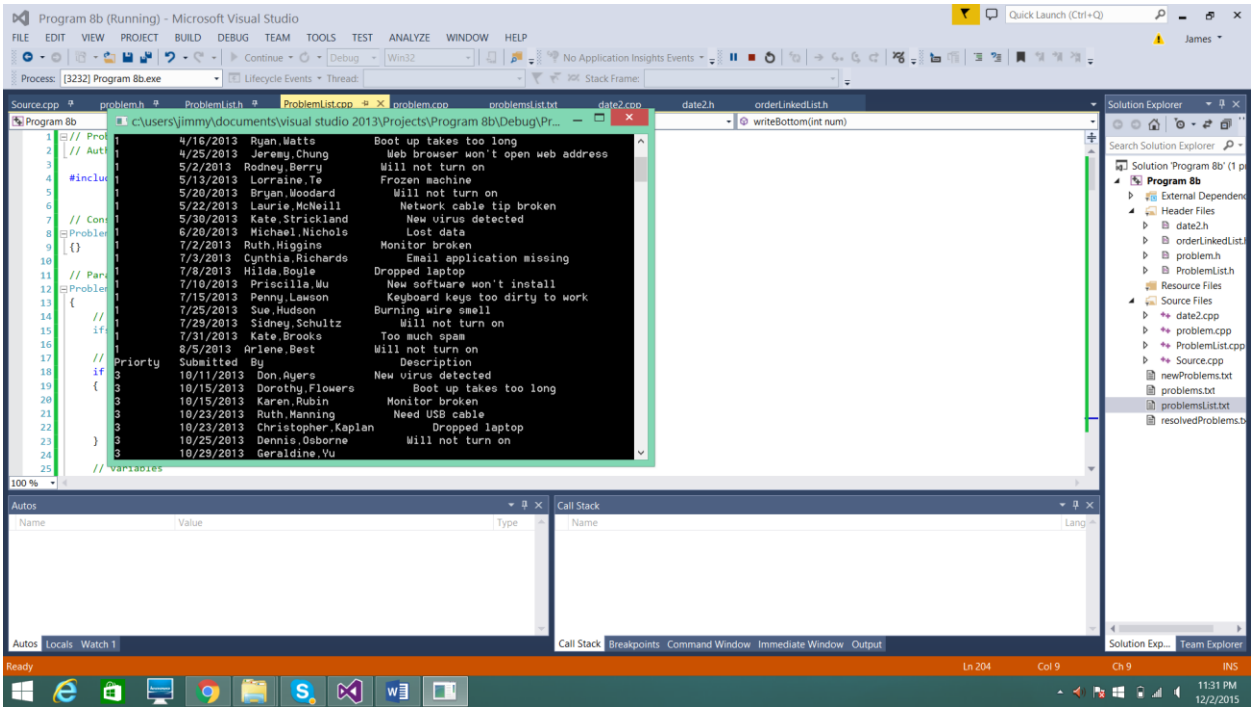
# Program 8 James Wetters




Out Put

| Priorty | Submitted | By | Description |
|---|---|---|---|
| 1 | 3/13/2013 | Stephanie,Winstead | Email inbox full |
| 1 | 3/18/2013 | Ian,Blanton | USB port dead |
| 1 | 3/21/2013 | Mike,Shields | Fist damange to monitor |
| 1 | 3/21/2013 | Tim,Frank | Computer running especially slow |
| 1 | 3/22/2013 | Philip,Tuttle | File will not print |
| 1 | 3/28/2013 | Jack,Winstead | |
| 1 | 4/11/2013 | Wayne,Fitzpatrick | Will not turn on |
| 1 | 4/12/2013 | Danielle,Kaufman | Monitor broken |
| 1 | 4/16/2013 | Ryan,Watts | Boot up takes too long |
| 1 | 4/25/2013 | Jeremy,Chung | Web browser won't open web address |

| | | | |
|---|---|---|---|
| 1 | 5/2/2013 | Rodney,Berry | Will not turn on |
| 1 | 5/13/2013 | Lorraine,Te | Frozen machine |
| 1 | 5/20/2013 | Bryan,Woodard | Will not turn on |
| 1 | 5/22/2013 | Laurie,McNeill | Network cable tip broken |
| 1 | 5/30/2013 | Kate,Strickland | New virus detected |
| 1 | 6/20/2013 | Michael,Nichols | Lost data |
| 1 | 7/2/2013 | Ruth,Higgins | Monitor broken |
| 1 | 7/3/2013 | Cynthia,Richards | Email application missing |
| 1 | 7/8/2013 | Hilda,Boyle | Dropped laptop |
| 1 | 7/10/2013 | Priscilla,Wu | New software won't install |
| 1 | 7/15/2013 | Penny,Lawson | Keyboard keys too dirty to work |
| 1 | 7/25/2013 | Sue,Hudson | Burning wire smell |
| 1 | 7/29/2013 | Sidney,Schultz | Will not turn on |
| 1 | 7/31/2013 | Kate,Brooks | Too much spam |
| 1 | 8/5/2013 | Arlene,Best | Will not turn on |

| Priorty | Submitted | By | Description |
|---|---|---|---|
| 3 | 10/11/2013 | Don,Ayers | New virus detected |
| 3 | 10/15/2013 | Dorothy,Flowers | Boot up takes too long |
| 3 | 10/15/2013 | Karen,Rubin | Monitor broken |
| 3 | 10/23/2013 | Ruth,Manning | Need USB cable |
| 3 | 10/23/2013 | Christopher,Kaplan | Dropped laptop |
| 3 | 10/25/2013 | Dennis,Osborne | Will not turn on |
| 3 | 10/29/2013 | Geraldine,Yu | |
| 3 | 11/1/2013 | Clifford,Wooten | New software won't install |
| 3 | 11/4/2013 | Joel,Winstead | Lost data |

```
3          11/6/2013  Eddie,Moore          Dropped laptop
3          11/14/2013  Marcus,Forbes          Boot up takes too
long
3          11/19/2013  Stacey,Harmon          File will not print
3          11/19/2013  Harvey,Bowling          Slow network
3          11/20/2013  Dawn,Newton          Burning wire smell
3          11/21/2013  Neil,Dickens          File will not print
3          11/22/2013  Erik,Young          Too much spam
3          11/29/2013  Vicki,Carlton          Frozen machine
3          12/9/2013  Carl,Pace          Burning wire smell
3          12/11/2013  Marilyn,Hanna          Too much spam
3          12/16/2013  Julian,Britt          New software crashes
3          12/17/2013  Mike,McKenna          Lost data
3          12/18/2013  Erica,Wong          Network cable tip
broken
3          12/18/2013  Dianne,Strickland          Hard drive not
storing files
3          12/20/2013  Katie,Noble          New software crashes
3          12/23/2013  Jean,Williford          Will not turn on
Press any key to continue . . .
```

```
// This program manages a list of computer problems and sorts
// each problem based on criticality
// Author James Wetters

#include <iostream>
#include <string>
#include <iomanip>

#include "ProblemList.h"

using namespace std;

int main()
{
      // Create problem lists
```

```
	ProblemList problems("problems.txt");
	ProblemList newProblems("newProblems.txt");
	ProblemList solvedProblems("resolvedProblems.txt");

	// Add new problems to the list
	problems += newProblems;
	// Delete old problems from the list
	problems -= solvedProblems;

	// Write top 25 problems
	problems.writeTop(25);
	// Write bottome 25 problems
	problems.writeBottom(25);

	system("pause");
	return 0;
}
```

```
// Problem Header
// Author James Wetters

#ifndef PROBLEM_H
#define PROBLEM_H

#include <iostream>
#include <string>

using namespace std;
#include "date2.h"

class Problem
{
private:
	// Variables
	int problemCode;
	int critLevel;
	Date date;
	string contact;

public:
	// Gets
	int getProblemCode()
	{
		return problemCode;
	}
	int getCritLevel()
	{
		return critLevel;
	}
	Date getDate()
	{
		return date;
	}
	string getContact()
```

```
	{
		return contact;
	}

	// Sets
	void setProblemCode(int change)
	{
		problemCode = change;
	}

	void setCritLevel(int change)
	{
		critLevel = change;
	}

	void setDate(Date change)
	{
		date = change;
	}

	void setContact(string change)
	{
		contact = change;
	}

	// Problem constructor
	Problem();
	// Problem paramaterized constructor
	Problem(int cCode, int cCrit, Date cDate, string cCon);

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


};
#endif


// Problem cpp file
// Author James Wetters

#include "problem.h"

// Constructor
Problem::Problem()
{}

// Paramaterized Constructor
Problem::Problem(int cCode, int cCrit, Date cDate, string cCon)
{
	setProblemCode(cCode);
	setCritLevel(cCrit);
	setDate(cDate);
	setContact(cCon);
}
```

```
//---------------------------------------------------------------
// Overloaded operator < less than
//---------------------------------------------------------------
bool Problem::operator< (Problem& p)
{
	if (critLevel < p.critLevel)
		return true;

	if (critLevel == p.critLevel && date < p.date)
		return true;

	return false;
}

//---------------------------------------------------------------
// Overloaded operator == equal to
//---------------------------------------------------------------
bool Problem::operator== (Problem& p)
{

	if (critLevel == p.critLevel && date == p.date)
	{
		return true;
	}

	return false;
}

//---------------------------------------------------------------
// Overloaded operator != not equal to
//---------------------------------------------------------------
bool Problem::operator!= (Problem& p)
{

	if (critLevel != p.critLevel || !(date == p.date))
	{
		return true;
	}

	return false;
}

// ProblemList Header
// Author James Wetters

#ifndef PROBLEMLIST_H
#define PROBLEMLIST_H

#include <iostream>
#include <string>
#include <fstream>
#include "orderLinkedList.h"
#include "problem.h"

using namespace std;

// Constants
const int MAXPARRAY = 30;
```

```
class ProblemList
{
private:
	OrderLinkedList<Problem> theProblemList;
	int problemID[MAXPARRAY];
	string problemName[MAXPARRAY];

public:
	// Problem List constructor
	ProblemList();
	// Problem List paramaterized constructor
	ProblemList(string theTextFileName);

	// Problem List Overloaded Oporators
	ProblemList ProblemList::operator+=(ProblemList pL);
	ProblemList ProblemList::operator-=(ProblemList pL);

	// Input Problem codes and descriptions
	void inputProblemListCodes();

	// Write the problem code description
	void ProblemList::writeProblemCode(int probNum);

	// Write Problem List
	void ProblemList::writeTop(int num);
	// Write Problem List
	void ProblemList::writeBottom(int num);

};
#endif

// Problem List cpp
// Author James Wetters

#include "ProblemList.h"


// Constructor
ProblemList::ProblemList()
{}

// Paramaterized Constructor
ProblemList::ProblemList(string theTextFileName)
{
	// Open File
	ifstream inputFile(theTextFileName.c_str());

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

	// variables
	int pCTempInt, critTempInt;
```

```
	string pCTemp, critTemp, dTemp, contTemp;

	// Priming read
	getline(inputFile, pCTemp, ',');

	// Read in data file
	while (!inputFile.eof())
	{
		// Read in data
		getline(inputFile, critTemp, ',');
		getline(inputFile, dTemp, ',');
		getline(inputFile, contTemp);

		// Convert string to int
		pCTempInt = atoi(pCTemp.c_str());
		critTempInt = atoi(critTemp.c_str());

		// Create Temp Problem Object
		Problem p(pCTempInt, critTempInt, dTemp, contTemp);

		// Insert into Problem List
		theProblemList.insertNode(p);

		// Prime the next line
		getline(inputFile, pCTemp, ',');
	}


	// Close file
	inputFile.close();
	inputFile.clear();

	// List Of Problems
	inputProblemListCodes();

}

//--------------------------------------------------------------------
//	Operator +=
//
// Sets += operator to add problem lists together
//--------------------------------------------------------------------
ProblemList ProblemList::operator+=(ProblemList pL)
{
	pL.theProblemList.resetList();

	while (!pL.theProblemList.atEnd())
	{
		Problem temp = pL.theProblemList.getNextItem();

		theProblemList.insertNode(temp);
	}

	return *this;
}

//--------------------------------------------------------------------
//	Operator -=
```

```
//
// Sets -= operator to subtract problem lists from each other
//-----------------------------------------------------------------
ProblemList ProblemList::operator-=(ProblemList pL)
{
	pL.theProblemList.resetList();

	while (!pL.theProblemList.atEnd())
	{
		Problem temp = pL.theProblemList.getNextItem();

		theProblemList.deleteNode(temp);
	}
	return *this;
}

//-----------------------------------------------------------------
//	Input Problem List Codes
//
// Reads in problem list codes and descriptions and sets them to
// parallel arrays
//-----------------------------------------------------------------
void ProblemList::inputProblemListCodes()
{
	// Open File
	ifstream inFile("problemsList.txt");

	// Test File

	if (inFile.fail())
	{
		cout << "Problem opening file";
		system("pause");
		exit(-1);
	}

	// variables
	int iDTemp, numElems = 0;
	string desTemp, iDStrTemp;

	// Priming read
	getline(inFile, iDStrTemp, '-');

	// Read in data file
	while (!inFile.eof())
	{
		// Convert id from string to int
		iDTemp = atoi(iDStrTemp.c_str());

		// Set iD to a spot in the array
		problemID[numElems] = iDTemp;

		// Get the description
		getline(inFile, desTemp);

		problemName[numElems] = desTemp;

		numElems++;
```

```
			// Get next line
			getline(inFile, iDStrTemp, '-');
		}


		// Close file
		inFile.close();
		inFile.clear();
}

//-----------------------------------------------------------------
//	Write Problem Code
//
// Recives a problem number as an int
//
// Writes problem description
//-----------------------------------------------------------------
void ProblemList::writeProblemCode(int probNum)
{
		// Initilize variables
		int index = 0;
		bool found = false;

		// While id not found and not greater than the array search
		while (found != true && index < MAXPARRAY)
		{
			// If prolem is found get out
			if (problemID[index] == probNum)
			{
				found = true;
			}

			index++;
		}

		// If problem is found write the problem description
		if (found == true)
		{
			cout << problemName[index] << "\n";
		}

}

//-----------------------------------------------------------------
//	Write Top
//
// Recives the number of objects to display
//
// Writes number of objects from the top
//-----------------------------------------------------------------
void ProblemList::writeTop(int num)
{
		// Write header
		cout << "Priorty" << "    " << "Submitted" << "  " << "By" << "
"
			<< "Description" << endl;
```

```
	// Reset the problem list from the beginning
	theProblemList.resetList();

	// Write number of objects
	for (int i = 0; i < num; i++)
	{
		// Initilize variables
		Problem p = theProblemList.getNextItem();
		int probNumCode = p.getProblemCode();

		// Write the object
		cout << p.getCritLevel() << "         " << p.getDate() <<
			"  " << p.getContact() << "         ";
		// Write the problem description
		writeProblemCode(probNumCode);
	}

}

//-----------------------------------------------------------------
//	Write Bottom
//
// Recives the number of objects to display
//
// Writes number of objects from the bottom
//-----------------------------------------------------------------
void ProblemList::writeBottom(int num)
{
	// Initilize variables
	int numFromEnd;
	numFromEnd = theProblemList.getLength() - num;

	// Write header
	cout << "Priorty" << "   " << "Submitted" << "  " << "By" << "
"
		<< "Description" << endl;

	// Reset the problem list from the beginning
	theProblemList.resetList();

	// Count back from the end
	for (int i = 0; i < numFromEnd; i++)
	{
		Problem p = theProblemList.getNextItem();
	}

	// Write number of objects
	for (int i = 0; i < num; i++)
	{
		// Initilize variables
		Problem p = theProblemList.getNextItem();
		int probNumCode = p.getProblemCode();

		// Write the object
		cout << p.getCritLevel() << "         " << p.getDate() <<
			"  " << p.getContact() << "         ";
		// Write the problem description
		writeProblemCode(probNumCode);
```

```
	}

}



// Class Modified Files


//*******************************************
// The ListNode class creates a type used to  *
// store a node of the linked list.           *
// PRECONDITIONS:                             *
//   Choice for ItemType implements 'cout'    *
//   as well as "==" and "<" operators        *
//*******************************************
#ifndef OrderOrderLinkedList_H
#define OrderOrderLinkedList_H

template <class ItemType>
class ListNode
{
public:
	ItemType info;                // Node value
	ListNode<ItemType> *next;     // Pointer to the next node

	// Constructor
	ListNode(ItemType nodeValue)
	{
		info = nodeValue;
		next = NULL;
	}
};

//*******************************************
// OrderLinkedList class                      *
//*******************************************

template <class ItemType>
class OrderLinkedList
{
private:
	ListNode<ItemType> *head;        // List head pointer
	ListNode<ItemType> *currentPos;  // Pointer to "current" list item
	int length;                      // Length

public:
	OrderLinkedList();                                                  // Constructor
	~OrderLinkedList();                                                 // Destructor
	OrderLinkedList(const OrderLinkedList<ItemType>& anotherList); // Copy constructor
	void operator= (const OrderLinkedList<ItemType>&);             // Assignment op

	void insertNode(ItemType);
	void deleteNode(ItemType);
	bool searchList(ItemType& item);
```

```
        int getLength();
        void displayList();

        void resetList();
        ItemType getNextItem();        // Iterator
        bool atEnd();
};

//************************************************
// Constructor                                   *
// Initial list head pointer and length          *
//************************************************
template <class ItemType>
OrderLinkedList<ItemType>::OrderLinkedList()
{
        head = NULL;
        length = 0;
}

//************************************************
// displayList shows the value stored in each node *
// of the linked list pointed to by head.        *
// Precondition: "cout" operator enabled for     *
// ItemType data type.                           *
//************************************************

template <class ItemType>
void OrderLinkedList<ItemType>::displayList()
{
        ListNode<ItemType> *nodePtr;

        nodePtr = head;
        while (nodePtr != NULL)
        {
                cout << nodePtr->info << endl;
                nodePtr = nodePtr->next;
        }
}
//************************************************
// The insertNode function inserts a node with   *
// newValue copied to its value member.          *
//************************************************

template <class ItemType>
void OrderLinkedList<ItemType>::insertNode(ItemType newValue)
{
        ListNode<ItemType> *newNode, *nodePtr, *previousNode = NULL;

        // Allocate a new node & store newValue
        newNode = new ListNode<ItemType>(newValue);

        // If there are no nodes in the list
        // make newNode the first node
        if (head == NULL)
        {
                head = newNode;
                newNode->next = NULL;
```

```
	}
	else    // Otherwise, insert newNode
	{
		// Initialize nodePtr to head of list and previousNode to NULL.
		nodePtr = head;
		previousNode = NULL;

		// Skip all nodes whose value member is less
		// than newValue.
		while (nodePtr != NULL && nodePtr->info < newValue)
		{
			previousNode = nodePtr;
			nodePtr = nodePtr->next;
		}

		// If the new node is to be the 1st in the list,
		// insert it before all other nodes.
		if (previousNode == NULL)
		{
			head = newNode;
			newNode->next = nodePtr;
		}
		else    // Otherwise, insert it after the prev. node.
		{
			previousNode->next = newNode;
			newNode->next = nodePtr;
		}
		length++;
	}

}

//********************************************************
// The deleteNode function searches for a node        *
// with searchValue as its value. The node, if found, *
// is deleted from the list and from memory.          *
//********************************************************

template <class ItemType>
void OrderLinkedList<ItemType>::deleteNode(ItemType searchValue)
{
	ListNode<ItemType> *nodePtr, *previousNode;

	// If the list is empty, do nothing.
	if (!head)
		return;

	// Determine if the first node is the one.
	if (head->info == searchValue)
	{
		nodePtr = head->next;
		delete head;
		head = nodePtr;
	}
	else
	{
		// Initialize nodePtr to head of list
		//nodePtr = head;
```

```
		nodePtr = previousNode = head;		//This had to be added to be
compiled

		// Skip all nodes whose value member is
		// not equal to searchValue.
		while (nodePtr != NULL && nodePtr->info != searchValue)
		{
			previousNode = nodePtr;
			nodePtr = nodePtr->next;
		}

		// If nodePtr is not at the end of the list,
		// link the previous node to the node after
		// nodePtr, then delete nodePtr.
		if (nodePtr)
		{
			previousNode->next = nodePtr->next;
			delete nodePtr;
		}
	}
	length--;
}

//************************************************
// Linear search                                 *
// Post: If found, item's key matches an element's *
// key in the list and a copy of that element has  *
// been stored in item; otherwise, item is         *
// unchanged.  Return value is boolean to indicate *
// status of search.                             *
//************************************************

template <class ItemType>
bool OrderLinkedList<ItemType>::searchList(ItemType& item)
{
	bool moreToSearch;
	ListNode<ItemType>* nodePtr;

	nodePtr = head;			// Start search from head of list
	bool found = false;		// Assume value not found
	moreToSearch = (nodePtr != NULL);

	while (moreToSearch && !found)
	{
		if (nodePtr->info < item)
		{
			nodePtr = nodePtr->next;
			moreToSearch = (nodePtr != NULL);
		}
		else if (item == nodePtr->info)
		{
			found = true;
			item = nodePtr->info;
		}
		else
			moreToSearch = false;
	}
	return found;
```

```
}

//************************************************
// Iterator reset function                       *
// Resets pointer of current item in list to the  *
// head of the list.                             *
//************************************************

template <class ItemType>
void OrderLinkedList<ItemType>::resetList()
// Post: Current position has been initialized.
{
	currentPos = head;
}

//************************************************
// Function: Gets the next element in list as
//           referenced by currPtr
// Pre:  Current position is defined.
//       Element at current position is not last in list.
// Post: Current position is updated to next position.
//       item is a copy of element at current position.
//************************************************
template <class ItemType>
ItemType OrderLinkedList<ItemType>::getNextItem()
{
	ItemType item;

	if (currentPos == NULL)
		currentPos = head;  // wrap if getnext is called at past-end
	//else
	item = currentPos->info;
	currentPos = currentPos->next;

	return item;
}

//************************************************
// Observer function to return current list length *
//************************************************
template <class ItemType>
int OrderLinkedList<ItemType>::getLength()
{
	return length;
}

//************************************************
// Observer function to determine if current      *
// is the end of the list                         *
//************************************************
template <class ItemType>
bool OrderLinkedList<ItemType>::atEnd()
{
	if (currentPos == NULL)
		return true;
	else
		return false;
}
```

```
//*************************************************
// Copy Constructor                               *
//*************************************************
template<class ItemType>
OrderLinkedList<ItemType>::OrderLinkedList(const OrderLinkedList<ItemType>& anotherList)
{
	ListNode<ItemType>* ptr1;
	ListNode<ItemType>* ptr2;

	if (anotherList.head == NULL)
		head = NULL;
	else
	{
		head = new ListNode<ItemType>(anotherList.head->info);
		ptr1 = anotherList.head->next;
		ptr2 = head;
		while (ptr1 != NULL)
		{
			ptr2->next = new ListNode<ItemType>(ptr1->info);
			ptr2 = ptr2->next;
			ptr1 = ptr1->next;
		}
		ptr2->next = NULL;
	}
	length = anotherList.length;
}

//*************************************************
// Overloaded Assignment Operator                 *
//*************************************************
template<class ItemType>
void OrderLinkedList<ItemType>::operator=(const OrderLinkedList<ItemType>& anotherList)
{
	ListNode<ItemType>* ptr1;
	ListNode<ItemType>* ptr2;

	if (anotherList.head == NULL)
		head = NULL;
	else
	{
		head = new ListNode<ItemType>(anotherList.head->info);
		ptr1 = anotherList.head->next;
		ptr2 = head;
		while (ptr1 != NULL)
		{
			ptr2->next = new ListNode<ItemType>(ptr1->info);
			ptr2 = ptr2->next;
			ptr1 = ptr1->next;
		}
		ptr2->next = NULL;
	}
	length = anotherList.length;
}

//*************************************************
// Destructor                                     *
// This function deletes every node in the list.  *
```

```
//************************************************

template <class ItemType>
OrderLinkedList<ItemType>::~OrderLinkedList()
{
	ListNode<ItemType> *nodePtr, *nextNode;

	nodePtr = head;
	while (nodePtr != NULL)
	{
		nextNode = nodePtr->next;
		delete nodePtr;
		nodePtr = nextNode;
	}
}

#endif



// Date.h
// This file defines the specifications for the Date class.  This class
// is a utility for any work with calendar dates.
#include <ostream>

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);
};


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

#include <iostream>
#include <string>
#include <cstring>
using namespace std;
#include "date2.h"
```

```
#define _CRT_SECURE_NO_WARNINGS 1



//---------------------------------------------------------------
// 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, ptr;			// To mark positions for substring actions
	char tempCharArray[5];		// For text to number conversions
	string tempStr;			// Temporary holding string

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

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

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

}


//---------------------------------------------------------------
// 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_s(descript, "SUN"); break;
	case 1: strcpy_s(descript, "MON"); break;
	case 2: strcpy_s(descript, "TUE"); break;
	case 3: strcpy_s(descript, "WED"); break;
	case 4: strcpy_s(descript, "THU"); break;
	case 5: strcpy_s(descript, "FRI"); break;
```

```
	case 6: strcpy_s(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_s(descript, "January"); break;
	case 2: strcpy_s(descript, "February"); break;
	case 3: strcpy_s(descript, "March"); break;
	case 4: strcpy_s(descript, "April"); break;
	case 5: strcpy_s(descript, "May"); break;
	case 6: strcpy_s(descript, "June"); break;
	case 7: strcpy_s(descript, "July"); break;
	case 8: strcpy_s(descript, "August"); break;
	case 9: strcpy_s(descript, "September"); break;
	case 10: strcpy_s(descript, "October"); break;
	case 11: strcpy_s(descript, "November"); break;
	case 12: strcpy_s(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;
}
```