CT331 Assignment 1

Procedural Programming with C

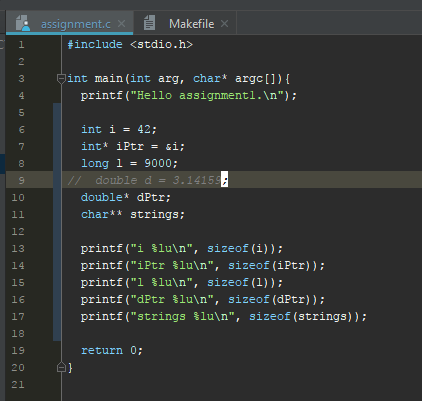
Andrew Reid East

16280042

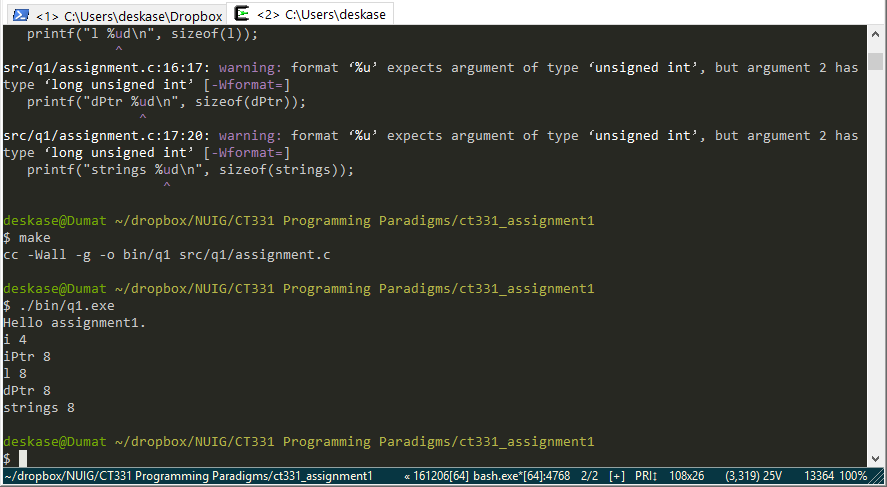
<https://github.com/reideast/ct331_assignment1>

# Question 1

## Code Editor



## Command Line Output



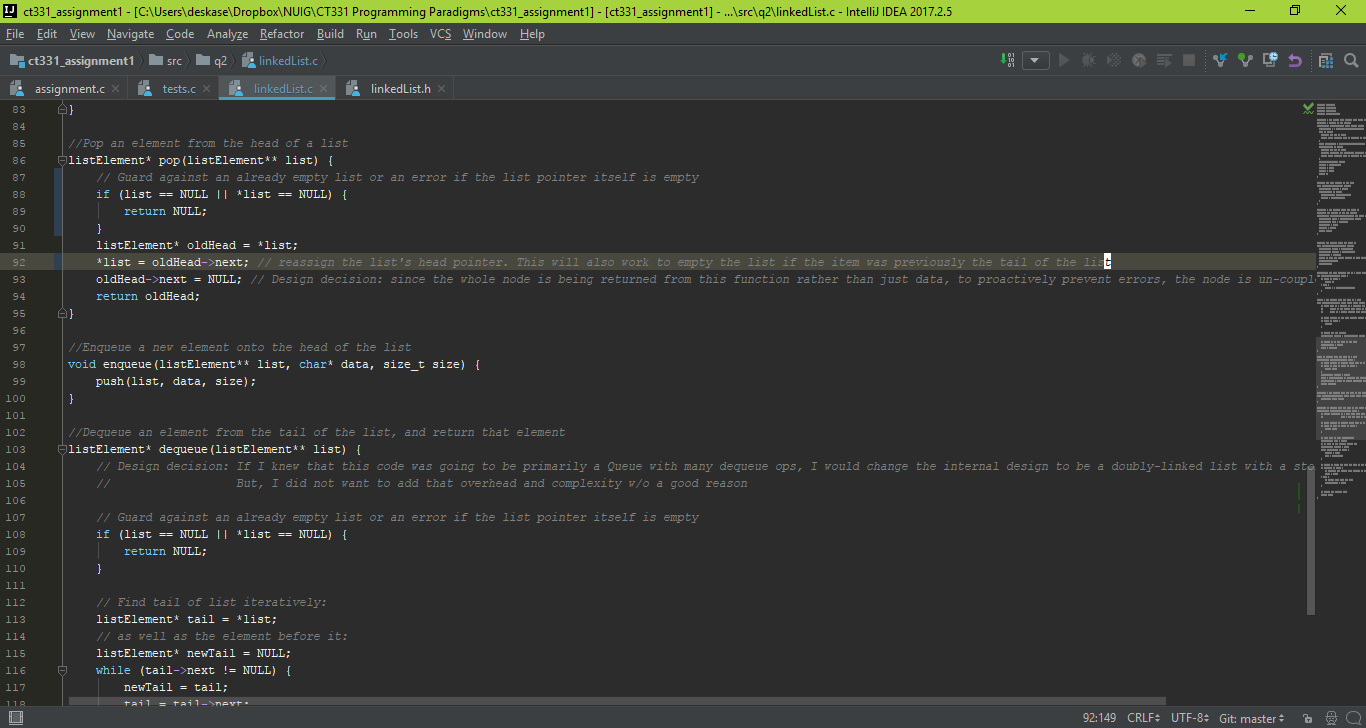
## Comments

The size of the stack data variables, the integer and the long are as expected: an int is 4 bytes, 32 bits, and a long is twice that much, 8 bytes. Four bytes is the most common integer size for modern compilers, and I expected to see a long be twice as much storage (although it is not required to be larger to be compliant).

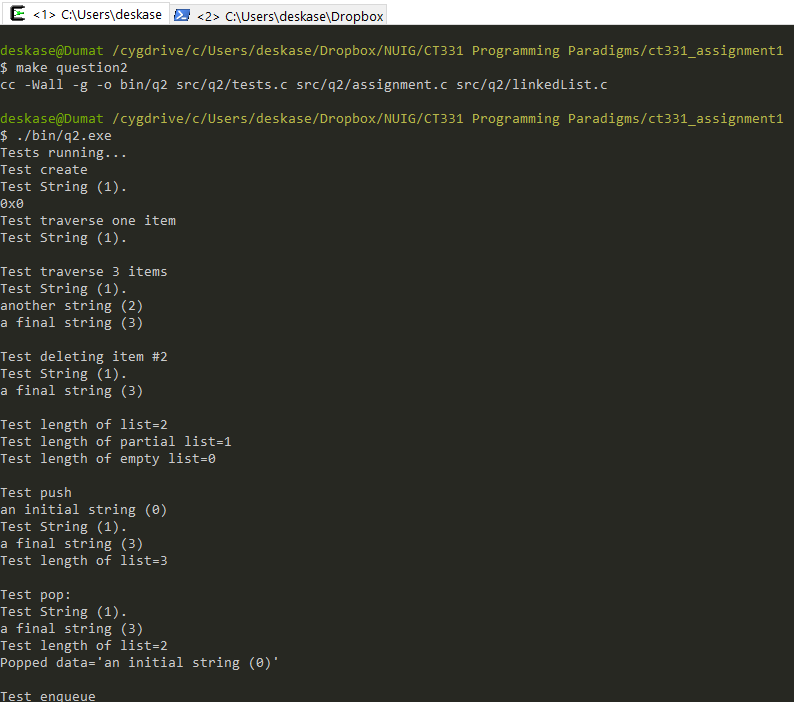
The amount of memory needed to store all of the different pointer types ended up being identical, even though the storage needed for the data they may point to is different. Since this was compiled on 64-bit built Cygwin on a 64-bit Windows build on an x64 Intel processer, the 8-byte/64-bit pointer sizes were as expected. To further experiment, I installed a 32-bit build of Cygwin, (had to change the printf format specifiers for 32-bit gcc’s requirements) and then saw that in this case, 4 bytes were required for the pointers.

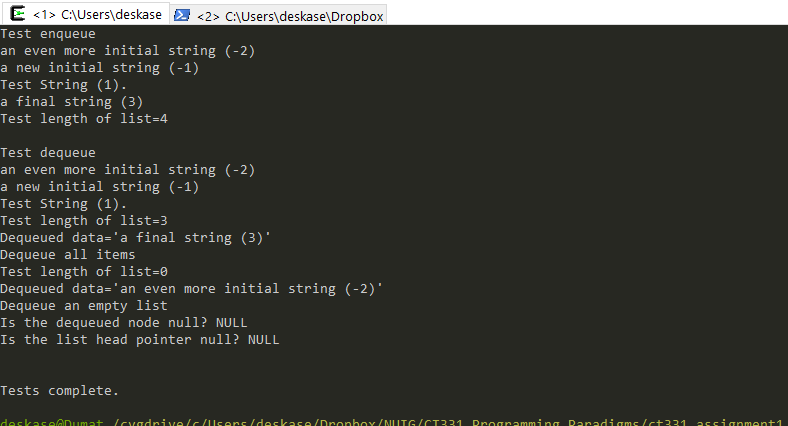
# Question 2

## Code Editor



## Command Line Output



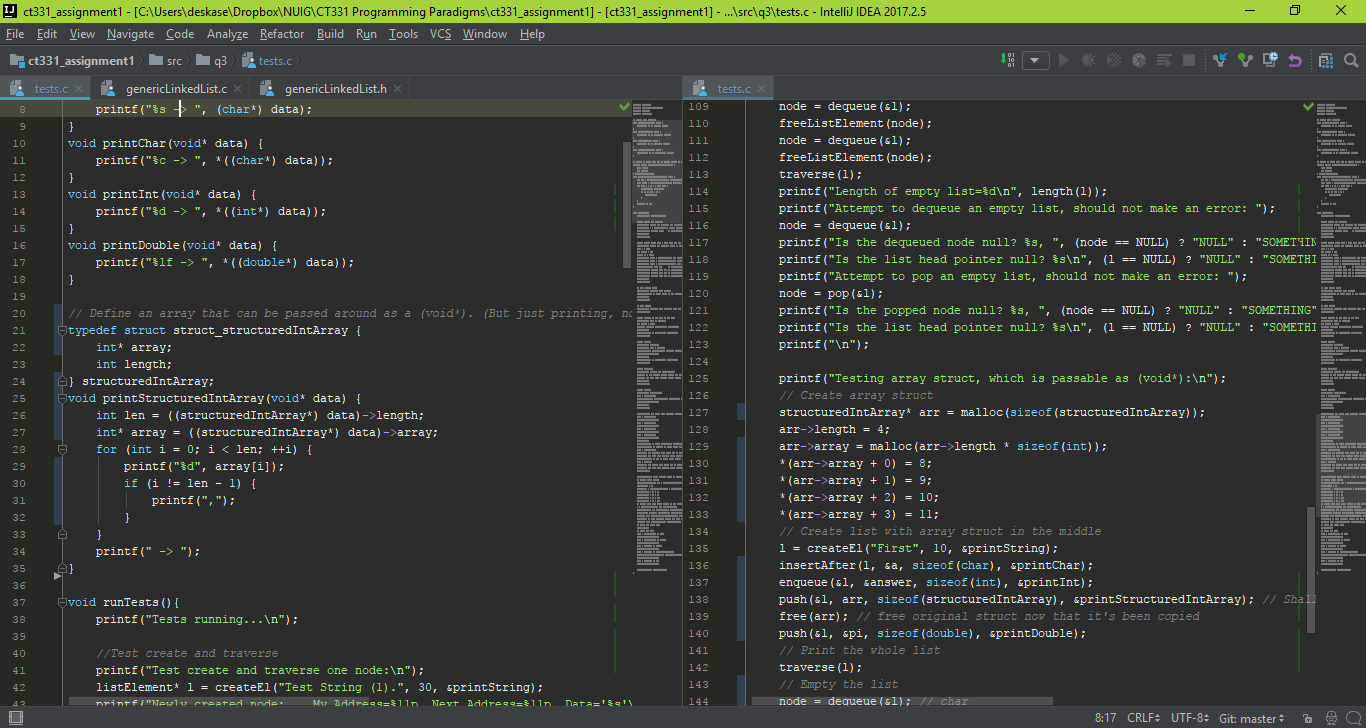


## Comments

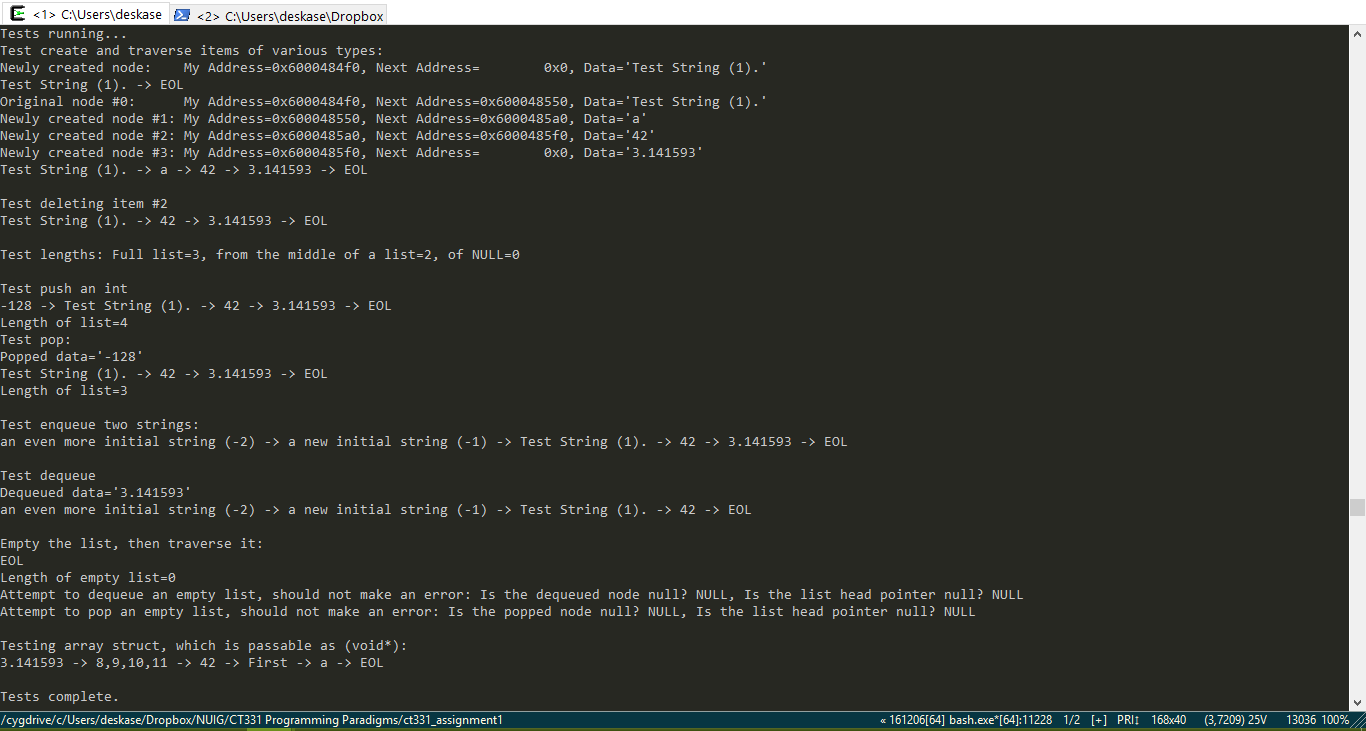
It was particularly interesting to figure out why push(), pop(), and enqueue() needed a double-star pointer to pass the head of the list as an argument, as I have not seen that idiom before. I briefly questioned to myself how push() could be implemented without returning a reference to the new head of the list, then I was confused when I saw the \*\*. Finally, after a complier error on dereferencing the pointer, I realized the connection, and knew the reasoning how to make the function work properly.

# Question 3

## Code Editor



## Command Line Output



## Comments

After I finished adding the function pointers, I spent the most time for this problem attempting to implement adding an array to the linked list—without modifying the linked list itself. This necessitated creating a more advanced printer-function, which would still take a void\* holding the array. Since C arrays do not store their own length, I had to use a struct storing both the array and a length variable (taking inspiration from Java). Unfortunately, I was not able to make this functionality work for generic arrays, and just stuck with implementing it for an int array.

# Question 4

When traversing a singly-linked list from tail to head, there is no way to start at the tail. Even if a tail pointer is maintained, the program still cannot move backwards through the nodes. Iterative methods to traverse the list forward before going backward again creates the problem of how to store each node as it is passed until the tail is reached. If the length of the list is known, each piece of data or node could be stored in an array as it is passed, then the array iterated backwards to get a final result. The memory needs would be the size of an array of pointers to nodes, as well as counter variables. If the length is not known, an array could be dynamically allocated as each node is passed (a process and memory-manager intensive proposition), or one could build a second linked list with data pointers to each element, next pointers reverse. This would take as much memory as the first list (just not for the data, which would just be pointed too), as well as counter variables for the loop.

A much less complex way to traverse a linked list from tail to head would be with a recursive method. Each node visited would first recursively attempt call upon its next neighbour and after would visit itself. When the tail is reach, the visit operations would then be executed off the stack in reverse order. This would require no extra data to be stored in memory, such as an array, but each element would be a function call that would need to be stored as a stack frame. The actual memory needed for this could potentially be larger than an array or second linked list needed for the iterative method, and would definitely be at least one memory “unit” for each item. Further, since the recursive call happens *before* the function is completed, it’s likely the compiler could not optimize the call stack as a tail recursion. Because of the limits on the height of a call stack, the recursive operation could also fail even when there is plenty of main memory free. However, despite possibly needing more memory than an iterative method, this method may still be useful because it may execute faster.

To improve the memory usage of a reversed traversal, the program would have to store a “previous” pointer within each node (doubly-linked list) as well as maintaining a pointer to the tail node of the list. This would afford the ability to traverse the list from head to tail iteratively, with the only memory requirement being a pointer to the current node. The recursive method could also be used tail to head, and since the recursive function call would then be the final statement of the subroutine, the compiler could likely optimise the stack frame as a tail call, and not need nearly as much space for each frame. However, this change to the data structure would require another pointer stored for every node. As observed in Question 1, on this 64-bit system, those pointers would require 8 bytes each—this could increase the base requirements for the list by several times. If the program will never (or not very often) access the data backwards, this will be wasted memory. As with most things in programming, there is a trade-off for each choice.