Spike: *4*
Title: None Blocking Game Loop

Author: Steven Efthimiadis, 1627406

Goals / deliverables:
The goal is to create a game loop that continuously runs while waiting for a trigger to activate it. Use the Gridworld game made in Spike 1 to build on.

To create this spike, you require:

- Spike 1
- Thread that take input
- Thread that operates the output and rendering

Technologies, Tools, and Resources used:
List of information needed by someone trying to reproduce this work

- Visual Studio 2015
- Multithreading
    - https://solarianprogrammer.com/2011/12/16/cpp-11-thread-tutorial/

Tasks undertaken:

- Copy Spike 1 into a new project
- Separate the input into one thread
- Separate the output and rendering into another thread

What we found out:

- By separating the input and rendering into 2 threads we stop the blocking game loop by the threads waiting to be activated and when it's complete it will delete the thread.

```
/*****************************************************************************
* An implementatrion of a double-linked list (http://en.wikipedia.org/wiki/Double_linked_list)
* This code has 13 bugs. You don't have to find all of them.
* Some bugs will be easy to find, some will be quite hard.
* You do not need to understand what this code does in order to find the easier bugs,
* and you certainly don't need to understand it to finish the spike.
* You'll need to understand the code to go 13 for 13, but consider that a challenge,
* rather than a requirement.
* As its written, none of the bugs will prevent the the code from compiling.
* (Tested with Visual Studio 2010 under Windows 7)
*****************************************************************************/

#include <string>
#include <iostream>
#include <stdexcept>
#include "Spike3.h"

using namespace std;

/*************************************************************************************************
* DoubleLinkedNode class definition. DoubleLinkedNode forms a single node in a doubly linked list.
* Templates are used so the list may be made of any datatype.
*************************************************************************************************/
template <class DataType>
class DoubleLinkedNode {
public:
	typedef DoubleLinkedNode<DataType> Node;

	/*********************************************************************************************
	* DoubleLinkedNode Class Constructor: Creates DoubleLinkedNode object with value = initvalue
	*********************************************************************************************/
	DoubleLinkedNode (const DataType initvalue) : value(initvalue){
		cout << "Creating Node..."<< endl;
		// Error 2: nextnode and perviousnode are not set
		nextnode = NULL;
		previousnode = NULL;
	}

	/*********************************************************************************************
	* DoubleLinkedNode Class Destructor: Destroys DoubleLinkedNode object, freeing memory
	*********************************************************************************************/
	~DoubleLinkedNode () {
		cout << "Destroying Node... " << endl;
	}

	/*********************************************************************************************
	* Function: Inserts the Node newnode behind this Node, moving all other Nodes down one.
	* Input: Node*. The Node to be inserted.
	* Output: -
	*********************************************************************************************/
	void insertNodeAfter(Node* newnode){
		if(nextnode){
			newnode->insertNodeAfter(nextnode);
		}
		setNext(newnode);
		newnode->setPrevious(this);
	};

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

```
	* Function: Inserts the Node newnode in front of this Node, moving all other Nodes, including this
node down one.
	* Input: Node*. The Node to be inserted.
	* Output: -
	*********************************************************************************************/
	void insertNodeBefore(Node* newnode){
		if(previousnode){
			newnode->insertNodeBefore(nextnode); //should be newnode-
>insertNodeBefore(previousnode);
		}
		setPrevious(newnode);
		newnode->setNext(this);
	};

	/*********************************************************************************************
	* Function: Removes this Node from the list, reconnecting other nodes so the list is unbroken.
	* Input: -
	* Output: -
	*********************************************************************************************/
	void dropNode(){
		if(previousnode){
			previousnode->setNext(nextnode);
		}
		if(nextnode){
			nextnode->setPrevious(previousnode);
		}
		delete this;
	};

	/*********************************************************************************************
	* Function: Returns the next Node in the list.
	* Input: -
	* Output: Node*. The next Node in the list.
	*********************************************************************************************/
	Node* getNext() const{
		return nextnode;
	};

	/*********************************************************************************************
	* Function: Returns the previous Node in the list.
	* Input: -
	* Output: Node*. The previous Node in the list.
	*********************************************************************************************/
	Node* getPrevious() const{
		return previousnode;
	};

	/*********************************************************************************************
	* Function: Sets the next Node in the list.
	* Input: Node*. A pointer to the Node that is to be made the next Node in the list.
	* Output: -
	*********************************************************************************************/
	void setNext(Node* N){
		nextnode = N;
	}

	/*********************************************************************************************
	* Function: Sets the previous Node in the list.
	* Input: Node*. A pointer to the Node that is to be made the previous Node in the list.
	* Output: -
```

```
	*****************************************************************************************/
	void setPrevious(Node* N){
		previousnode = N;
	}

	/*****************************************************************************************
	* Function: Returns the data contained in this this Node.
	* Input: -
	* Output: const DataType&. The data contained in this Node.
	*****************************************************************************************/
	const DataType& getValue() const{
		return value;
	};

private:
	//The data contained in this node
	DataType value;
	//The next Node in the list
	Node* nextnode;
	//The previous Node in the list
	Node* previousnode;
};

/*****************************************************************************************
* DoubleLinkedList class definition. Constructs a double linked list made up of DoubleLinkedNode objects.
* Templates are used so the list and its nodes may be made of any datatype.
*****************************************************************************************/
template <class T>
class DoubleLinkedList {
private:
	typedef DoubleLinkedNode<T> Node;

	//The first node in the list
	Node* first;
	//The last node in the list
	Node* last;
	//The length of the list
	int _length;
public:
	/*****************************************************************************************
	* List Class Constructor: Creates List object.
	* The list is created without any contents.
	*****************************************************************************************/
	DoubleLinkedList() : first(0), last(0), _length(0){};

	/*****************************************************************************************
	* List Class Destructor: Destroys all nodes allocated as a part of the list and frees memory.
	*****************************************************************************************/
	~DoubleLinkedList(){
		while(first->getNext() != (Node*)0){
			first->getNext()->dropNode();
		}
	};


	/*****************************************************************************************
	* Function: Appends the Node newelement to the end of the List
	*****************************************************************************************/
	void append(const T &newelement){
		Node *N = new Node(newelement);
```

```
        if(first == (Node*)0){
                first = N;
                last = N;
                _length = 1;
        }else{
                last->insertNodeAfter(N);
                last = N;
                _length++;
        }
};

/*********************************************************************************************
* Function: Appends the Node newelement to the end of the List
*********************************************************************************************/
void prepend(const T &newelement){
        Node *N = new Node(newelement);

        if(first == (Node*)0){
                first = N;
                last = N;
                _length = 1;
        }else{
                first->insertNodeBefore(N);
                first = N;
                _length++;
        }
};

/*********************************************************************************************
* Function: Drops the first Node found with a value matching element, if one exits. If a node
* was found and dropped, true is returned, false otherwise.
*********************************************************************************************/
bool drop(const T &element){
        // Error 8: Not reducing the size of the list when deleting an item
        if(first->getValue() == element){
                dropFirst();
                _length--;
                return true;
        }else if(last->getValue() == element){
                dropLast();
                _length--;
                return true;
        }

        Node *N = first;
        while(N != last){
                if(N->getValue() == element){
                        N->dropNode();
                        _length--;
                        return true;
                }
                // Error 5: N is not looping to the next value
                N = N->getNext();
        }
        return false;
};

/*********************************************************************************************
* Function: Drops the first Node in the List.
```

```
	**********************************************************************************************/
	void dropFirst(){
		first = first->getNext();
		first->getPrevious()->dropNode();
	};

	/**********************************************************************************************
	* Function: Drops the last Node in the List.
	**********************************************************************************************/
	void dropLast(){
		last = last->getPrevious();
		last->getNext()->dropNode();
	};

	/**********************************************************************************************
	* Function: Iterates over the contents of the list, printing the value of each node in turn.
	**********************************************************************************************/
	void print (void) {
		// Error 3: N is not initialized
		Node *N;
		N = first;
		bool lastNode = false;
		do {
			cout << N->getValue() << endl;
			// Error 4: N is not looping and printing out the next value
			N = N->getNext();
			// Error 7 : Not printing the last value of the list
			if (N == last) {
				cout << N->getValue() << endl;
				break;
			}
		} while (!lastNode);
		cout << "-----------------------------------------------" << endl;
	}
};

int main(int argc, char* argv[]){
	string s1("One");
	string s2("Two");
	string s3("Three");
	string s4("Four");
	string s5("Five");
	string s6("Six");


	DoubleLinkedList<string>* L;
	// Error 1: L is not initialized
	L = new DoubleLinkedList<string>();
	//Add some numbers to the list
	L->append(s3);
	L->append(s4);
	L->append(s5);
	L->print();			//Looks good, but we forgot One and Two
	//Lets add them
	L->prepend(s1);
	L->prepend(s2);
	L->print();			//Oh, no - they're on backwards
	//Remove them
	L->drop(s1);
	L->drop(s2);
```

```
        L->print();                 //Yep, they're gone
        //Add them again, this time in the right order.
        L->prepend(s2);
        L->prepend(s1);
        L->print();                 //All good
        //add the last number
        // Error 6: s6 is set at the front of the list
        L->append(s6);
        L->print();                 //Done!

        //Error 9: Doesn't delete L when you finish the program
        L = NULL;
        delete(L);



        return 0;
}
```