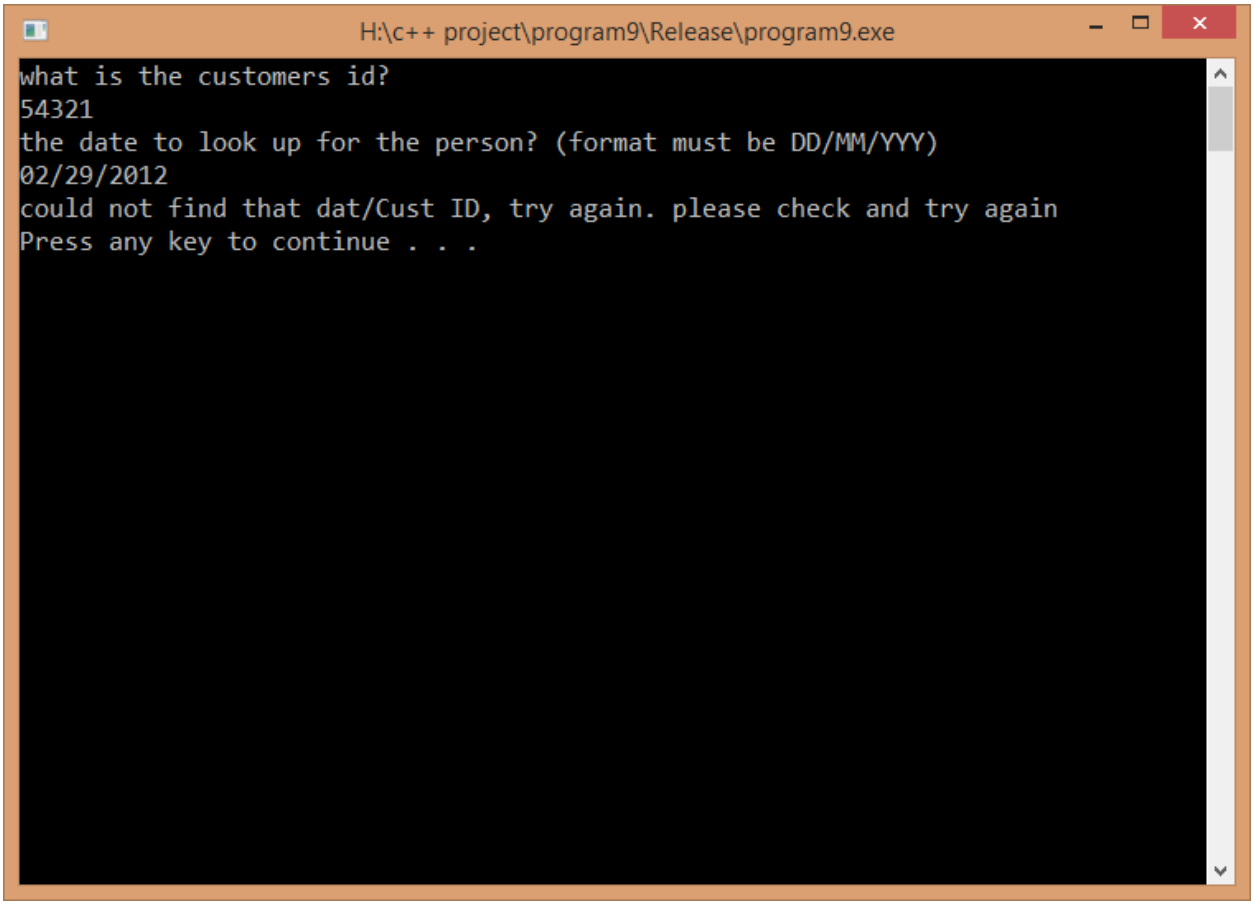
```
//Author: michael lapan
//assignment: program9
/*
*this program reads two files, loads both into BST
*and then allows the user to search for a user on a certin date
*/

#include <fstream>
#include <iostream>
using namespace std;

#include "wrap.h"
int main()
{
	//vars to hold user input.
	int custID = 0;
	string date;

	wrap BST;
	cout << "what is the customers id?" << endl;
	cin >> custID;
	cout << "the date to look up for the person? (format must be DD/MM/YYY)" << endl;
	cin >> date;

	//find this custID on this date
	BST.find(custID, date);
	system("pause");
}
```

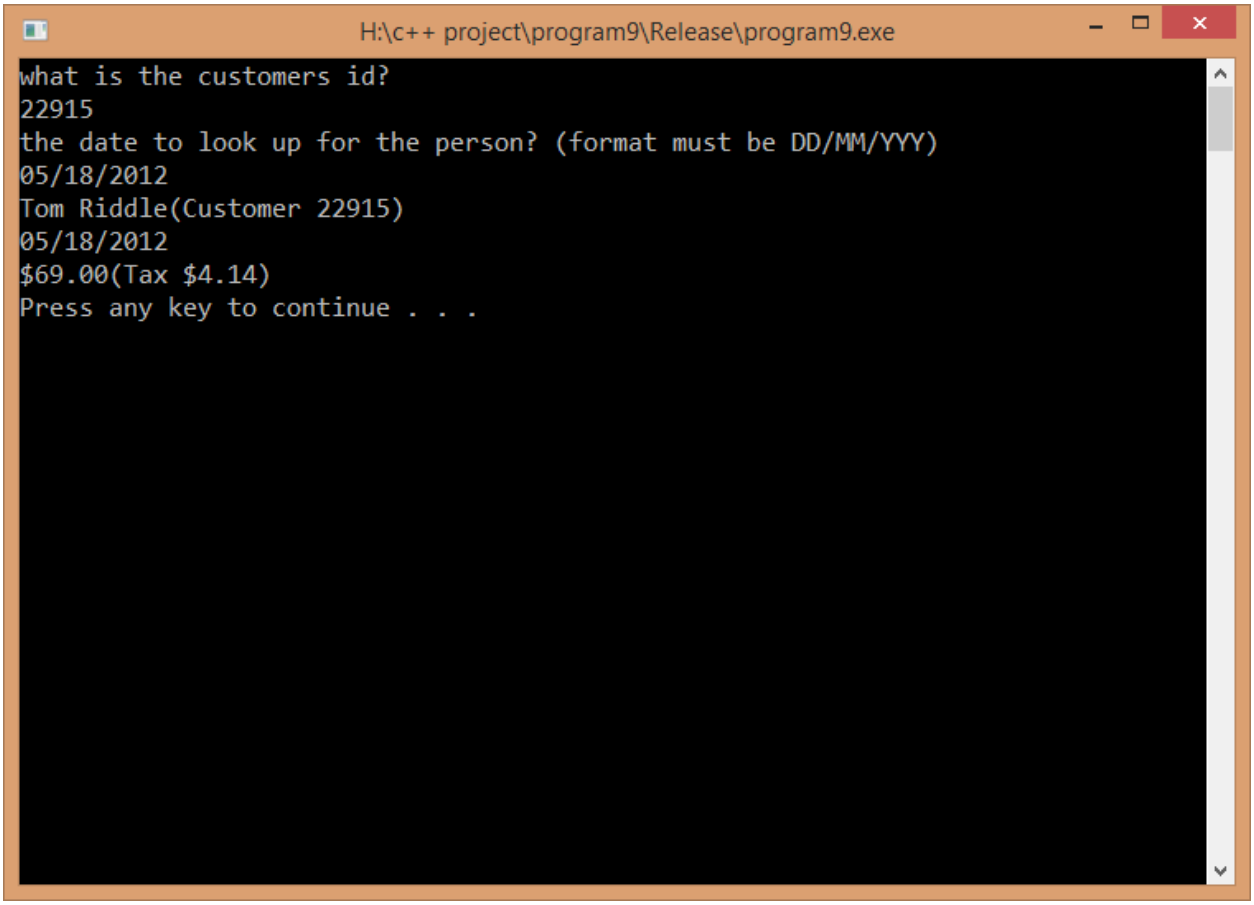
Wrap.cpp

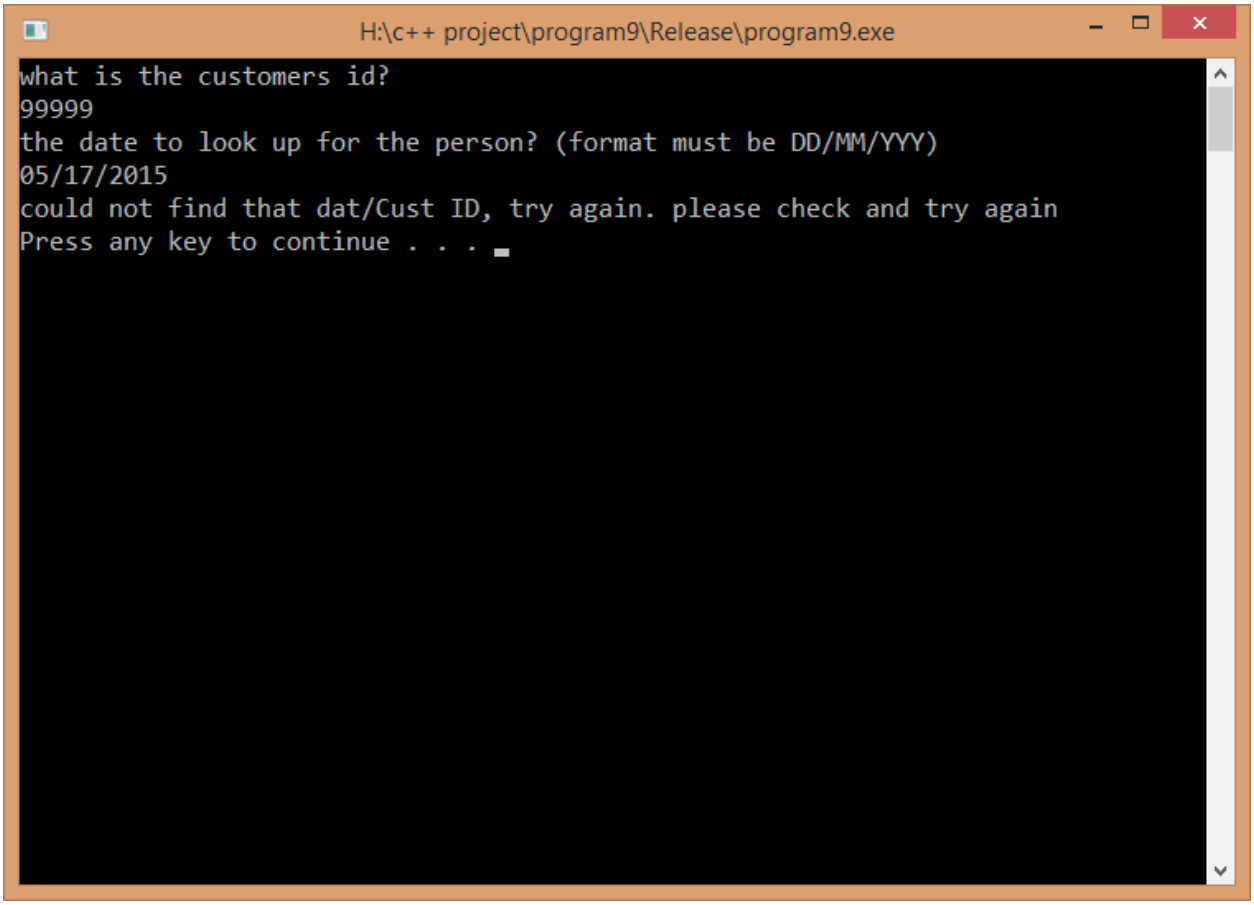
```
#include "wrap.h"
#include <fstream>
#include <iostream>
#include <string>
#include <sstream>
using namespace std;


//on obj creation call makeBST, to make the trees
wrap::wrap()
{
	makeBST();
}

/*
*this will populate the transation BST. this has to be
*called after the people tree is populated. becuase i do
*a look up for each of the peoples names and match, then store
*them in the amatrans obj as it goes into this tree
*/
void wrap::popTransBST()
{
	//my temp vars
	string aWord;
	amazonTrans temp;
	person tempP;
	int i = 0;

	// Open word list file
	ifstream wordFile("transactions.txt");
	if (wordFile.fail())
	{
		cout << "Problem opening document file";
		exit(-1);
	}
	int q = 0;

	// Build list of words in document
	wordFile >> aWord;		// Get first word
	while (!wordFile.eof())
	{
		stringstream ss(aWord);
		string s;

		//split word appart at the comma
		while (getline(ss, s, ','))
		{
			//split the word and then place it into the obj
			if (i == 0)
			{
				temp.setCustomerID(atoi(s.c_str()));
			}
			else if (i == 1)
			{
				temp.setTransactionAmount(atoi(s.c_str()));
			}
			else if (i == 2)
			{

				//convert the day from YY-MM-DD to MM/DD/YYY
				string month, day, year;
				stringstream sss(s);
				string ll;
				int po = 0;
```

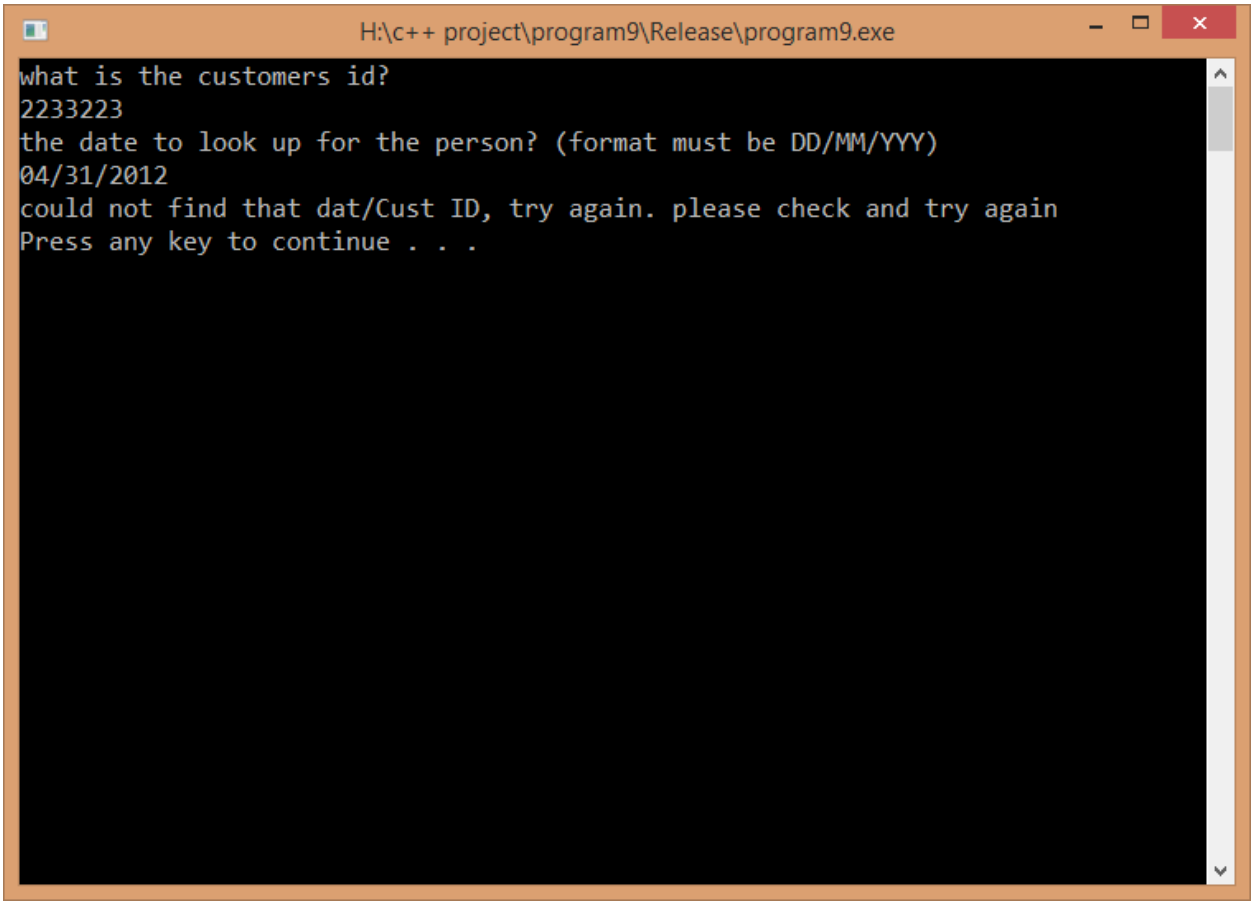
```
                                        //break the string apart at the -
                                        while (getline(sss, ll, '-'))
                                        {
                                                if (po == 0)
                                                {
                                                        year = ll;
                                                }
                                                else if (po == 1)
                                                {
                                                        month = ll;
                                                }
                                                else if (po == 2)
                                                {
                                                        day = ll;
                                                        po = -1;
                                                }
                                                po++;
                                        }

                                        //build the new date string
                                        temp.setDate(month+"/"+day+"/"+year);
                                        i = -1;
                                }

                                i++;
                        }

                        //set the temp persons transid
                        tempP.setTransID(temp.getCustomerID());

                        //check to see if the person is in the tree
                        if (personTree.searchNode(tempP))
                        {
                                //if they are it returns that person
                                //then sets the name of the person in the
                                //temp transction
                                tempP = personTree.get(tempP);
                                temp.setFName(tempP.getFName());
                                temp.setLName(tempP.getLName());

                        }
                        else
                        {
                                //did not find
                                //profound error would go here in
                                //business world
                        }

                        // add the temp transtion to the trans tree

                        transTree.insertNode(temp);

                        //move to next word
                        wordFile >> aWord;
                }

                //after all is done close
                wordFile.close();
}

//this will find a transction in the bst
void wrap::find(int toFind, string findIt)
{
                amazonTrans temp;
                amazonTrans temp2;

                //build temp transction
                temp.setCustomerID(toFind);
```

```
	temp.setTransID(toFind);
	temp.setDate(findIt);
	transTree.findme(temp);
}

//this will call the populate bst's
//they have to be called in this order
void wrap::makeBST()
{
	popPepsBST();
	popTransBST();
}

/*
*this will populate the people BST. read info in
*from the file puts it into people objs, then to tree
*/
void wrap::popPepsBST()
{
	//temp things to hold info
	string aWord;
	person temp;
	int i = 0;
	// Open word list file
	ifstream wordFile("nameid.txt");
	if (wordFile.fail())
	{
		cout << "Problem opening document file";
		exit(-1);
	}

	// Get first word
	wordFile >> aWord;
	while (!wordFile.eof())
	{
		stringstream ss(aWord);
		string s;
		//split word appart at the comma
		while (getline(ss, s, ','))
		{
			//build the new person
			//everything is being split at the comma
			if (i == 0)
			{
				temp.setFName(s);

			}
			else if (i == 1)
			{
				temp.setLName(s);
			}
			else if (i == 2)
			{
				temp.setTransID(atoi(s.c_str()));

				i = -1;
			}
			i++;

		}
		//add the new person to the tree
		personTree.insertNode(temp);

		// Get next word
		wordFile >> aWord;
	}
	//close the file
	wordFile.close();
}
```


H:\c++ project\program9\Release\program9.exe
what is the customers id?
54321
the date to look up for the person? (format must be DD/MM/YYY)
02/29/2012
could not find that dat/Cust ID, try again. please check and try again
Press any key to continue . . .


H:\c++ project\program9\Release\program9.exe
what is the customers id?
22915
the date to look up for the person? (format must be DD/MM/YYY)
05/18/2012
Tom Riddle(Customer 22915)
05/18/2012
$69.00(Tax $4.14)
Press any key to continue . . .


H:\c++ project\program9\Release\program9.exe
what is the customers id?
99999
the date to look up for the person? (format must be DD/MM/YYY)
05/17/2015
could not find that dat/Cust ID, try again. please check and try again
Press any key to continue . . .


H:\c++ project\program9\Release\program9.exe
what is the customers id?
2233223
the date to look up for the person? (format must be DD/MM/YYY)
04/31/2012
could not find that dat/Cust ID, try again. please check and try again
Press any key to continue . . .