CT331 Assignment 1

# **Jobe George – 16323856**

<https://github.com/JobeGeo26/CT331_Assignment1>

# **Question 1**

**(A)**

**Assignment.c**

#include <stdio.h>

int main(int arg, char\* argc[]) {

printf("Jobe George- 16323856\n");

int a = 2;

printf("Size of int is %d", sizeof(a));

int b = 2;

int \*c = &b;

printf("\nSize of int\* is %d", sizeof(c));

long d = 200;

printf("\nSize of long is %d", sizeof(d));

double e = 20.1;

double \*f = &e;

printf("\nSize of double\* is %d", sizeof(f));

char g = 'a';

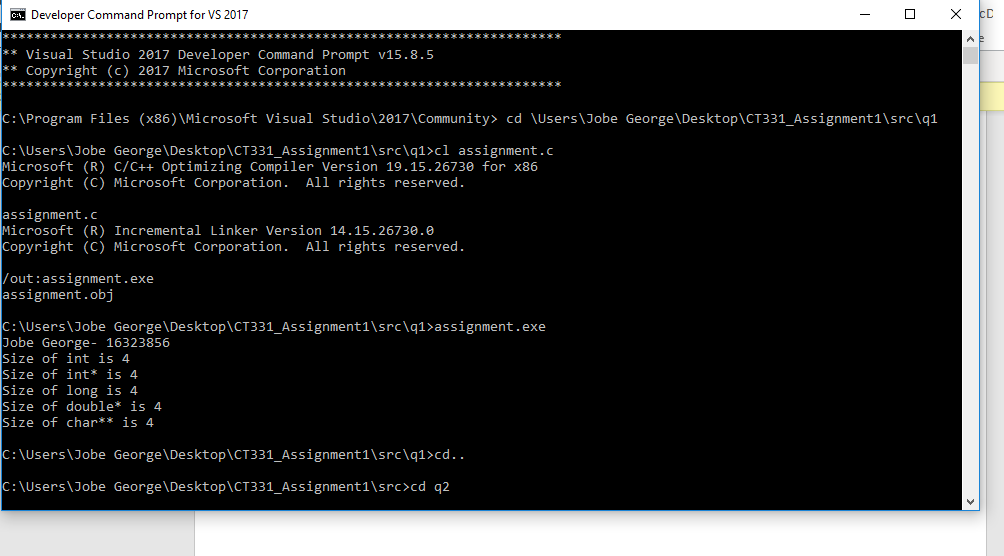
char \*h = &g;

char \*\*i = &h;

printf("\nSize of char\*\* is %d\n", sizeof(i));

return 0;

}



1(B) The size of each pointer type turns out to be 4. Through knowledge we know that pointers typically have a size of 4 bytes. Also, data types int and long also have 4 bytes size in C.

# **Question 2**

**LinkedList.h**

#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(char\* data, size\_t size);

//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, char\* data, size\_t size);

//Delete the element after the given el

void deleteAfter(listElement\* after);

//Returns the number of elements in a linked list

int length(listElement\* list);

//push a new element onto the head of a list.

void push(listElement\*\* list, char\* data, size\_t size);

//pop an element from the head of a list.

listElement\* pop(listElement\*\* list);

//enqueue a new element onto the head of the list.

void enqueue(listElement\*\* list, char \*data, size\_t size);

//dequeue an element from the tail of the list.

listElement\* dequeue(listElement\* list);

#endif

**LinkedList.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "linkedList.h"

typedef struct listElementStruct {

char\* data;

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(char\* data, size\_t size) {

listElement\* e = malloc(sizeof(listElement));

if (e == NULL) {

//malloc has had an error

return NULL; //return NULL to indicate an error.

}

char\* dataPointer = malloc(sizeof(char)\*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->size = size;

e->next = NULL;

return e;

}

//Prints out each element in the list

void traverse(listElement\* start) {

listElement\* current = start;

while (current != NULL) {

printf("%s\n", current->data);

current = current->next;

}

}

//Inserts a new element after the given el

//Returns the pointer to the new element

listElement\* insertAfter(listElement\* el, char\* data, size\_t size) {

listElement\* newEl = createEl(data, size);

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 this linked list.

int length(listElement\* list) {

int lengthCount = 0;

listElement\* temp = list;

while (temp != NULL) {

temp = temp->next;

lengthCount++;

}

return lengthCount;

}

// push a new element onto the head of a list.

void push(listElement\*\* list, char \*data, size\_t size) {

listElement\* newElement = createEl(data, size);

newElement->next = \*list;

\*list = newElement;

}

// pop an element from the head of a list.

listElement\* pop(listElement\*\* list) {

if (\*list != NULL) {

listElement\* element = (\*list)->next;

\*list = (\*list)->next;

return element;

}

return \*list;

}

//enqueue a new element onto the head of the list.

void enqueue(listElement\*\* list, char\* data, size\_t size) {

push(list, data, size);

}

//dequeue an element from the tail of the list.

listElement\* dequeue(listElement\* list) {

listElement\* temp = list;

while ((temp->next)->next != NULL)

{

temp = temp->next;

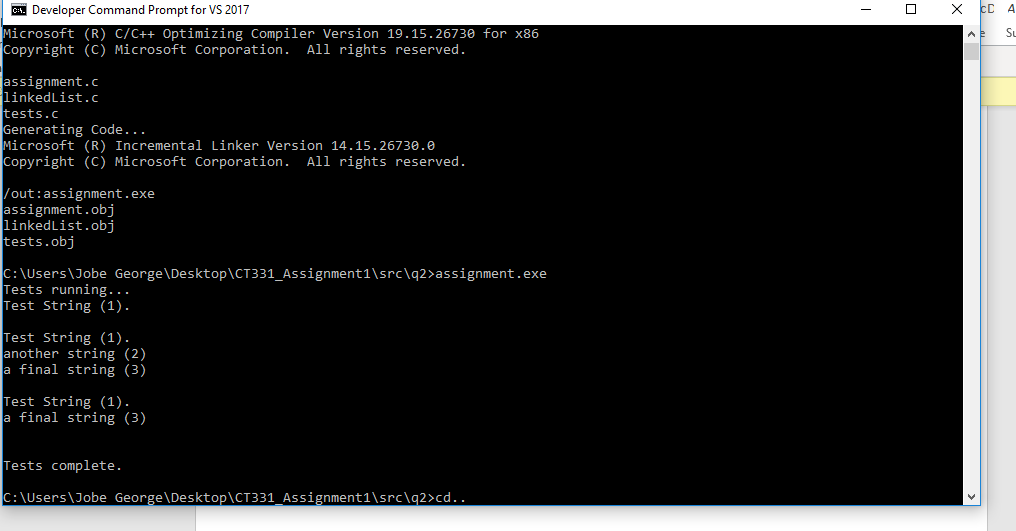
}

listElement\* tail = temp->next;

temp->next = NULL;

return tail;

}



# **Question 3**

**genericLinkedList.h**

#ifndef CT331\_ASSIGNMENT\_GENERIC\_LINKED\_LIST

#define CT331\_ASSIGNMENT\_GENERIC\_LINKED\_LIST

typedef void(\*printFn)(void\* data);

typedef struct genericlistElementStruct {

void\* data;

size\_t size;

printFn printFunction;

struct genericlistElementStruct\* next;

} genericlistElement;

//Creates a new linked list element with given content of size

//Returns a pointer to the element

genericlistElement\* createEl(void\* data, size\_t size, printFn print);

//Prints out each element in the list

void traverse(genericlistElement\* start);

//Inserts a new element after the given el

//Returns the pointer to the new element

genericlistElement\* insertAfter(genericlistElement\* after, void\* data, size\_t size, printFn print);

//Delete the element after the given el

void deleteAfter(genericlistElement\* after);

//returns the number of elements in the linked list

int length(genericlistElement\* list);

//push a new element onto the head of a list.

void push(genericlistElement\*\* list, void\* data, size\_t size, printFn print);

//pop an element from the head of a list.

genericlistElement\* pop(genericlistElement\*\* head);

//enqueue a new element onto the head of the list.

void enqueue(genericlistElement\*\* list, void\* data, size\_t size, printFn print);

//dequeue an element from the tail of the list.

genericlistElement\* dequeue(genericlistElement\* list);

//prints out an integer element

void printInt(void\* data);

void printChar(void\* data);

//prints out a string element

void printStr(void\* data);

//prints out a float element

void printFloat(void\* data);

#endif

**genericLinkedList.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "genericLinkedList.h"

//Creates a new linked list element with given content of size

//Returns a pointer to the element

genericlistElement\* createEl(void\* data, size\_t size, printFn print) {

genericlistElement\* e = malloc(sizeof(genericlistElement));

if (e == NULL) {

//malloc has had an error

return NULL; //return NULL to indicate an error.

}

void\* dataPointer = malloc(size);

if (dataPointer == NULL) {

//malloc has had an error

free(e); //release the previously allocated memory

return NULL; //return NULL to indicate an error.

}

memmove(dataPointer, data, size);

e->data = dataPointer;

e->size = size;

e->printFunction = print;

e->next = NULL;

return e;

}

//Prints out each element in the list

void traverse(genericlistElement\* head) {

genericlistElement\* current = head;

while (current != NULL) {

current->printFunction(current->data);

current = current->next;

}

}

//Inserts a new element after the given el

//Returns the pointer to the new element

genericlistElement\* insertAfter(genericlistElement\* el, void\* data,

size\_t size, printFn printFunc) {

genericlistElement\* newEl = createEl(data, size, printFunc);

genericlistElement\* next = el->next;

newEl->next = next;

el->next = newEl;

return newEl;

}

//Delete the element after the given element

void deleteAfter(genericlistElement\* after) {

genericlistElement\* delete = after->next;

genericlistElement\* 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 a linked list.

int length(genericlistElement\* list) {

int lengthCount = 0;

genericlistElement\* temp = list;

while (temp != NULL) {

temp = temp->next;

lengthCount++;

}

return lengthCount;

}

// push a new element onto the head of a list.

void push(genericlistElement\*\* list, void\* data, size\_t size, printFn print) {

genericlistElement\* newElement = createEl(data, size, print);

newElement->next = \*list;

\*list = newElement;

}

// pop an element from the head of a list.

genericlistElement\* pop(genericlistElement\*\* list) {

if (\*list != NULL) {

genericlistElement\* element = (\*list)->next;

\*list = (\*list)->next;

return element;

}

return \*list;

}

//enqueue a new element onto the head of the list.

void enqueue(genericlistElement\*\* list, void\* data, size\_t size, printFn print) {

push(list, data, size, print);

}

//dequeue an element from the tail of the list.

genericlistElement\* dequeue(genericlistElement\* list) {

genericlistElement\* temp = list;

while (temp->next->next != NULL) {

temp = temp->next;

}

genericlistElement\* tail = temp->next;

temp->next = NULL;

return tail;

}

void printChar(void\* data) {

printf("%c\n", \*(char\*)data);

}

void printFloat(void\* data) {

printf("%f\n", \*(float\*)data);

}

void printStr(void\* data) {

printf("%s\n", data);

}

void printInt(void\* data) {

printf("%d\n", \*(int\*)data);

}



# **Question 4**

1. A Singly Linked List would result in huge memory intensity to traverse in reverse because it has no pointer to previous node so would require you to iterate through the list once again which results in a lot of memory.
2. A solution should be to change its structure to a Doubly Linked List which has a reference to the previous nodes which makes traversing in reverse much easier and much more efficient. Doubly Linked List require more storage than Singly Linked List but it’s much more efficient and requires lot less memory for traversing in reverse in huge lists using Singly Linked List