CS2110
Summer 2016
Homework 9

Kevin Simon

Rules and Regulations

General Rules

1. Starting with the assembly homeworks, Any code you write (if any) must be clearly commented and the comments must be meaningful. You should comment your code in terms of the algorithm you are implementing we all know what the line of code does.
2. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. This means that (in the case of demos) you should come prepared to explain to the TA how any piece of code you submitted works, even if you copied it from the book or read about it on the internet.
3. Please read the assignment in its entirety before asking questions.
4. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.
5. If you find any problems with the assignment it would be greatly appreciated if you reported them to the author (which can be found at the top of the assignment). Announcements will be posted if the assignment changes.

Submission Conventions

1. All files you submit for assignments in this course should have your name at the top of the file as a comment for any source code file, and somewhere in the file, near the top, for other files unless otherwise noted.

2. When preparing your submission you may either submit the files individually to T-Square or you may submit an archive (zip or tar.gz only please) of the files (preferred). You can create an archive by right clicking on files and selecting the appropriate compress option on your system.

3. If you choose to submit an archive please don't zip up a folder with the files, only submit an archive of the files we want. (See **Deliverables**).

4. Do not submit compiled files that is .class files for Java code and .o files for C code. Only submit the files we ask for in the assignment.

5. Do not submit links to files. We will not grade assignments submitted this way as it is easy to change the files after the submission period ends.

Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know ***IN ADVANCE*** of the due time supplying documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.
2. You are also responsible for ensuring that what you turned in is what you meant to turn in. After submitting you should be sure to download your submission into a brand new folder and test if it works. No excuses if you submit the wrong files, what you turn in is what we grade. In addition, your assignment must be turned in via T-Square. When you submit the assignment you should get an email from T-Square telling you that you submitted the assignment. If you do not get this email that means that you did not complete the submission process correctly. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over T-Square.
3. There is a 6-hour grace period added to all assignments. You may submit your assignment without penalty up until 11:55PM, or with 25% penalty up until 5:55AM. *So what you should take from this is not to start assignments on the last day and plan to submit right at 11:54AM.* You alone are responsible for submitting your homework before the grace period begins or ends; neither T-Square, nor your flaky internet are to blame if you are unable to submit because you banked on your computer working up until 11:54PM. The penalty for submitting during the grace period (25%) or after (no credit) is non-negotiable.

Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class.

Quizzes, timed labs and the final examination are individual work.

Homework assignments are collaborative, In addition many if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using electronic computer programs to find evidence of unauthorized collaboration.

What is unauthorized collaboration? Each individual programming assignment should be coded by you. You may work with others, but each student should be turning in their own version of the assignment. Submissions that are essentially identical will receive a zero and will be sent to the Dean of Students' Office of Academic Integrity. Submissions that are copies that have been superficially modified to conceal that they are copies are also considered unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student via electronic means. This includes simply e-mailing it to them so they can look at it. If you supply an electronic copy of your homework to another student and they are charged with copying, you will also be charged. This includes storing your code on any site which would allow other parties to obtain your code such as but not limited to public repositories (Github), pastebin, etc. If you would like to use version control, use github.gatech.edu

Is collaboration allowed?

Collaboration is allowed on a high level, meaning that you may discuss design points and concepts relevant to the homework with your peers, as well as help each other debug code. What you shouldn't be doing, however, is paired programming where you collaborate with each other on a low level. Furthermore, sending an electronic copy of your homework to another student for them to look at and figure out what is wrong with their code is not an acceptable way to help them, and it is often the case that the recipient will simply modify the code and submit it as their own.

Overview

NOTICE: please read the WHOLE assignment before you begin coding.

You will be writing a generic double-ended queue (deque) library whose underlying structure is implemented with doubly-linked nodes. The structure will contain a head and tail pointer, which are the nodes at the ends of the deque. You may not use a "sentinel" or dummy node (a node with no data at the ends). Each node will have valid data.

https://en.wikipedia.org/wiki/Double-ended_queue

What to Code

The first part of this assignment is implementing the library. We have provided you two files: *deque.c* and *deque.h*. **Please review the definitions in deque.h, as you may see these types of statements on the final.** We have also provided prototypes for all of the functions you will be implementing. You may not change the definitions in this file. If you change anything in this file, then it won't compile when it is graded and you will get a 0. We have provided a failsafe, which is that **the Makefile will not function if you change even a single character in the deque.h file**. If it has been modified, you will need to re-download it from Canvas (or re-extract it from the archive)

Next, turn your attention to *deque.c*. We've provided you with function headers to all the library functions we would like you to implement. Implement these functions!

Make sure you read the comments before asking any questions. The comments tell you exactly what is expected of each function.

Function Pointers

If you'll look closely at the files we've provided you, some functions take in pointers to functions as parameters. This is no mistake (so don't change what we've given you!), this is where the concept
of function pointers comes into play. You will be using what you've learned about function pointers in class to manage this portion of the homework. To briefly put into perspective what you should be doing, take a look at this line in *deque.c*:

void traverse(deque *d, deque_op do_func) {

This function is supposed to do something with the data at each node in the deque. However, the library has no idea what the user wants to do to the data, so it instead takes in a function pointer, which is stored in the parameter do_func. This is a pointer to a function that the user themselves wrote, defining what to do with each piece of data in the

deque. The traverse function should call do_func with each piece of data contained in the deque.

do_func should be called on a node's data and not the node itself. The user of your deque library shouldn't have to deal with or even have knowledge of the node implementations in the deque, so do_func is written by the user to run on the node's data.

Part 1.5 – Design Issues

The design of this deque library is such that the person using your library does not have to deal with the details of the implementation of your library (i.e. the node struct used to implement the deque). None of these functions return a node nor do any of these functions take in a node.

It is your responsibility to create the nodes and add them into the deque yourself, not the user. For example, to use the deque library, I can decide that I want a deque of person structs. I can then define these functions to work for a person struct (examples in test.c).

If I want to print the persons' data, I would write a *print_person* function that matches the function pointer type *deque_op* from deque.h. If I wanted to print them all, I would call the *traverse* function passing in my *print_person* function as a parameter.

If the user wants to destroy the deque, then they will write their own *free_person* function that also matches *deque_op*. When the user is done with the deque, they will call *empty_deque* which removes all of the person structs from the deque and frees the nodes that contained pointers to the person structs. Finally, you must free the deque structure yourself (see the end of test.c) so that no memory is leaked by your function.

Part 2

Install valgrind and gdb:

```
sudo apt-get install valgrind gdb
```

Data structures in C must be tested with all those pointers flying everywhere, and it's hard to get them right the first time. For this reason, you should thoroughly test your code.

We have provided you with a file called *test.c* with which to test your code. It contains multiple test cases, all of which create, destroy, add to and query a deque. The given test cases are not comprehensive, and are rather just an example of how you can try to test your code. Write more test cases!

Printing out the contents of your structures can't catch all logical and memory errors, so we also require you run your code through valgrind. If you need help with debugging, there is a C debugger called gdb that will help point out problems. A tutorial with examples of using these debuggers is included in this assignment in the "debugging" directory. We certainly will be checking for memory leaks by using valgrind, so if you learn how to use it, you'll catch any memory errors before we do.

Here is a small tutorial on valgrind: http://cs.ecs.baylor.edu/~donahoo/tools/valgrind/

Your code must not crash, run infinitely, nor generate memory leaks/errors. Any test we run for which valgrind reports a memory leak or memory error will receive half credit.

Running your code

We have provided a Makefile for this assignment that will build your project. Here are the commands you should be using with this Makefile:

1. To run the tests in *test.c*: **make run-test**
2. To debug your code using gdb: **make run-gdb**
3. To run your code with valgrind: **make run-valgrind**

Debugging

If your code generates a segmentation fault then you should first run gdb on the debug version of your executable before asking questions. (We will not look at your code to find your segmentation fault. This is why gdb was written to help you find your segmentation fault yourself.).

Here are some tutorials on gdb

http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html

http://heather.cs.ucdavis.edu/~matloff/debug.html

http://www.delorie.com/gnu/docs/gdb/gdb_toc.html

Getting good at debugging will make your life with C that much easier.

In addition to the debugging folder (which you should definitely give a read), I have provided some useful debugging macros for your use:

1) IF_DEBUG(statement)

Runs statement if compiled in debug mode (run-debug / run-gdb /run-valgrind)

Example usage:

```
IF_DEBUG(printf("HE
LLO 2110"));
IF_DEBUG(
{
        x = 3;
        x++;
        x = 7;
        x++;
});
```

2) DEBUG_PRINT(string)

Prints out string (in red to distinguish it) if compiled in debug mode
This will also print out the file and line the print occurs printing goes to stderr
DEBUG_PRINT("Hello 2110");

3) DEBUG_ASSERT(expression)

If compiled in debug mode if expression is false
The program terminates with a message and exits it also tells you where the assertion failed.

```
/* This will fail since 8^8 is 0
which is false*/
DEBUG_ASSERT(8 ^ 8);

/* This will pass since 8^7 is 15
which is true */
DEBUG_ASSERT(8 ^ 7);
```

You may alternatively use assert, but you must #include <assert.h>

Remember

1. Write the contents of all of the functions in *deque.c*. Be sure to pay attention to special cases. Your code should never crash, run infinitely, nor should it leak.
2. Write more test cases in *test.c*. The provided test cases are not exhaustive.
3. Do not change any of the function prototypes in *deque.h*, because it will cause your submission not to compile when we test it. Non-compiling code is an automatic zero.

Deliverables

Submit ONLY the following file, which can be generated in the deque directory by using the command `make submit`:

- deque_submission.tar.gz

Your files MUST compile with the required 2110 flags, which are: -std=c99 -pedantic -Wall -Werror -Wextra

Remember, non-compiling homeworks receive a zero. Make sure you turn in everything you need! To be safe, redownload your submission into a new directory and try compiling it. Also, if your submission does not compile for us because you changed something in Makefile or deque.h, that will also get you a zero.