Name: Andrew Hayes
E-mail: a.hayes18@universityofgalway.ie
ID: 21321503

CT331

2023–10–21

Assignment 1: Procedural Programming with C

1 Question 1

1.1 Part (A): Code

```c
#include <stdio.h>

int main() {
    int my_int;
    int* my_int_pointer;
    long my_long;
    double * my_double_pointer;
    char ** my_char_pointer_pointer;

    printf("The size of my_int is %lu bytes\n", sizeof(my_int));
    printf("The size of my_int_pointer is %lu bytes\n", sizeof(my_int_pointer));
    printf("The size of my_long is %lu bytes\n", sizeof(my_long));
    printf("The size of my_double_pointer is %lu bytes\n", sizeof(my_double_pointer));
    printf("The size of my_char_pointer_pointer is %lu bytes\n", sizeof(my_char_pointer_pointer));
}
```

Listing 1: `question1.c`

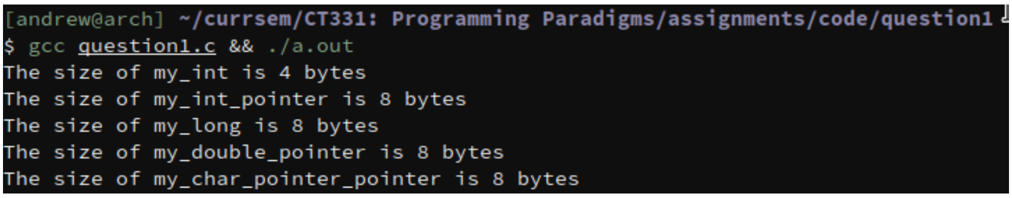

Figure 1: Console Output of `question1.c`

1.2 Part (B): Comments

The amount of memory allocated to variables of different types in C is determined at compile-time, and is dependent on the architecture of the machine for which it is being compiled and the compiler used.

- On my machine, using GCC, an `int` is allocated 4 bytes. This is the usual amount allocated on both 32-bit and 64-bit systems (my machine being of the latter kind), although older 32-bit systems used 2 bytes for an `int` (the same amount as for a `short`). 4 bytes is used even on 64-bit machines to maintain backwards compatibility with older 32-bit architectures.
- An `int*` (a pointer to a variable of type `int`) is allocated 8 bytes on my machine. This is because that my machine has a 64-bit architecture, and therefore an address in memory is represented using 64 bits (8 bytes). If this were compiled for a 32-bit machine, the size of an pointer would be 4 bytes since addresses are 32-bit.
- A `long` is allocated 8 bytes on my machine. This is because my machine is 64-bit and a `long` is typically 8 bytes in length on such machines. On 32-bit machines, a `long` is typically 4 bytes.
- The size of a pointer to a `double` is the same as the size of any other pointer on the same machine; on 64-bit machines, pointers are 8 bytes, and on 32-bit machines, they are 4 bytes. The type of data to which a pointer points has no effect on the size of the pointer, as the pointer is just a memory address.
- A pointer to a `char` pointer is the same size as any other pointer: 8 bytes on a 64-bit machine and 4 bytes on a 32-bit machine. Note: it might be more intuitive to refer to a "character pointer pointer" as a pointer to a string in certain situations, as strings are character arrays, and an array variable acts as a pointer to the first element in the array.

2 Question 2

```c
// returns the number of elements in the list
int length(listElement* list);

// push a new element onto the head of a list and update the list reference using side effects
void push(listElement** list, char* data, size_t size);

// pop an element from the head of a list and update the list reference using side effects
listElement* pop(listElement** list);

// enque a new element onto the head of the list and update the list reference using side effects
void enqueue(listElement** list, char* data, size_t size);

// dequeue an element from the tail of the list
listElement* dequeue(listElement* list);
```

Listing 2: My Additions to `linkedList.h`

```c
// returns the number of elements in the list
int length(listElement* list) {
    int length = 0;
    listElement* current = list;

    // traversing the list and counting each element
    while(current != NULL){
        length++;
        current = current->next;
    }

    return length;
}

// push a new element onto the head of a list and update the list reference using side effects
void push(listElement** list, char* data, size_t size) {
    // create the new element
    listElement* newElement = createEl(data, size);

    // handle malloc errors
    if (newElement == NULL) {
        fprintf(stderr, "Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }

    // make the the new element point to the current head of the list
    newElement->next = *list;

    // make the list reference to point to the new head element
    *list = newElement;
}


// pop an element from the head of a list and update the list reference using side effects
// assuming that the desired return value here is the popped element, as is standard for POP operations
listElement* pop(listElement** list) {
    // don't bother if list is non existent
    if (*list == NULL) { return NULL; }
;
    // getting reference to the element to be popped
    listElement* poppedElement = *list;

```

```c
    // make the the second element the new head of the list -- this could be NULL, so the list would be NULL also
    *list = (*list)->next;

    // detach the popped element from the list
    poppedElement->next = NULL;

    return poppedElement;
}


// enque a new element onto the head of the list and update the list reference using side effects
// essentially the same as push
void enqueue(listElement** list, char* data, size_t size) {
    // create the new element
    listElement* newElement = createEl(data, size);

    // handle malloc errors
    if (newElement == NULL) {
        fprintf(stderr, "Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }

    // make the the new element point to the current head of the list
    newElement->next = *list;

    // make the list reference to point to the new head element
    *list = newElement;
}


// dequeue an element from the tail of the list by removing the element from the list via side effects, and returning the removed item
// assuming that we want to return the dequeued element rather than the list itself, as enqueue returns nothing and uses side effects, so dequeue should also use side effects
listElement* dequeue(listElement* list) {
    // there are three cases that we must consider: a list with 0 elements, a list with 1 element, & a list with >=2 elements

    // don't bother if list is non existent
    if (list == NULL) { return NULL; }

    // if there is only one element in the list, i.e. the head element is also the tail element, just returning this element
    // this means that the listElement pointer that was passed to this function won't be updated
    // ideally, we would set it to NULL but we can't do that since `list` is a pointer that has been passed by value, so we can't update the pointer itself. we would need a pointer to a pointer to have been passed
    if (list->next == NULL) {
        return list;
    }

    // traversing the list to find the second-to-last element
    listElement* current = list;
    while (current->next->next != NULL) {
        current = current->next;
    }

    // get reference to the element to be dequeued
    listElement* dequeuedElement = current->next;

```

```c
    // make the penultimate element the tail by removing reference to the old tail
    current->next = NULL;

    return list;
}
```

Listing 3: My Additions to `linkedList.c`

```c
    // test length function
    printf("Testing length()\n");
    int l_length = length(l);
    printf("The length of l is %d\n\n", l_length);

    // test push
    printf("Testing push()\n");
    push(&l, "yet another test string", sizeof("yet another test string"));
    traverse(l);
    printf("\n\n");

    // test pop
    printf("Testing pop()\n");
    listElement* popped = pop(&l);
    traverse(l);
    printf("\n\n");

    // Test delete after
    printf("Testing deleteAfter()\n");
    deleteAfter(l);
    traverse(l);
    printf("\n");

    // test enqueue
    printf("Testing enqueue()\n");
    enqueue(&l, "enqueued test string", sizeof("enqueued test string"));
    traverse(l);
    printf("\n");

    // test dequeue
    printf("Testing dequeue()\n");
    dequeue(l);
    traverse(l);
    printf("\n");

    printf("\nTests complete.\n");
```

Listing 4: My Additions to `tests.c`

```
[andrew@arch] ~/currsem/CT331: Programming Paradigms/assignments/assignment1/code/question2
$ gcc *.c && ./a.out
Tests running...
Test String (1).

Testing insertAfter()
Test String (1).
another string (2)
a final string (3)

Testing length()
The length of l is 3

Testing push()
yet another test string
Test String (1).
another string (2)
a final string (3)


Testing pop()
Test String (1).
another string (2)
a final string (3)


Testing deleteAfter()
Test String (1).
a final string (3)

Testing enqueue()
enqueued test string
Test String (1).
a final string (3)

Testing dequeue()
enqueued test string
Test String (1).


Tests complete.
```

Figure 2: Console Output for Question 2

3 Question 3

```c
#ifndef CT331_ASSIGNMENT_LINKED_LIST
#define CT331_ASSIGNMENT_LINKED_LIST

typedef struct listElementStruct listElement;

//Creates a new linked list element with given content of size
//Returns a pointer to the element
listElement* createEl(void* data, size_t size, void (*printFunction)(void*));

//Prints out each element in the list
void traverse(listElement* start);

//Inserts a new element after the given el
//Returns the pointer to the new element
listElement* insertAfter(listElement* after, void* data, size_t size, void (*printFunction)(void*));

//Delete the element after the given el
void deleteAfter(listElement* after);

// returns the number of elements in the list
int length(listElement* list);

// push a new element onto the head of a list and update the list reference using side effects
```

```c
void push(listElement** list, void* data, size_t size, void (*printFunction)(void*));

// pop an element from the head of a list and update the list reference using side effects
listElement* pop(listElement** list);

// enque a new element onto the head of the list and update the list reference using side effects
void enqueue(listElement** list, void* data, size_t size, void (*printFunction)(void*));

// dequeue an element from the tail of the list
listElement* dequeue(listElement* list);

#endif
```

Listing 5: genericLinkedList.h

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "genericLinkedList.h"

typedef struct listElementStruct{
    void* data;
    void (*printFunction)(void*);
    size_t size;
    struct listElementStruct* next;
} listElement;

//Creates a new linked list element with given content of size
//Returns a pointer to the element
listElement* createEl(void* data, size_t size, void (*printFunction)(void*)) {
    listElement* e = malloc(sizeof(listElement));
    if(e == NULL){
        //malloc has had an error
        return NULL; //return NULL to indicate an error.
    }
    void* dataPointer = malloc(sizeof(void)*size);
    if(dataPointer == NULL){
        //malloc has had an error
        free(e); //release the previously allocated memory
        return NULL; //return NULL to indicate an error.
    }
    strcpy(dataPointer, data);
    e->data = dataPointer;

    e->printFunction = printFunction;

    e->size = size;
    e->next = NULL;
    return e;
}

//Prints out each element in the list
void traverse(listElement* start){
    listElement* current = start;
    while(current != NULL){
        current->printFunction(current->data);
        // printf("%s\n", current->data);
        current = current->next;
    }
}

//Inserts a new element after the given el
```

```c
//Returns the pointer to the new element
listElement* insertAfter(listElement* el, void* data, size_t size, void (*printFunction)(void*)){
  listElement* newEl = createEl(data, size, printFunction);
  listElement* next = el->next;
  newEl->next = next;
  el->next = newEl;
  return newEl;
}

//Delete the element after the given el
void deleteAfter(listElement* after){
  listElement* delete = after->next;
  listElement* newNext = delete->next;
  after->next = newNext;
  //need to free the memory because we used malloc
  free(delete->data);
  free(delete);
}

// returns the number of elements in the list
int length(listElement* list) {
    int length = 0;
    listElement* current = list;

    // traversing the list and counting each element
    while(current != NULL){
        length++;
        current = current->next;
    }

    return length;
}

// push a new element onto the head of a list and update the list reference using side effects
void push(listElement** list, void* data, size_t size, void (*printFunction)(void*)) {
    // create the new element
    listElement* newElement = createEl(data, size, printFunction);

    // handle malloc errors
    if (newElement == NULL) {
        fprintf(stderr, "Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }

    // make the the new element point to the current head of the list
    newElement->next = *list;

    // make the list reference to point to the new head element
    *list = newElement;
}


// pop an element from the head of a list and update the list reference using side effects
// assuming that the desired return value here is the popped element, as is standard for POP operations
listElement* pop(listElement** list) {
    // don't bother if list is non existent
    if (*list == NULL) { return NULL; }
;
    // getting reference to the element to be popped
    listElement* poppedElement = *list;

```

```c
    // make the the second element the new head of the list -- this could be NULL, so the list would be NULL also
    *list = (*list)->next;

    // detach the popped element from the list
    poppedElement->next = NULL;

    return poppedElement;
}


// enque a new element onto the head of the list and update the list reference using side effects
// essentially the same as push
void enqueue(listElement** list, void* data, size_t size, void (*printFunction)(void*)) {
    // create the new element
    listElement* newElement = createEl(data, size, printFunction);

    // handle malloc errors
    if (newElement == NULL) {
        fprintf(stderr, "Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }

    // make the the new element point to the current head of the list
    newElement->next = *list;

    // make the list reference to point to the new head element
    *list = newElement;
}


// dequeue an element from the tail of the list by removing the element from the list via side effects, and returning the removed item
// assuming that we want to return the dequeued element rather than the list itself, as enqueue returns nothing and uses side effects, so dequeue should also use side effects
listElement* dequeue(listElement* list) {
    // there are three cases that we must consider: a list with 0 elements, a list with 1 element, & a list with >=2 elements

    // don't bother if list is non existent
    if (list == NULL) { return NULL; }

    // if there is only one element in the list, i.e. the head element is also the tail element, just returning this element
    // this means that the listElement pointer that was passed to this function won't be updated
    // ideally, we would set it to NULL but we can't do that since `list` is a pointer that has been passed by value, so we can't update the pointer itself. we would need a pointer to a pointer to have been passed
    if (list->next == NULL) {
        return list;
    }

    // traversing the list to find the second-to-last element
    listElement* current = list;
    while (current->next->next != NULL) {
        current = current->next;
    }

    // get reference to the element to be dequeued
    listElement* dequeuedElement = current->next;

```

```
    // make the penultimate element the tail by removing reference to the old tail
    current->next = NULL;

    return list;
}
```

Listing 6: genericLinkedList.c

```
#include <stdio.h>
#include "tests.h"
#include "genericLinkedList.h"

// functions to print out different data types
// a more professional design might be to put these in the genericLinkedList header file but i only need these for testing purposes
void printChar(void* data) {
    printf("%c\n", *(char*) data);
}

void printStr(void* data) {
    printf("%s\n", (char*) data);
}

void printInt(void* data) {
    printf("%d\n", *(int*) data);
}

void runTests(){
    printf("Tests running...\n");

    listElement* l = createEl("Test String (1).", sizeof("Test String (1)."), printStr);
    //printf("%s\n%p\n", l->data, l->next);
    //Test create and traverse
    traverse(l);
    printf("\n");

    //Test insert after
    printf("Testing insertAfter()\n");
    listElement* l2 = insertAfter(l, "another string (2)", sizeof("another string (2)"), printStr);
    insertAfter(l2, "a final string (3)", sizeof("a final string (3)"), printStr);
    traverse(l);
    printf("\n");

    // test length function
    printf("Testing length()\n");
    int l_length = length(l);
    printf("The length of l is %d\n\n", l_length);

    // test push
    printf("Testing push()\n");
    push(&l, "yet another test string", sizeof("yet another test string"), printStr);
    traverse(l);
    printf("\n\n");

    // test pop
    printf("Testing pop()\n");
    listElement* popped = pop(&l);
    traverse(l);
    printf("\n\n");

    // Test delete after
    printf("Testing deleteAfter()\n");
```

```
    deleteAfter(l);
    traverse(l);
    printf("\n");

    // test enqueue
    printf("Testing enqueue()\n");
    enqueue(&l, "enqueued test string", sizeof("enqueued test string"), printStr);
    traverse(l);
    printf("\n");

    // test dequeue
    printf("Testing dequeue()\n");
    dequeue(l);
    traverse(l);
    printf("\n");

    printf("Testing pushing different data types\n");
    int myint = 42;
    push(&l, &myint, sizeof(myint), printInt);
    char mychar = 'c';
    push(&l, &mychar, sizeof(mychar), printChar);
    traverse(l);
    printf("\n\n");

    printf("\nTests complete.\n");
}
```

Listing 7: `tests.c`

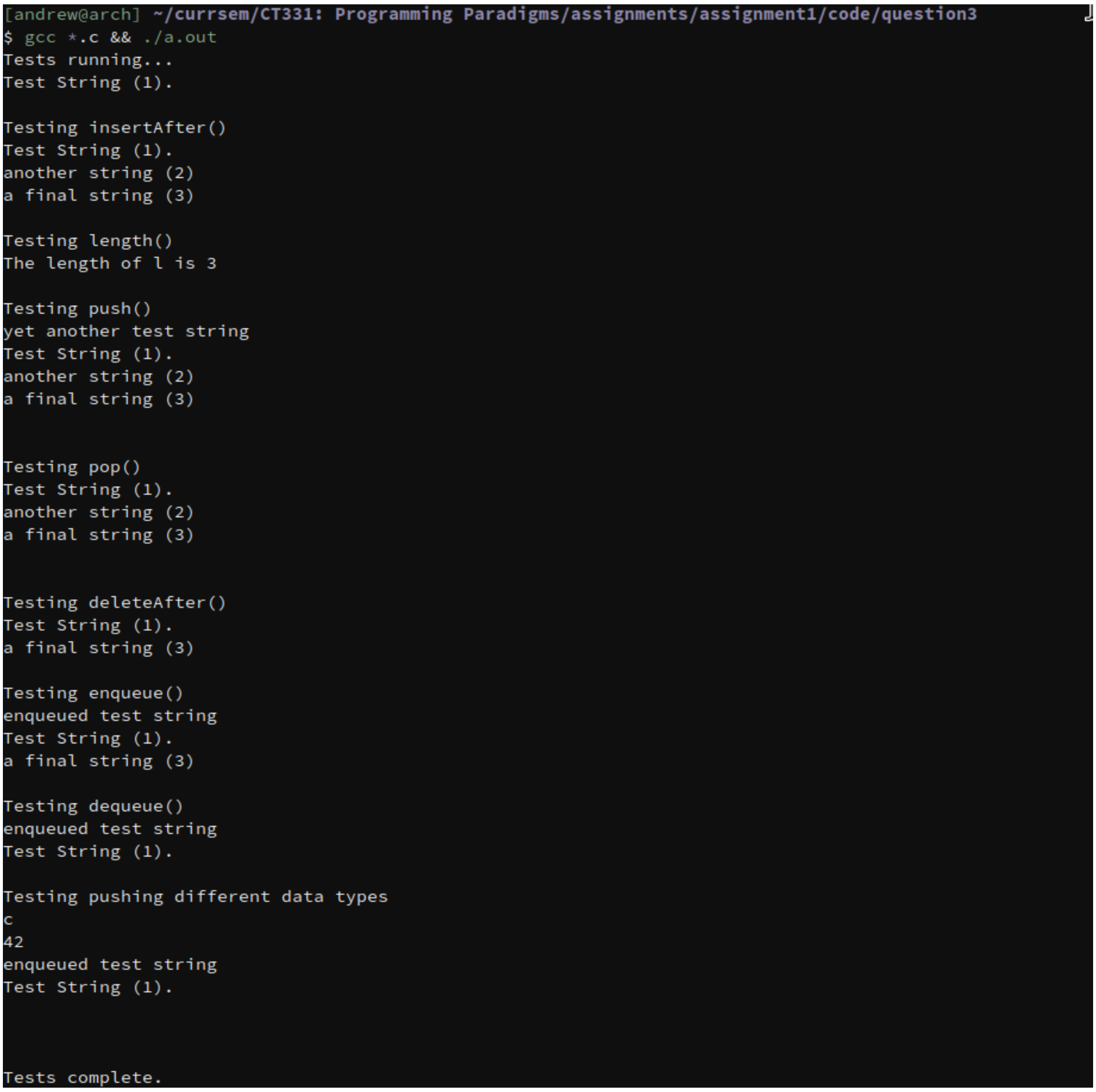

Figure 3: Console Output for Question 3

4 Question 4

4.1 Part 1

Any algorithm for traversing a singly linked list in reverse will always first require traversing the list forwards, and will therefore be *at least* somewhat less efficient than a forwards traversal. One of the simplest ways to traverse a linked list in reverse is to use a recursive function.

```c
void reverse_traverse(listElement* current){
    if (current == NULL) { return; }
    reverse_traverse(current->next);
    current->printFunction(current->data);
}
```

Listing 8: Recursive Function to Traverse a Singly Linked List in Reverse

This is quite inefficient as it requires that the function call for each node persists on the stack until the last node is reached, using a lot of stack memory. Another approach would be to iteratively reverse the linked list, by making some kind of data structure, linked list or otherwise, that contains the data of the original linked list but in reverse, and then iterating over that forwards. This would likely be more efficient in terms of memory & computation.

Because traversing a linked list in reverse always requires traversing it forwards first, any reverse algorithm will take at least twice as much memory & computation as traversing it forwards, which is $O(n)$. It will also require that some way of storing the data in reverse in memory, either explicitly with a data, like in the iterative approach, or in the same manner as the recursive approach, wherein the data is stored in reverse by the nested structure of the function calls: as each function call returns, the call structure is iterated through in reverse. Therefore, we also have at least $O(n)$ memory usage, as we have to store some sort of reverse data structure.

4.2 Part 2

The simplest way in which the structure of a linked list could be changed to make backwards traversal less intensive is to change it from a singly linked list to a doubly linked list, i.e. instead of each node in the list containing a pointer to just the next node, make each node contain a pointer to both the next node & the previous node. The backwards traversal of a doubly linked list is no more intensive than the forwards traversal of a linked list. The drawback of using a doubly linked list is that it requires slightly more memory per node than a singly linked list, as you're storing an additional pointer for every node.