Homework 1: Diving Into C

Due: 11:59 PM, September 9, 2021

This homework assignment will require you to create a series of "header" files that I will be able to link to in order to run your code against test cases that I have designed. Header files are quite easy to use. You will end up creating a header file for each of the included exercises (`ex_1.h`, `ex_2.h`, `ex_3.h`, `ex_4.h`, and `ex_5.h`). In each of these header files you will write a single function that is specific to the exercise at hand. That header file can then be accessed by a `.c` file (by adding the line: `#include "ex_#.h"` to the top of file). Each of the exercises will provide you with a sample C program that you can use to test the header file you have created for that exercise. Each header file need only consist of the function you were asked to write for that exercise (perhaps with a reference to other header files if you need to use additional functions found in `<string.h>` or in support code which I provide to you in `"linkedList.h"`).

Note the different use of brackets and quotation marks when including files. When including header files from the standard library you will use the angle brackets. When including header files that you supply from your current working directory then you will use the quotations marks while specifying the relative path to the header file. You can find an example usage of header files on CS-Data in this location:

```
\\CS-Data\Courses\cmsc432\support_code\using_header_files
```

Ex 1 (20 points): Fun with Iteration & `printf`

Your first task is to write a function that prints ASCII triangles to the screen. You will be provided with the length of the base of the triangle, and you must then construct the triangle. Triangles will be of the type shown in the example below. To ensure that the triangles are balanced and look "nice" you will only be given base lengths that have an odd value. For example, a triangle with a base length of 7 would look like the following:

```
   #
  ###
 #####
#######
```

Note that there are 7 characters at the base of the triangle and the triangle has a height of $\lceil \frac{baseLength}{2} \rceil$. The function you are to implement can be described as follows:

Function Prototype:

```
void printTriangle(int baseLength);
```

Inputs:

`baseLength`: The length of the triangle's base.

Return Value:

`void` – nothing.

Suppose that you created the following driver program to call the `printTriangle` function within your `ex_1.h` file:

```
#include "ex_1.h"

int main(int argc, char** argv) {
  printTriangle(5);
  printTriangle(15);

  return 0;
}
```

Executing this code should result in the following output:

```
  #
 ###
#####
       #
      ###
     #####
    #######
   #########
  ###########
 #############
###############
```

You should test your function accordingly and place the function within the `ex_1.h` header file to be tested by a driver file of your own creation.

Ex 2 (20 points): Fun with C Strings

Your goal for this exercise is to write a function that will create and return a new string that is composed of two other strings. Specifically, this function (called `instr`) will insert one string into the other (at a specified location) to produce the combined result.

The function you are to implement can be described as follows:

Function Prototype:

```
char* instr(char* str1, char* str2, int n);
```

Inputs:

`str1`: The "outer" string that the "inner" string will be placed within.

`str2`: The "inner" string that will be inserted into the "outer" string.

`n`: The position in the "outer" string at which to insert the "inner" string.

Return Value:

`char*`: A pointer to a ***new*** string which contains the combined result.

Example: The following code would result in printing the string "Held!" (stored in `str`) to the screen:

```
#include <stdio.h>
#include <stdlib.h>
#include "ex_2.h"

int main(int argc, char** argv) {
    char *str = instr("Held!", "lo Worl", 3);
    printf("%s\n", str);

    free(str);
    return 0;
}
```

Place your `instr` function in the `ex_2.h` header file. Within that file you will likely find it useful to: `#include <string.h>` to access common string functions like `strlen`. Also, in this exercise (and future ones) you will need to use `malloc` to allocate new space for the array (and correspondingly `free`). You will have access to `malloc` and `free` by calling: `#include <stdlib.h>` at the top of your header file.

Ex 3 (20 points): Fun with Arrays

Your next task is to write a function that will merge two sorted arrays of integers into a single sorted array. The function you are to implement can be described as follows:

Function Prototype:

`int* merge(int *arr1, int arr1Length, int *arr2, int arr2Length);`

Inputs:

`arr1`: The first array, containing `arr1Length` integers, sorted in ascending order.

`arr1Length`: The number of items in array `arr1`.

`arr2`: The second array, containing `arr2Length` integers, sorted in ascending order.

`arr2Length`: The number of items in array `arr2`.

Return Value:

`int*`: A pointer to a ***new*** array containing all the integers in ascending order.

Example: The following code would result in printing the array: $\{1, 1, 2, 3, 4, 7, 8, 9, 13\}$ (which is stored in `arr3`):

```
#include <stdio.h>
#include "ex_3.h"

void printArray(int *arr, int length) {
    int i;
```

```
    printf("{");
    for(i=0; i < length; i++) {
        printf("%d", arr[i]);
        if (i != length-1)
            printf(", ");
    }
    printf("}\n");
}

int main(int argc, char** argv) {
    int arr1[] = {1, 4, 7, 9};
    int arr2[] = {1, 2, 3, 8, 13};

    int *arr3 = merge(arr1, 4, arr2, 5);
    printArray(arr3, 9);

    free(arr3);
    return 0;
}
```

Place your `merge` function in the `ex_3.h` header file.

Ex 4 (20 points): Fun with Pointers & `malloc`

Your next task is to write a function that inserts items into a sorted linked list. But first, a little intro on implementing data structures in the C programming language. As you all know, C is not an object-oriented language! Therefore, when writing functions to interact with data structures, we must pass in a pointer to the data structure that we intend to manipulate. We don't have methods to interact with objects as we do in C++! Therefore, instead of something like the following C++ code which uses objects and methods:

```
listOfNumbers.insert(42);
listOfNumbers.insert(18);
listOfNumbers.clear();
```

We would instead write something like the following:

```
list_insert(&listOfNumbers, 42);
list_insert(&listOfNumbers, 18);
list_clear(&listOfNumbers);
```

Again, lacking methods (an Object-Oriented feature that ties functionality to specific instances of classes) we must create functions that take pointers to the data structures we intend to manipulate.

Now, back to the task at hand – writing a function that inserts an item into a sorted linked list! The function you are to implement is described as follows:

Function Prototype:

```
void list_insert(struct Node** head, int value);
```

Inputs:

`head`: A pointer to the variable which points to the head `Node` of the list.

`value`: The value to be stored as data within the sorted linked list.

Return Value:

`void` – nothing.

You will also be provided with the `Node` struct as well as a handy `list_print` method which you can use to measure your progress and observe changes to your list. The corresponding code can be seen here:

```
struct Node {
    struct Node *next;
    int data;
} typedef Node;

void list_print(struct Node *head) {
    struct Node *curr;
    printf("{");
    for(curr=head; curr != NULL; curr=curr->next) {
        printf("%d", curr->data);
        if (curr->next != NULL)
            printf(", ");
    }
    printf("}\n");
}
```

In this model, `next` will point to `NULL` if the `Node` is the last in the list. Similarly, the pointer to the head of the list would contain `NULL` if the list is empty.

A quick reminder to those that haven't programmed in C++ in a while: similar to objects in C++, if you have a pointer to a struct, you can access the fields of that struct using the arrow operator `(->)` instead of first having to dereference the pointer (*) and then applying the dot operator (.) to access the field. Therefore, if you have a pointer to a `Node` called `n`, then you can access the fields of the `Node` pointed to by `n` via either:

```
(*n).data  // a little bit ugly
    or
n->data    // a bit cleaner
```

Note: Be aware that the type of the `head` parameter in the `list_insert` function is `struct Node**`. What is this? It's a pointer to a pointer! Why do this? Because getting a programmer to provide us with the address to the first node in a list isn't good enough. We also need the address of the variable in which the programmer stores the pointer to that first node so that we can change what it points to!

Consider this: if a programmer has a pointer called `head` that points to a list such as $\{2, 4, 6\}$ then that `head` pointer, by definition, points to the `Node` containing the value 2. If the programmer passes just that pointer to the `list_insert` function, then we receive a copy of what was pointed to (the `Node` with the 2 and, by extension, everything after it) but we will be unable to change the value that the programmer has stored in the `head` variable. We may **need** to change that variable's value if we end up adding a node that becomes the new front of the list – adding the value of 1, for example. Another example: if the list starts out empty (`head` points to NULL) then we need a way to update that programmer's pointer to point to the new node. If we simply receive `NULL` as our input parameter, there's nothing we can do to inform the programmer of the new first `Node` in the list – their pointer will simply still have `NULL` in it. Therefore, we need to pass in a pointer to the pointer that identifies the head node: `struct Node**`. This allows us to update the variable that the programmer uses to identify the front of their list.

Example: Take a look at the following sample code which leverages the `Node` struct, `list_insert` and `list_print`:

```
#include "ex_4.h"

int main(int argc, char** argv) {
    struct Node *head = NULL;
    list_print(head);

    list_insert(&head, 3);
    list_insert(&head, 1);
    list_insert(&head, 2);
    list_print(head);
    return 0;
}
```

This code will create an empty linked list, print the linked list, insert three values, and print the resulting sorted list. The output should match the following:

```
{}
{1, 2, 3}
```

Place your `list_insert` function in the `ex_4.h` header file. You will find the support code (`struct Node` and `list_print`) located at:

`\\CS-Data\Courses\cmsc432\homework\hw_1\linkedList.h`

You should copy this file to your working directory and then: `#include "linkedList.h"` at the top of your `ex_4.h` file to utilize the code within it.

Ex 5 (20 points): Fun with Pointers & `free`

Your next task is very much related to the previous task. You must write a function that will delete an entire linked list – after all, `ex_4` has a memory leak in it that we need to fix! You will have the same support code as before (`linkedList.h`). All you have to do is provide a single function that will delete the entire linked list when provided with a pointer to the pointer which identifies the list's head.

The function you are to implement is described as follows:

Function Prototype:

```
void list_clear(struct Node** head);
```

Inputs:

`head`: A pointer to the variable which points to the head `Node` of the list.

Return Value:

`void` – nothing.

Note: Just as before, the type of `head` is `struct Node**`. The ultimate goal is to delete and reclaim the memory for all of the nodes **and** to set the programmers head pointer to `NULL` to represent and empty list. In order to update the programmer's pointer to the head `Node`, we must pass in a pointer to that pointer so that we can modify it directly and set it to `NULL`.

Example: Take a look at the following sample code:

```
#include "ex_4.h"
#include "ex_5.h"

int main(int argc, char** argv) {
    struct Node *head = NULL;
    list_insert(&head, 3);
    list_insert(&head, 1);
    list_insert(&head, 2);
    list_print(head);

    list_clear(&head);
    list_print(head);
    return 0;
}
```

This code will create a linked list, print the linked list, delete the list, and print the resulting empty list. The output should match the following:

```
{1, 2, 3}
{}
```

Place your `deleteList` function in the `ex_5.h` header file. You should access the support code exactly as you did before and insert the following line: `#include "linkedList.h"` at the top of your `ex_5.h` file.

Submission: Turning in the assignment

To turn in your assignment, place your work on the CS-Data server (`\\CS-Data`) under the following directory within your student folder:

```
\\CS-Data\Students\your_user_name\cmsc432\hw_1\
```

Make sure your folder structure is identified **exactly** as it is outlined here or you will lose 5 points for not following directions and hindering my scripts which run and test your code.

After completing all of the exercises in this homework assignment, you should have (at a minimum) the following files/folders in your network drive:

```
\\CS-Data\Students\your_user_name\cmsc432\hw_1\ex_1.h
\\CS-Data\Students\your_user_name\cmsc432\hw_1\ex_2.h
\\CS-Data\Students\your_user_name\cmsc432\hw_1\ex_3.h
\\CS-Data\Students\your_user_name\cmsc432\hw_1\ex_4.h
\\CS-Data\Students\your_user_name\cmsc432\hw_1\ex_5.h
```

Again, points will be deducted if your files are not stored in exactly these locations. Make sure that you test your functions to ensure that they work in a variety of cases (including using test cases that I didn't povide to you in the example portion of each exercise).