

# CT331 Assignment 1

## Procedural Programming with C

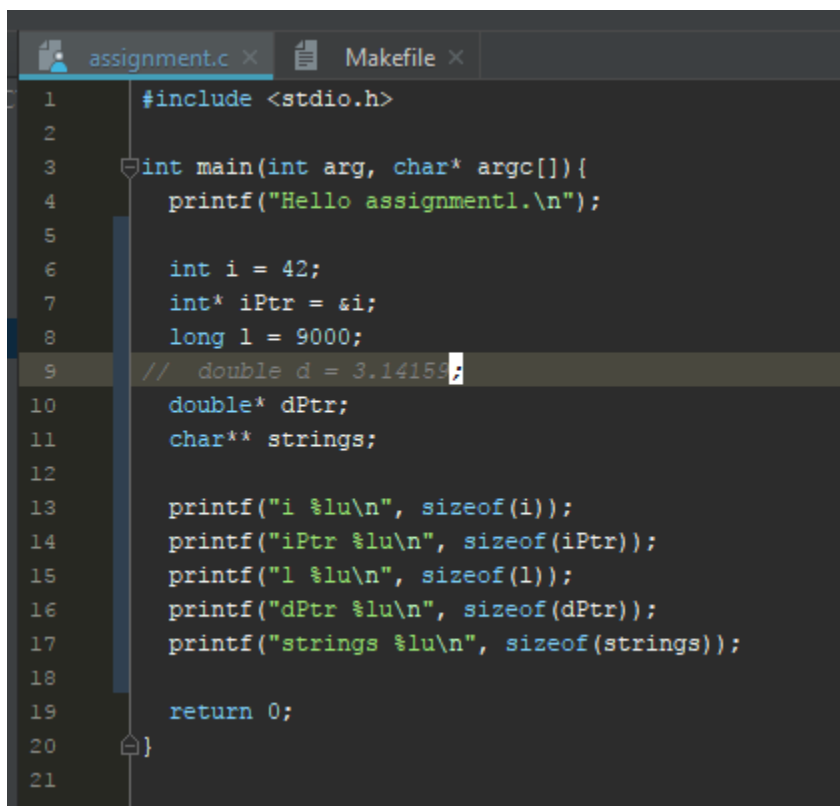
Andrew Reid East

16280042

[https://github.com/reideast/ct331\\_assignment1](https://github.com/reideast/ct331_assignment1)

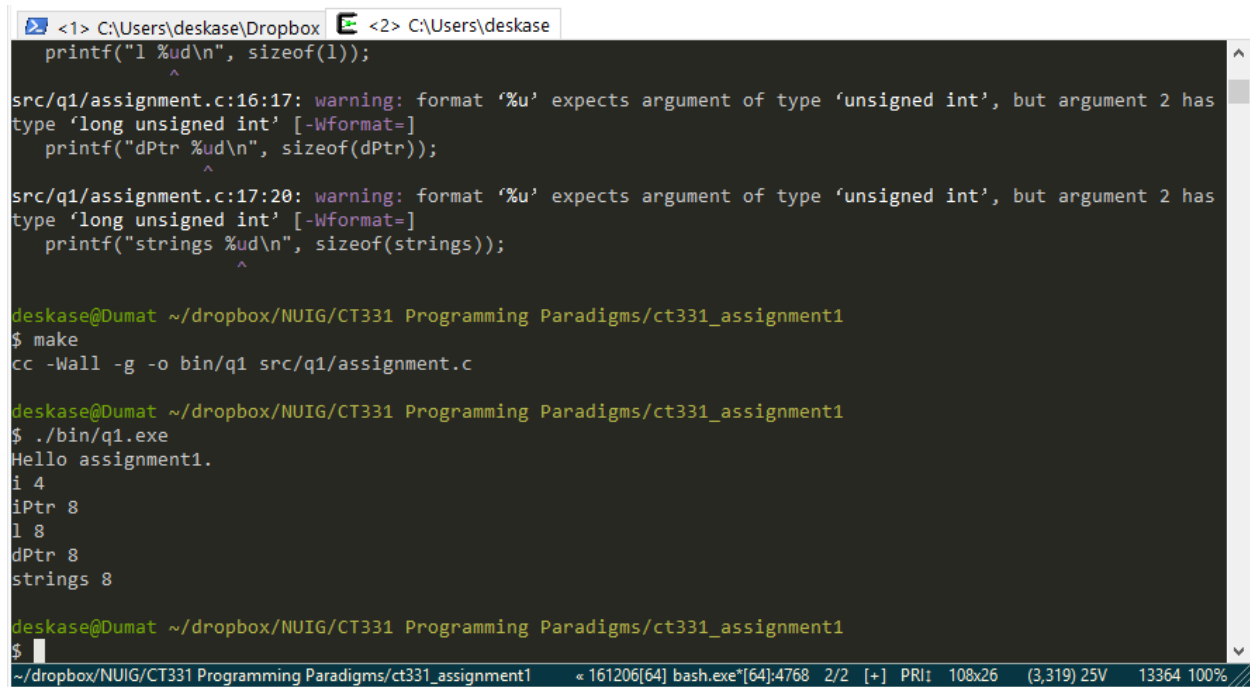
### Question 1

#### Code Editor



```
assignment.c x Makefile x
1  #include <stdio.h>
2
3  int main(int arg, char* argc[]){
4      printf("Hello assignment1.\n");
5
6      int i = 42;
7      int* iPtr = &i;
8      long l = 9000;
9      // double d = 3.14159;
10     double* dPtr;
11     char** strings;
12
13     printf("i %lu\n", sizeof(i));
14     printf("iPtr %lu\n", sizeof(iPtr));
15     printf("l %lu\n", sizeof(l));
16     printf("dPtr %lu\n", sizeof(dPtr));
17     printf("strings %lu\n", sizeof(strings));
18
19     return 0;
20 }
21
```

## Command Line Output



```
<1> C:\Users\deskase\Dropbox <2> C:\Users\deskase
printf("l %ud\n", sizeof(l));
src/q1/assignment.c:16:17: warning: format '%u' expects argument of type 'unsigned int', but argument 2 has
type 'long unsigned int' [-Wformat=]
printf("dPtr %ud\n", sizeof(dPtr));
src/q1/assignment.c:17:20: warning: format '%u' expects argument of type 'unsigned int', but argument 2 has
type 'long unsigned int' [-Wformat=]
printf("strings %ud\n", sizeof(strings));

deskase@Dumat ~/dropbox/NUIG/CT331 Programming Paradigms/ct331_assignment1
$ make
cc -Wall -g -o bin/q1 src/q1/assignment.c

deskase@Dumat ~/dropbox/NUIG/CT331 Programming Paradigms/ct331_assignment1
$ ./bin/q1.exe
Hello assignment1.
i 4
iPtr 8
l 8
dPtr 8
strings 8

deskase@Dumat ~/dropbox/NUIG/CT331 Programming Paradigms/ct331_assignment1
$
```

## 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 processor, 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

```
ct331_assignment1 - [C:\Users\deskase\Dropbox\NUIG\CT331 Programming Paradigms\ct331_assignment1] - [ct331_assignment1] - ...src\q2\LinkedList.c - IntelliJ IDEA 2017.2.5
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help

ct331_assignment1 src q2 q2\LinkedList.c
assignment.c x tests.c x q2\LinkedList.c x linkedList.c x

83 }
84
85 //Pop an element from the head of a list
86 listElement* pop(listElement** list) {
87     // Guard against an already empty list or an error if the list pointer itself is empty
88     if (list == NULL || *list == NULL) {
89         return NULL;
90     }
91     listElement* oldHead = *list;
92     *list = oldHead->next; // reassign the list's head pointer. This will also work to empty the list if the item was previously the tail of the list
93     oldHead->next = NULL; // Design decision: since the whole node is being returned from this function rather than just data, to proactively prevent errors, the node is un-coupled
94     return oldHead;
95 }
96
97 //Enqueue a new element onto the head of the list
98 void enqueue(listElement** list, char* data, size_t size) {
99     push(list, data, size);
100 }
101
102 //Dequeue an element from the tail of the list, and return that element
103 listElement* dequeue(listElement** list) {
104     // Design decision: If I knew that this code was going to be primarily a Queue with many dequeue ops, I would change the internal design to be a doubly-linked list with a stack
105     // But, I did not want to add that overhead and complexity v/o a good reason
106
107     // Guard against an already empty list or an error if the list pointer itself is empty
108     if (list == NULL || *list == NULL) {
109         return NULL;
110     }
111
112     // Find tail of list iteratively:
113     listElement* tail = *list;
114     // as well as the element before it:
115     listElement* newTail = NULL;
116     while (tail->next != NULL) {
117         newTail = tail;
118         tail = tail->next;
119     }
120     return newTail;
121 }
```

### Command Line Output

```
<1> C:\Users\deskase <2> C:\Users\deskase\Dropbox
deskase@Dumat /cygdrive/c/Users/deskase/Dropbox/NUIG/CT331 Programming Paradigms/ct331_assignment1
$ make question2
cc -Wall -g -o bin/q2 src/q2/tests.c src/q2/assignment.c src/q2/LinkedList.c

deskase@Dumat /cygdrive/c/Users/deskase/Dropbox/NUIG/CT331 Programming Paradigms/ct331_assignment1
$ ./bin/q2.exe
Tests running...
Test create
Test String (1).
0x0
Test traverse one item
Test String (1).

Test traverse 3 items
Test String (1).
another string (2)
a final string (3)

Test deleting item #2
Test String (1).
a final string (3)

Test length of list=2
Test length of partial list=1
Test length of empty list=0

Test push
an initial string (0)
Test String (1).
a final string (3)
Test length of list=3

Test pop:
Test String (1).
a final string (3)
Test length of list=2
Popped data='an initial string (0)'

Test enqueue
```

```
<1> C:\Users\deskase <2> C:\Users\deskase\Dropbox
Test enqueue
an even more initial string (-2)
a new initial string (-1)
Test String (1).
a final string (3)
Test length of list=4

Test dequeue
an even more initial string (-2)
a new initial string (-1)
Test String (1).
Test length of list=3
Dequeued data='a final string (3)'
Dequeue all items
Test length of list=0
Dequeued data='an even more initial string (-2)'
Dequeue an empty list
Is the dequeued node null? NULL
Is the list head pointer null? NULL

Tests complete.

deskase@Dumst: /cydrive/c:/Users/deskase/Dropbox/NUIG/CT331 Programming Paradigms/ct331_assignment1
```

## 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 compiler error on dereferencing the pointer, I realized the connection, and knew the reasoning how to make the function work properly.

## Question 3

### Code Editor

```
ct331_assignment1 - [C:\Users\deskase\Dropbox\NUIG\CT331 Programming Paradigms\ct331_assignment1] - [ct331_assignment1] - ...src\q3\tests.c - IntelliJ IDEA 2017.2.5
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help

ct331_assignment1 src q3 tests.c
tests.c genenclinkedlist.c genenclinkedlist.h tests.c
8 printf("%s\n", (char*) data);
9
10 void printChar(void* data) {
11     printf("%c -> ", *(char*) data);
12 }
13 void printInt(void* data) {
14     printf("%d -> ", *(int*) data);
15 }
16 void printDouble(void* data) {
17     printf("%lf -> ", *(double*) data);
18 }
19
20 // Define an array that can be passed around as a (void*). (But just printing, n
21 typedef struct structuredIntArray {
22     int* array;
23     int length;
24 } structuredIntArray;
25 void printStructuredIntArray(void* data) {
26     int len = ((structuredIntArray*) data)->length;
27     int* array = ((structuredIntArray*) data)->array;
28     for (int i = 0; i < len; ++i) {
29         printf("%d", array[i]);
30         if (i != len - 1) {
31             printf(",");
32         }
33     }
34     printf("\n -> ");
35 }
36
37 void runTests() {
38     printf("Tests running...\n");
39
40     //Test create and traverse
41     printf("Test create and traverse one node:\n");
42     listElement* l1 = createEl("Test String (1).", 30, sprintfString);
43     printf("Created node: %s\n", l1->data->str);
44
45     node = dequeue(&l1);
46     freeListElement(node);
47     node = dequeue(&l1);
48     freeListElement(node);
49     traverse(l1);
50     printf("Length of empty list=%d\n", length(l1));
51     printf("Attempt to dequeue an empty list, should not make an error: ");
52     node = dequeue(&l1);
53     printf("Is the dequeued node null? %s, ", (node == NULL) ? "NULL" : "SOMETHING");
54     printf("Is the list head pointer null? %s\n", (l1 == NULL) ? "NULL" : "SOMETHING");
55     printf("Attempt to pop an empty list, should not make an error: ");
56     node = pop(&l1);
57     printf("Is the popped node null? %s, ", (node == NULL) ? "NULL" : "SOMETHING");
58     printf("Is the list head pointer null? %s\n", (l1 == NULL) ? "NULL" : "SOMETHING");
59     printf("\n");
60
61     printf("Testing array struct, which is passable as (void*):\n");
62     // Create array struct
63     structuredIntArray* arr = malloc(sizeof(structuredIntArray));
64     arr->length = 4;
65     arr->array = malloc(arr->length * sizeof(int));
66     *(arr->array + 0) = 8;
67     *(arr->array + 1) = 9;
68     *(arr->array + 2) = 10;
69     *(arr->array + 3) = 11;
70
71     // Create list with array struct in the middle
72     l1 = createEl("First", 10, sprintfString);
73     insertAfter(l1, arr, sizeof(char), sprintfChar);
74     enqueue(&l1, answer, sizeof(int), sprintfInt);
75     push(&l1, arr, sizeof(structuredIntArray), printStructuredIntArray); // Shall
76     free(arr); // free original struct now that it's been copied
77     push(&l1, spl, sizeof(double), sprintfDouble);
78     // Print the whole list
79     traverse(l1);
80     // Empty the list
81     node = dequeue(&l1); // char
```

## Command Line Output

```
<1> C:\Users\deskase <2> C:\Users\deskase\Dropbox
Tests running...
Test create and traverse items of various types:
Newly created node: My Address=0x6000484f0, Next Address= 0x0, Data='Test String (1).'
Test String (1). -> EOL
Original node #0: My Address=0x6000484f0, Next Address=0x600048550, Data='Test String (1).'
Newly created node #1: My Address=0x600048550, Next Address=0x6000485a0, Data='a'
Newly created node #2: My Address=0x6000485a0, Next Address=0x6000485f0, Data='42'
Newly created node #3: My Address=0x6000485f0, Next Address= 0x0, Data='3.141593'
Test String (1). -> a -> 42 -> 3.141593 -> EOL

Test deleting item #2
Test String (1). -> 42 -> 3.141593 -> EOL

Test lengths: Full list=3, from the middle of a list=2, of NULL=0

Test push an int
-128 -> Test String (1). -> 42 -> 3.141593 -> EOL
Length of list=4
Test pop:
Popped data='-128'
Test String (1). -> 42 -> 3.141593 -> EOL
Length of list=3

Test enqueue two strings:
an even more initial string (-2) -> a new initial string (-1) -> Test String (1). -> 42 -> 3.141593 -> EOL

Test dequeue
Dequeued data='3.141593'
an even more initial string (-2) -> a new initial string (-1) -> Test String (1). -> 42 -> EOL

Empty the list, then traverse it:
EOL
Length of empty list=0
Attempt to dequeue an empty list, should not make an error: Is the dequeued node null? NULL, Is the list head pointer null? NULL
Attempt to pop an empty list, should not make an error: Is the popped node null? NULL, Is the list head pointer null? NULL

Testing array struct, which is passable as (void*):
3.141593 -> 8,9,10,11 -> 42 -> First -> a -> EOL

Tests complete.
/cygdrive/c/Users/deskase/Dropbox/NUIG/CT331 Programming Paradigms/ct331_assignment1 « 161206[64] bash.exe*[64]:112
```

## 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.