Software Project

Assignment 3 – Due on 9.6.2016 (23:55)

Introduction

In this assignment we will implement different modules that will be later used in the final project. You will practice:

- Your C programming skills,
- programming big systems , and
- efficiently debugging big systems using unit tests.

Modularity

Before heading to the implementation of the final project, we need to be able to work efficiently when developing a big system. One of the techniques we will use is called modular programming. In general, modular programming is a design technique in which we separate the functionality of a program into independent and interchangeable modules.

A module could be a data structure, a logger for handling message printing or a simple data type which encapsulates data to form a special meaning. In this assignment you will be guided on how to use this technique and you are expected to use it throughout the final project.

SPPoint

In this section you will implement a Point data type. Don't modify the header SPPoint.h

SPPoint.h

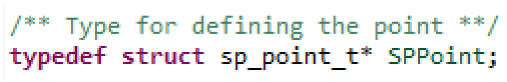
In Assignment 2, we represented a feature using an array of doubles. This representation is not clear and could result in many errors. For this reason, we will implement a Point module. The reason we have chosen the name **Point** and not **Feature** is because feature is specific to our system, and one may want to use our implementation of a point in different context. This is the important principle of *reusability*.

Before heading to the implementation of the point data type, we need to decide what functionality we are expecting when using a point data type (try to think of a list of functionalities before proceeding ahead).

First we need to be able to define a new data type. The right way to do this is by defining a header file (in our case it's called **SPPoint.h**) which will contain only the declaration of relevant functions and types that can be used by the user. As mentioned in Assignment 1, header files are considered a contract between the programmer and the user. The user cannot know anything about our implementation. The only thing the user cares about is the behavior of the code. The programmer should and must implement the code as stated in the documentation of the header file. Thus you should hide your implementation as much as needed.

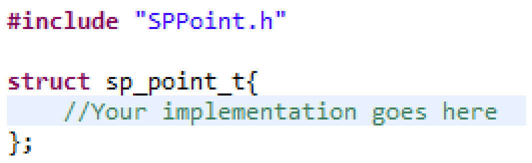
For this reason, the user should not know how we implemented the point. To hide this information from the user, we only need to define the data type (in our case struct sp_point_t) and the implementation should be in SPPoint.c.

If you look at the header file SPPoint.h you will find the following line:



```
/** Type for defining the point **/
typedef struct sp_point_t* SPPoint;
```

This line defines the data type **SPPoint**, which is a pointer to **struct sp_point_t;** note that the actual implementation of the struct should only be in SPPoint.c. That is, your source file should contain something similar to this:



```
#include "SPPoint.h"

struct sp_point_t{
    //Your implementation goes here
};
```

Doing so, will hide the implementation of **SPPoint**, hence the user know nothing about the internal structure of **SPPoint**.

The functionalities we are going to support are as follow:

1- Create – It's very common to implement a function which allocates a new data type.
2- Copy – It is very common to implement a copy function, which receives a point and creates a new copy (thus allocating new memory for the copy). The returned copy should contain the same information as the source point.
3- Destroy – Memory management is very important in C, and we need to be able to free resources in an efficient and clear way. Thus for every module, we need to support a destroy function, which frees all memory resources associated with a point.
4- Get Dimension – Returns the dimension of a point.
5- Get a coordinate – Returns a coordinate of a point.
6- Get Index – Returns an index of a point (read the header file for more information).
7- Calculate L2 Squared distance – Calculates the L2 distance between two points.

Please review the documentation in SPPoint.h.

Assertion

In many cases, the user of **SPPoint** could send invalid arguments to the functions provided by **SPPoint**. This may result in undefined behavior (usually segmentation fault). One way to prevent this is by assuming that the user is a good programmer and won't do such a horrible mistake.

Let us look at the documentation of the function **spPointL2SquaredDistance**:

```
/**
 * Calculates the L2-squared distance between p and q.
 * The L2-squared distance is defined as:
 * (p_1 - q_1)^2 + (p_2 - q_1)^2 + ... + (p_dim - q_dim)^2
 *
 * @param p - The first point
 * @param q - The second point
 * @assert p!=NULL AND q!=NULL AND dim(p) == dim(q)
 * @return
 * The L2-Squared distance between p and q
 */
double spPointL2SquaredDistance(SPPoint p, SPPoint q);
```

Notice that we have stated that the function assumes that p and q are valid:

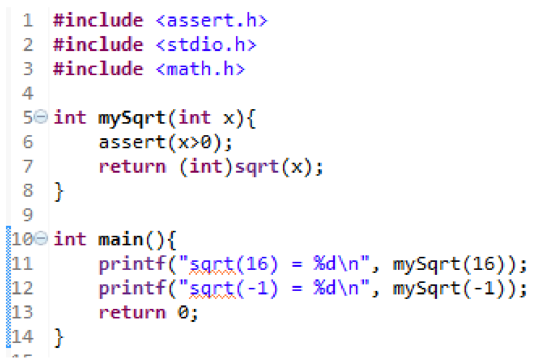
```
* @assert p!=NULL AND q!=NULL AND dim(p) == dim(q)
```

This way, we can implement the function under the assumption that whatever appears in the @assert line holds.

But what if the user is not that good of a programmer? What can we do (except from firing him)?

It is usually common to include the header file<assert.h>, which contains a macro assert which is used during the production process of a program. This macro receives a Boolean expression: if this expression is false at runtime, the program will terminate with a message stating that the assertion has failed.

For example:



```
#include <assert.h>
#include <stdio.h>
#include <math.h>

int mySqrt(int x){
    assert(x>0);
    return (int)sqrt(x);
}

int main(){
    printf("sqrt(16) = %d\n", mySqrt(16));
    printf("sqrt(-1) = %d\n", mySqrt(-1));
    return 0;
}
```

If we look at the above program we see that in line 6, the function **mySqrt** asserts that x>0. And when we try to run the main program, the assertion fails due to the call in line 12 thus receiving the following message:

```
Assertion failed: x>0, file ..\main.c, line 6

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

SPPoint.c

Write your own implementation in SPPoint.c. Use assertion in case the documentation of a function contains @assert lines.

Unit Testing

Before using **SPPoint**, you need to make sure you have implemented the Point data type properly and as expected from the interface. In programming principles, this is called Unit Testing. You are provided with a basic unit test for your Point implementation (sp_point_unit_test.c)—please review it and make sure you understand how to use the unit testing utility (unit_test_util.h). This unit test doesn't cover all cases and you need to extend the test to make sure that your implementation works properly. Your unit test should take the following consideration into account:

1- **Memory Leaks** – use valgrind (more on valgrind can be found on moodle) to check that all the memory resources which were dynamically allocated are freed.
2- **Functionality** – make sure all the functionalities of the point are well behaved and that you get the expected result.
3- **Edge Cases – m**ake sure your code works properly in edge cases.

SPLogger

Printing mechanism is useful for many purposes. We will implement a logger which will be used to print all messages during the run of our program. Four types of messages will be considered:

1- Error – messages that are used to indicate an error during run-time. Usually after error messages the program should terminate.
2- Warning – messages that are used to indicate a warning during run-time. Usually if a warning occurred the program will continue running.
3- Info- informational messages used to indicate the state of the system (for example –they may be used to inform the user about the progress of the program). Info messages should only contain reasonably significant information that will make sense to end users and system administrators.
4- Debug- messages are used for debugging purposes. These messages may contain anything the programmer sees fit.

The logger has 4 active levels. At each level only certain messages will be printed. The levels are:

1- Error level – In this level only error messages will be printed.
2- Warning level – In this level only warning and error messages will be printed.

3- Info- In this level all messages **EXCEPT** debug messages will be printed (i.e., error, warning and info).
4- Debug- In this level all messages will be printed.

For example, if the logger is initialized with SP_LOGGER_WARNING_ERROR_LEVEL. After executing the following lines:

```
spLoggerPrintErrorMsg("Error msg","..\unit_tests\sp_logger_unit_test.c",
"basicDebugTest",72);
spLoggerPrintWarningMsg("Warning msg","..\unit_tests\sp_logger_unit_test.c",
"basicDebugTest",74);
spLoggerPrintInfoMsg("Info msg”);
```

The resulting output should be:

```
---ERROR---
- file: ..\unit_tests\sp_logger_unit_test.c
- function: basicDebugTest
- line: 72
- message: Error msg
---WARNING---
- file: ..\unit_tests\sp_logger_unit_test.c
- function: basicDebugTest
- line: 74
- message: Warning msg
```

Notice that the info message wasn’t printed. For further information, please review the header file.

Message Mechanism

Unlike java, an exception mechanism, which sometimes can be used to inform the user if an error has occurred, is not supported in C. We will bypass this problem by defining a new type called SP_LOGGER_MSG. Each function in SPLogger, returns an enum of type SP_LOGGER_MSG which informs the user in case any error has occurred. Each name that appears in SP_LOGGER_MSG represents an error (or success) that may occur when the user calls SPLogger functions.

Carefully review the header file SPLogger.h for further information.

Special Macros

In order to pass the line number/function name/ filename to SPLoggerPrint functions, you can use special macros used by the preprocessor. These macros are as follow:

__FILE__ - The preprocessor replaces this macro with the filename in which it appeared.
__func__ - The preprocessor replaces this macro with the function name in which it appeared.
__LINE__ - The preprocessor replaces this macro with the line number in which it appeared.

SPLogger.c

Partial implementation of the logger is given. You will need to extend the implementation to support all functionalities provided in the header file SPLogger.h.

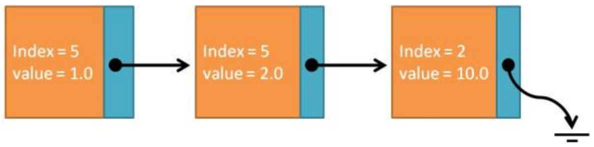
sp_logger_unit_test.c

You need to extend the unit test given in sp_logger_unit_test.c in order for your implementation to work as defined by the documentation in the header file.

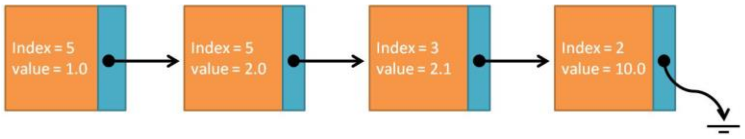
Bounded Priority Queue

A bounded priority queue is similar to a regular minimum priority queue, except that there is a fixed upper bound on the number of elements that can be stored in the BPQ. Whenever a new element is added to the queue, if the queue is at capacity, the element with the highest priority value is ejected from the queue.

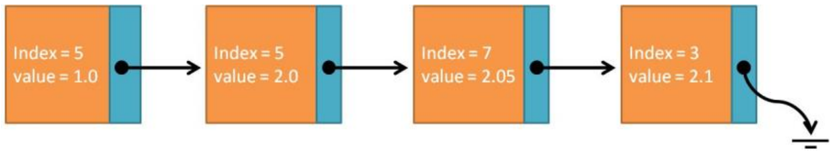
Elements in the queue will have an **int** index and a **double** value. Let us look at the example bellow. Suppose the maximum capacity of the bounded queue is **4**. Initially the queue looks like this:



If we insert (enqueue) the new element (index = 3, value 2.1) our queue will look like this:



Now the queue is at full capacity and adding the element (index = 7, value = 2.05) will result in the following queue:



If we try to add the new element (index = 100, value=20.0) it will not be added since the queue is full and the value 20.0 is bigger than the maximum value in the queue.

SPBPriorityQueue.h

Code documentation is critical when working in teams. In the documentation, you need to write the behavior of your code so that the user will know exactly what to expect when using your code. Your documentation should cover all edge cases and the implementation must follow the documentation details.

Complete the documentation in the header file SPBPriorityQueue, use SP_BPQUEUE_MSG to indicate if any errors has occurred. Note that you are not allowed to add/change any part of the header. Your task is to change only the comments to provide full documentation of the code.

Refer to SPLogger.h/SPListElement.h/SPList.h/SPPoint.h.

Your priority queue should have the following functionalities:

1- Create- creates an empty queue with a given maximum capacity
2- Copy- creates a copy of a given queue
3- Destroy- frees all memory allocation associated with the queue
4- Clear- removes all the elements in the queue
5- GetSize- returns the number of elements in the queue
6- GetMaxSize- returns the maximum capacity of the queue
7- Enqueue- Inserts a **NEW COPY (must be allocated)** element to the queue
8- Dequeue- removes the element with the **lowest value**
9- Peek- returns a **NEW COPY** of the element with **the lowest value**
10- PeekLast – returns a **NEW COPY** of the element with the highest value
11- MinValue- returns the minimum value in the queue
12- MaxValue- returns the maximum value in the queue
13- IsEmpty – returns true if the queue is empty
14- IsFull- returns true if the queue is full

Note that there are more efficient ways to implement a BPQ, but for the purpose of our project this should be enough.

SPBPriorityQueue.c

Implement the bounded priority queue. Your implementation should include all functionalities given in the header file. Your code should be well documented (insert comments if needed), and the behavior should follow the documentation you wrote in the previous section.

In order to implement the priority queue, use the linked list and linked list element given in the assignment zip file (SPList.h/SPList.c/SPListElement.h/SPListElement.c). You may assume that the list works properly.

sp_bpqueue_unit_test.c

Write a unit test using the utility header file as we did in previous sections. Your test should cover all edge cases and no memory leaks should occur when running the unit test.

Please make sure your unit test file name is **sp_bpqueue_unit_tests.c.**

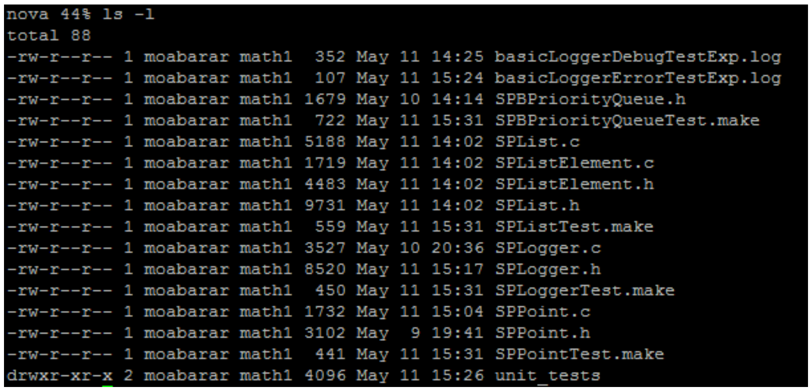
Make sure your unit test is in the unit_tests directory.

Testing On Nova

Copy your files to Nova, and put all the unit tests in the directory **unit_tests**.

In the assignment zip file you will find 4 makefiles (SPPointTest.make, SPLoggerTest.make, SPBPriorityQueueTest.make and SPListTest.make), each makefile builds the unit test corresponding to its name. The resulting executable name is simply the unit test name without the extension. For example the makefile **SPListTest.make** builds the unit test **sp_list_unit_test.c**, and the executable filename is **sp_list_unit_test**

Make sure all files are uploaded to nova (see below):



```
nova 44% ls -l
total 88
-rw-r--r-- 1 moabarar math1  352 May 11 14:25 basicLoggerDebugTestExp.log
-rw-r--r-- 1 moabarar math1  107 May 11 15:24 basicLoggerErrorTestExp.log
-rw-r--r-- 1 moabarar math1 1679 May 10 14:14 SPBPriorityQueue.h
-rw-r--r-- 1 moabarar math1  722 May 11 15:31 SPBPriorityQueueTest.make
-rw-r--r-- 1 moabarar math1 5188 May 11 14:02 SPList.c
-rw-r--r-- 1 moabarar math1 1719 May 11 14:02 SPListElement.c
-rw-r--r-- 1 moabarar math1 4483 May 11 14:02 SPListElement.h
-rw-r--r-- 1 moabarar math1 9731 May 11 14:02 SPList.h
-rw-r--r-- 1 moabarar math1  559 May 11 15:31 SPListTest.make
-rw-r--r-- 1 moabarar math1 3527 May 10 20:36 SPLogger.c
-rw-r--r-- 1 moabarar math1 8520 May 11 15:17 SPLogger.h
-rw-r--r-- 1 moabarar math1  450 May 11 15:31 SPLoggerTest.make
-rw-r--r-- 1 moabarar math1 1732 May 11 15:04 SPPoint.c
-rw-r--r-- 1 moabarar math1 3102 May  9 19:41 SPPoint.h
-rw-r--r-- 1 moabarar math1  441 May 11 15:31 SPPointTest.make
drwxr-xr-x 2 moabarar math1 4096 May 11 15:26 unit_tests
```

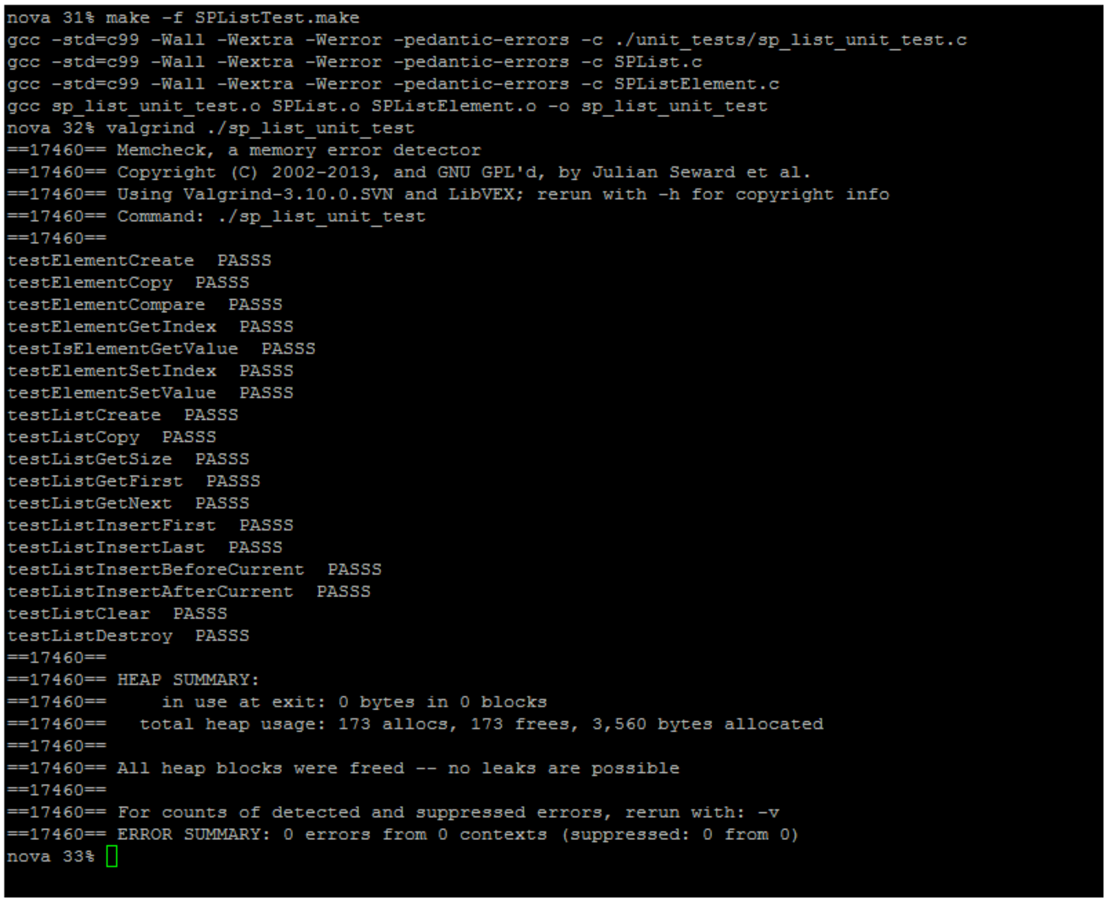
In order to run the make file you should enter the following command:

>> make -f <filename>

To clean previous build you should enter:

>> make -f <filename> clean

Check memory leaks using valgrind. See example below:



```
nova 31% make -f SPListTest.make
gcc -std=c99 -Wall -Wextra -Werror -pedantic-errors -c ./unit_tests/sp_list_unit_test.c
gcc -std=c99 -Wall -Wextra -Werror -pedantic-errors -c SPList.c
gcc -std=c99 -Wall -Wextra -Werror -pedantic-errors -c SPListElement.c
gcc sp_list_unit_test.o SPList.o SPListElement.o -o sp_list_unit_test
nova 32% valgrind ./sp_list_unit_test
==17460== Memcheck, a memory error detector
==17460== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==17460== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==17460== Command: ./sp_list_unit_test
==17460==
testElementCreate  PASSS
testElementCopy  PASSS
testElementCompare  PASSS
testElementGetIndex  PASSS
testIsElementGetValue  PASSS
testElementSetIndex  PASSS
testElementSetValue  PASSS
testListCreate  PASSS
testListCopy  PASSS
testListGetSize  PASSS
testListGetFirst  PASSS
testListGetNext  PASSS
testListInsertFirst  PASSS
testListInsertLast  PASSS
testListInsertBeforeCurrent  PASSS
testListInsertAfterCurrent  PASSS
testListClear  PASSS
testListDestroy  PASSS
==17460==
==17460== HEAP SUMMARY:
==17460==     in use at exit: 0 bytes in 0 blocks
==17460==   total heap usage: 173 allocs, 173 frees, 3,560 bytes allocated
==17460==
==17460== All heap blocks were freed -- no leaks are possible
==17460==
==17460== For counts of detected and suppressed errors, rerun with: -v
==17460== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
nova 33% 
```

Submission

Please submit a zip file with the name **id1_id2_assignment3.zip,** where id1 and id2 are the ids of the partners. The zipped file must contain the following files:

- **unit_tests** – The directory which contains the all unit tests (make sure you extend the unit tests)
  - Source files: sp_bpqueue_unit_test.c, sp_logger_unit_test.c, sp_point_unit_test.c
  - Header files: unit_test_util.h
- **The makefiles provided in the zip file -** SPBPriorityQueueTest.make, SPLoggerTest.make, SPPointTest.make **(Don't modify)**
- **Source files –** SPBPriorityQueue.c, SPList.c, SPListElement.c, SPLogger.c and SPPoint.c
- **Header files–** SPBPriorityQueue.h, SPList.h, SPListElement.h, SPLogger.h and SPPoint.h
- **Partners.txt –** This file must contain the full name, id and moodle username for both partners. Please follow the pattern in the assignment files. **(Do not change the pattern)**

Remarks

- For any question regarding the assignment, please don't hesitate to contact Moab Arar by mail: moabarar@mail.tau.ac.il.
- Borrowing from others' work is not acceptable and may bear severe consequences.

Good luck