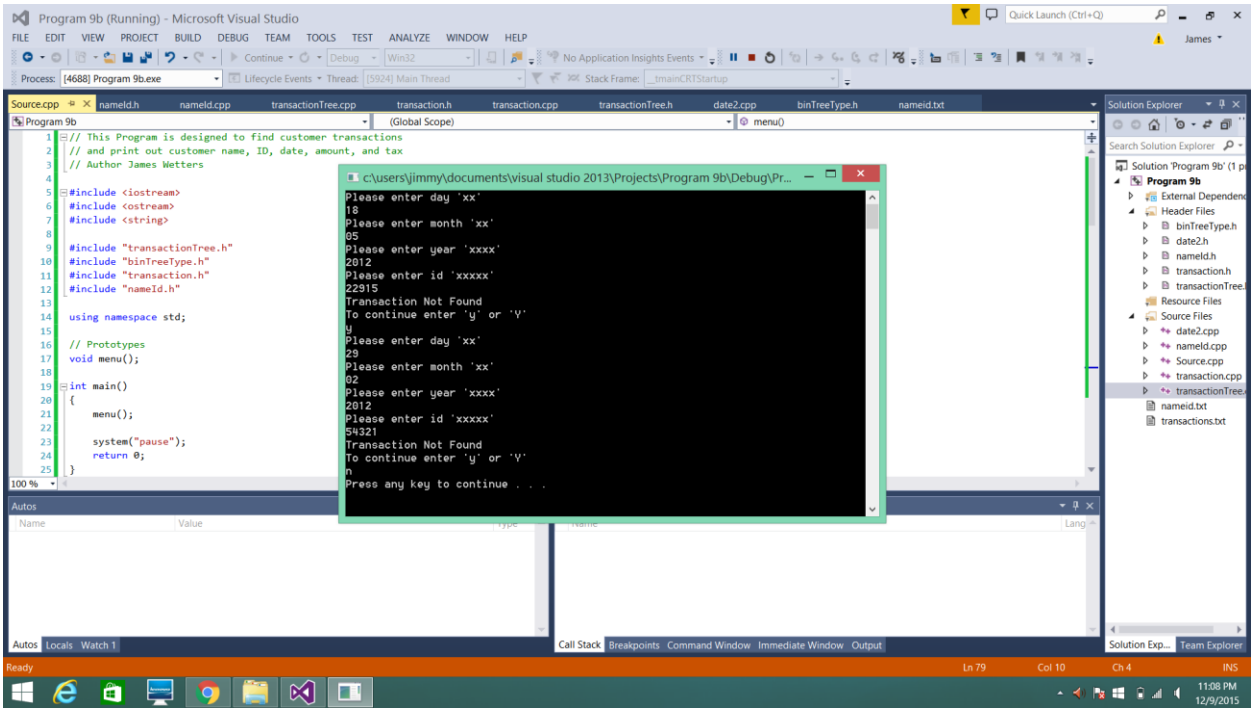
Program 9

James Wetters




```
Please enter day 'xx'
18
Please enter month 'xx'
05
Please enter year 'xxxx'
2012
Please enter id 'xxxxx'
22915
Transaction Not Found
To continue enter 'y' or 'Y'
y
Please enter day 'xx'
29
```

```
Please enter month 'xx'

02

Please enter year 'xxxx'

2012

Please enter id 'xxxxx'

54321

Transaction Not Found

To continue enter 'y' or 'Y'

n

Press any key to continue . . .
```

```
// This Program is designed to find customer transactions
// and print out customer name, ID, date, amount, and tax
// Author James Wetters

#include <iostream>
#include <ostream>
#include <string>

#include "transactionTree.h"
#include "binTreeType.h"
#include "transaction.h"
#include "nameId.h"

using namespace std;

// Prototypes
void menu();

int main()
{
	menu();

	system("pause");
	return 0;
}


void menu()
{
	// Initilize data structures
	TransactionTree theTree;
	NameID names;
```

```
	// Initilize variables
	int day, month, year;
	int id;
	double money;
	string custName;
	char select = 'N';

	// Menu
	do{
		cout << "Please enter day 'xx'" << endl;
		cin >> day;

		cout << "Please enter month 'xx'" << endl;
		cin >> month;

		cout << "Please enter year 'xxxx' " << endl;
		cin >> year;

		cout << "Please enter id 'xxxxx' " << endl;
		cin >> id;

		// Create Temp Date
		Date searchDate(month, day, year);

		// Create Temp Transaction
		Transaction trans;

		// Set date and id to tep Transaction
		trans.setDate(searchDate);
		trans.setCustomerID(id);


		//theTree.search(trans);
		// Search for the temp transaction and copy original to temp if found
		if (theTree.search(trans))
		{
			// Write Report to screen
			// Find name
			cout << names.findName(id) << "(Customer " << id << ")" << endl;
			cout << trans.getDate() << endl;
			cout << trans.getTransactionAmount() << "(Tax: $" << trans.tax()
<<")" << endl;
		}
		else {
			// If ID or Date are not found transaction is not found
			cout << "Transaction Not Found" << endl;
		}

		// User enters to continue or exit
		cout << "To continue enter 'y' or 'Y' " << endl;
		cin >> select;

	} while (select == 'Y' || select == 'y');
}
```

```
// Transaction Header
// Author James Wetters
#ifndef TRANSACTION_H
#define TRANSACTION_H

#include <iostream>
#include <string>

#include "date2.h"

using namespace std;


class Transaction
{
private:
	// Variables
	int customerID;
	double transactionAmount;
	Date date;

public:
	// Gets
	int getCustomerID()
	{
		return customerID;
	}
	double getTransactionAmount()
	{
		return transactionAmount;
	}
	Date getDate()
	{
		return date;
	}

	// Sets
	void setCustomerID(int change)
	{
		customerID = change;
	}
	void setTransactionAmount(double change)
	{
		transactionAmount = change;
	}
	void setDate(Date change)
	{
		date = change;
	}

	// Problem constructor
	Transaction();
	// Problem paramaterized constructor
	Transaction(int tCustId, double tAmount, Date tDate);

	// Problem Overloaded Operators
	bool Transaction::operator< (Transaction& p);
	bool Transaction::operator== (Transaction& p);
```

```
	//Transaction Transaction::operator=(Transaction t);

	double tax();
};
#endif
```

```
// Transaction Source
// Author James Wetters

#include "transaction.h"

// Constructor
Transaction::Transaction()
{}

// Paramaterized Constructor
Transaction::Transaction(int tCustId, double tAmount, Date tDate)
{
	setCustomerID(tCustId);
	setTransactionAmount(tAmount);
	setDate(tDate);
}

//---------------------------------------------------------------------
//	Tax
//
// Returns a double for the amount of tax on the transaction
//---------------------------------------------------------------------
double Transaction::tax()
{
	return (transactionAmount * .06);
}

//---------------------------------------------------------------------
// Overloaded operator < less than
//---------------------------------------------------------------------
bool Transaction::operator< (Transaction& t)
{
	if (customerID < t.customerID)
		return true;
	if (customerID == t.customerID && date < t.date)
		return true;
	return false;
}
//---------------------------------------------------------------------
// Overloaded operator == equal to
//---------------------------------------------------------------------
bool Transaction::operator== (Transaction& t)
{
	if (customerID == t.customerID && date == t.date)
	{
		return true;
	}
	return false;
```

```
}

//Transaction Transaction::operator=(Transaction t)
//{
//     setCustomerID(t.getCustomerID());
//     setTransactionAmount(t.getTransactionAmount());
//     setDate(t.getDate());

//     return *this;
//}
```

```
// Transaction Tree Header
// Author James Wetters

#ifndef TRANSACTIONTREE_H
#define TRANSACTIONTREE_H

#include <iostream>
#include <string>
#include <fstream>
#include "binTreeType.h"
#include "transaction.h"

using namespace std;

class TransactionTree
{
private:
       BinTreeType<Transaction> theTransactionTree;

public:
       // Transaction Tree constructor
       TransactionTree();

       // Search Transaction Tree
       bool search(Transaction &tSearch);
};
#endif
```

```
// Transaction Tree Source
// Author James Wetters

#include "transactionTree.h"
#include "date2.h"

// Constructor
TransactionTree::TransactionTree()
{
       // Variables
       int goodData = 0, convertedIData;
```

```
	double convertedDDData;
	string iData, dData, date;

	ifstream inTransFile("transactions.txt");

	// Test File
	if (inTransFile.fail())
	{
		cout << "Problem opening file";
		system("pause");
		exit(-1);
	}

	// Priming read
	getline(inTransFile, iData, ',');

	while (!inTransFile.eof())
	{
		// Convert string to int
		convertedIData = atoi(iData.c_str());

		getline(inTransFile, dData, ',');

		// Convert string to double
		convertedDData = atof(dData.c_str());

		// Read in date
		getline(inTransFile, date);

		// Create date
		Date D(date);

		// Create Transaction
		Transaction T(convertedIData, convertedDData, D);

		// Insert into Problem List
		theTransactionTree.insertNode(T);

		// Prime the next line
		getline(inTransFile, iData, ',');
	}

	inTransFile.close();
	inTransFile.clear();
}

//--------------------------------------------------------------------
//	Search
//
// Searches the transaction tree for a customer ID
//
// returns a copy of the original transaction and a bool
//--------------------------------------------------------------------
bool TransactionTree::search(Transaction &tSearch)
{
	if (theTransactionTree.searchNode(tSearch))
	{
		//cout << tSearch.getTransactionAmount() << endl;
```

```
                return true;
        }
        return false;
}
```

```
// NameID Header
// Author James Wetters
#ifndef NAMEID_H
#define NAMEID_H

#include <iostream>
#include <fstream>
#include <string>

using namespace std;

// Const
const int MAXCUSTOMERARRAY = 10010;

class NameID
{
private:
        // Variables
        int    ID[MAXCUSTOMERARRAY];
        string name[MAXCUSTOMERARRAY];
        int           goodData;

public:

        // Problem constructor
        NameID();

        // Search names by customer ID
        string findName(int custID);
};
#endif
```

```
// Name ID Source
// Author James Wetters

#include "nameId.h"


NameID::NameID()
{
        int convertedIData;
```

```
	string data, first, last, fullN;

	ifstream inNameIDFile("nameid.txt");

	// Test File
	if (inNameIDFile.fail())
	{
		cout << "Problem opening file";
		system("pause");
		exit(-1);
	}

	// Prime read
	getline(inNameIDFile, data, ',');
	goodData = 0;

	while (!inNameIDFile.eof())
	{
		first = data;

		getline(inNameIDFile, data, ',');

		last = data;

		fullN = first + " " + last;

		name[goodData] = fullN;

		//cout << fullN << "   ";

		getline(inNameIDFile, data);

		convertedIData = atoi(data.c_str());

		//cout << convertedIData << endl;

		ID[goodData] = convertedIData;

		getline(inNameIDFile, data, ',');

		goodData++;
	}

	inNameIDFile.close();
	inNameIDFile.clear();
}


string NameID::findName(int custID)
{
	bool found = false;
	int i = 0;
	string notFound = " ";

	while (found != true && i <= goodData)
	{
		if (ID[i] == custID)
		{
```

```
                found = true;
            }
            else i++;

        }

        if (found == true)
        {
            return name[i];
        }

        return notFound;
}
```

```
// 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

#ifndef BINARYTREE_H
#define BINARYTREE_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 getSuccessor(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);
```

```
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();

};


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

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

//****************************************************************
// 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.                     *
//*************************************************

// Changed item to a reference and returned a copy of item

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

    while (nodePtr != NULL)
    {
        if (nodePtr->info == item)
        {
            item = nodePtr->info;
            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
```

```
// Date.h
// This file defines the specifications for the Date class.  This class
// is a utility for any work with calendar dates.
#ifndef DATE_H
#define DATE_H

#include <iostream>
#include <string>


using namespace std;

class Date
{
private:
	int month;
	int day;
	int year;
```

```
public:
	// Default constructor; initialize to 1/1/1990
	Date();

	//-----------------------------------------------------------------
	// Parameterized constructor
	Date(int m, int d, int y);

	//-----------------------------------------------------------------
	// Parameterized constructor for coded string form mm/dd/yyyy
	Date(string codedDate);

	//-----------------------------------------------------------------
	// Set functions
	void setMonth(int m);
	void setDay(int d);
	void setYear(int y);

	//-----------------------------------------------------------------
	// Get functions
	int getMonth();
	int getDay();
	int getYear();

	//-----------------------------------------------------------------
	// This function returns true if the year is a leap year and false
	// otherwise.
	bool leapYear();

	//-----------------------------------------------------------------
	// This function returns an integer of the number of days in the
	// month.  Leap years are considered.
	int daysInMonth();

	//-----------------------------------------------------------------
	// This function returns the Julian date (the day number of the date
	// in that year).
	int julianDate();

	//-----------------------------------------------------------------
	// This method returns a boolean value defining the validity of the
	// date.
	bool validDate();

	//-----------------------------------------------------------------
	// This function returns a date code for the day of the week.  It
	// counts the number of days since 1/1/1900 which was on a Sunday.
	// Output is: 0=Sun,1=Mon, ..., 6=Sat.
	int weekDay();

	//-----------------------------------------------------------------
	// This function returns (via the parameter list) the 3-character
	// descriptor for the day of the week the date represents
	//void dayCode(char descript[]);

	//-----------------------------------------------------------------
	// This function returns (via the parameter list) the string
	// descriptor for the month the date represents
```

```
        //void monthCode(char descript[]);

        //---------------------------------------------------------------
        // Comparison operation for equality; returns true if dates identical
        bool operator== (Date secondDate);

        //---------------------------------------------------------------
        // Comparison operation for less than; returns true referencing date
        // (1st date) is less than date in parameter
        bool operator< (Date secondDate);

        //---------------------------------------------------------------
        // Overload the insertion operator to enable console output
        friend ostream& operator<< (ostream &strm, Date &theObj);
};

#endif
```

```
// This file includes implementations for date functions associated
// with the Date class

#include "date2.h"

//---------------------------------------------------------------
// Default constructor; initialize to 1/1/1990
Date::Date()
{
        month = 1;
        day = 1;
        year = 1990;
}  // end default constructor

//---------------------------------------------------------------
// Parameterized constructor
Date::Date(int m, int d, int y)
{
        month = m;
        day = d;
        year = y;
}  // end constructor

//---------------------------------------------------------------
// Parameterized constructor for coded string form mm/dd/yyyy
Date::Date(string codedDate)
{
        int start = 0, ptr;                     // To mark positions for substring
actions
        char tempCharArray[5];                // For text to number conversions
        string tempStr;                       // Temporary holding string

        // Changed Ordering

        // Get birth year
```

```
	ptr = codedDate.find('-', start);                    // Find first slash
	tempStr = codedDate.substr(0, ptr);
	//strcpy_s(tempCharArray, tempStr.data());
	year = atoi(tempStr.c_str());                   // Assign year


	// Get birth month
	ptr = codedDate.find('-', 0);                   // Find last dash
	tempStr = codedDate.substr(start, ptr - start);
	//strcpy_s(tempCharArray, tempStr.data());
	month = atoi(tempStr.c_str());                   // Assign month
	start = ptr + 1;


	// Get birth day
	ptr = codedDate.length();              // Find end of dash
	tempStr = codedDate.substr(start, ptr - start);
	//strcpy_s(tempCharArray, tempStr.data());         // Assign day
	day = atoi(tempStr.c_str());


}


//---------------------------------------------------------------
// SET functions
void Date::setMonth(int m)
{
	month = m;
}

void Date::setDay(int d)
{
	day = d;
}

void Date::setYear(int y)
{
	year = y;
}

//---------------------------------------------------------------
// GET functions
int Date::getMonth()  // Return current month value
{
	return month;
}

int Date::getDay()    // Return current day value
{
	return day;
}

int Date::getYear()   // Return current year value
{
	return year;
}
//---------------------------------------------------------------
```

```
// This function returns true if the year is a leap year and false
// otherwise.

bool Date::leapYear()
{
	if (year % 400 == 0 ||
		(year % 4 == 0 && year % 100 != 0))
		return true;
	else
		return false;

}  // end function leapYear

//-----------------------------------------------------------------
// This function returns an integer of the number of days in the
// month.  Leap years are considered.

int Date::daysInMonth()
{
	int days = 0;

	// 31 Day theMonths
	if (month == 1 || month == 3 || month == 5 ||
		month == 7 || month == 8 || month == 10 ||
		month == 12)
		days = 31;

	// 30 Day theMonths
	else if (month == 4 || month == 6 ||
		month == 9 || month == 11)
		days = 30;

	// February
	else  // month== 2
		if (leapYear())
			days = 29;
		else
			days = 28;

	return days;

}  // end function DaysInMonth

//-----------------------------------------------------------------
// This function returns the Julian date (the day number of the date
// in that year).

int Date::julianDate()
{
	int dayCnt = 0;
	int the_mon;

	int FebDays;
	if (leapYear())
		FebDays = 29;
	else
		FebDays = 28;
```

```
	for (the_mon = 1; the_mon < month; the_mon++)
		switch (the_mon)
	{
		case 2:		dayCnt += FebDays; break;
		case 4:
		case 6:
		case 9:
		case 11:	dayCnt += 30; break;
		default:	dayCnt += 31;
	};
	dayCnt += day;

	return dayCnt;

}  // end function julianDate

//---------------------------------------------------------------------
// This method returns a boolean value defining the validity of the
// date.
bool Date::validDate()
{
	bool valDate = true;   // Assume a good date

	// Test for conditions that would make the date validity false
	if (year < 1900)
		valDate = false;
	if ((month< 1) || (month> 12) || (day < 1) || (day > 31))
		valDate = false;
	else if (((month == 4) || (month == 6) || (month == 9) || (month == 11)) && (day
== 31))
		valDate = false;
	else if ((month == 2) && leapYear() && (day > 29))
		valDate = false;
	else if ((month == 2) && !leapYear() && (day > 28))
		valDate = false;

	return valDate;
}  // end function validDate

//---------------------------------------------------------------------
// This function returns a date code for the day of the week.  It
// counts the number of days since 1/1/1900 which was on a Sunday.
// Output is: 0=Sun,1=Mon, ..., 6=Sat.

int Date::weekDay()
{
	int DayCnt;
	int daynum, i;

	DayCnt = (year - 1900) * 365;
	DayCnt += ((year - 1900) / 4) + 1;
	for (i = 1; i <= month - 1; i++)
		switch (i)
	{
		case 2:		DayCnt += 28; break;
		case 4:
		case 6:
		case 9:
```

```
                case 11:    DayCnt += 30; break;
                default:    DayCnt += 31;
        };

        if (((year - 1900) % 4 == 0) && (month <= 2))
                DayCnt--;
        DayCnt += day;
        daynum = (DayCnt - 1) % 7;

        return daynum;
} // end function weekDay


/*
//---------------------------------------------------------------------
// This function returns (via the parameter list) the 3-character
// descriptor for the day of the week the date represents
void Date::dayCode(char descript[])
{
        int code = weekDay();   // Get week day code for THIS date

        switch (code)
        {
        case 0: strcpy(descript, "SUN"); break;
        case 1: strcpy(descript, "MON"); break;
        case 2: strcpy(descript, "TUE"); break;
        case 3: strcpy(descript, "WED"); break;
        case 4: strcpy(descript, "THU"); break;
        case 5: strcpy(descript, "FRI"); break;
        case 6: strcpy(descript, "SAT"); break;
        }; // end switch
}


//---------------------------------------------------------------------
// This function returns (via the parameter list) the string
// descriptor for the month the date represents
void  Date::monthCode(char descript[])
{
        switch (month)
        {
        case 1: strcpy(descript, "January"); break;
        case 2: strcpy(descript, "February"); break;
        case 3: strcpy(descript, "March"); break;
        case 4: strcpy(descript, "April"); break;
        case 5: strcpy(descript, "May"); break;
        case 6: strcpy(descript, "June"); break;
        case 7: strcpy(descript, "July"); break;
        case 8: strcpy(descript, "August"); break;
        case 9: strcpy(descript, "September"); break;
        case 10: strcpy(descript, "October"); break;
        case 11: strcpy(descript, "November"); break;
        case 12: strcpy(descript, "December"); break;
        }; // end switch
}

*/
//---------------------------------------------------------------------
```

```
// Comparison operation for equality; returns true if dates identical
bool Date::operator==(Date secondDate)
{
	if ((month == secondDate.month) && (day == secondDate.day) &&
		(year == secondDate.year))
		return true;
	else
		return false;
}  // end function EqualTo

//-----------------------------------------------------------------
// Comparison operation for less than; returns true referencing date
// (1st date) is less than date in parameter
bool Date::operator<(Date secondDate)
{
	bool outcome = false;		// Assume date not less than
	if (year < secondDate.year)
		outcome = true;
	else if (year == secondDate.year)
		if (month < secondDate.month)
			outcome = true;
		else if (month == secondDate.month)
			if (day < secondDate.day)
				outcome = true;
	return outcome;
}  // end function LessThan


// Overload the insertion operator to enable console output
ostream& operator<< (ostream &strm, Date &theObj)
{
	strm << theObj.month << "/" << theObj.day << "/" << theObj.year;
	return strm;
}
```