CS 240: Programming in C
Fall 2020
Homework 11

Prof. Turkstra

Due December 2, 2020 9:00 PM

1 Goals

This homework is a blending of trees and function pointers with an icing of casts. It is **long**, but there is a **lot of repetition** in the code. You should read the homework carefully and understand the concepts of casting and callbacks (pointers to functions). It may first look like there is too much to do—there isn't.

2 The Big Idea

The big idea is **generic trees**, trees with generic data types. `struct node` in the hw11.h file is the generic tree structure you will use.

```
/* Generic node structure */
struct node {                              /* generic tree node      */
  struct node *left;                       /* pointer to left child  */
  struct node *right;                      /* pointer to right child */
  void        *data;                       /* pointer to data struct    */
  void        (*print)(void *);            /* pointer to print function */
  void        (*delete)(void **)           /* ptr to delete function    */
  void        (*compare)(void *, void *); /* ptr to compare function  */
};
```

This structure has the regular `left` and `right` pointers of a node in a tree. What is special are the `data` field and the pointers to data specific functions in the node.

The void pointer to data (`void *data`) is a pointer to another structure that holds only data (i.e., for this homework cpu and memory structures). We use a void pointer because we don't point to only one specific structure type. You will **CAST** this pointer to the specific structure pointer types (more on this later).

The callbacks are just pointers to the `print()`, `delete()`, and `compare()` functions specific to each data type. Each data structure used will have its own functions for deleting the data

field of the node, printing the data in the node, and comparing the fields designated as the comparison field in the structure. When a node is created using `create_node()`, all of these pointers are initialized.

The data structures in this homework are:

```
/* Data structure 1: cpu structure */
struct cpu_s {
  char  *model;
  char  *manufacturer;
  int    speed;
  int    cores;
};

/* Data structure 2: memory structure */
struct memory_s {
  char  *model;
  char  *manufacturer;
  int    size;
  int    ddr_gen;
};
```

These are two different structures with different elements (fields). When you create a cpu node, for example, you allocate memory for the cpu node itself, `model`, `manufacturer`, `speed` and `cores`. You then copy the given arguments into the fields.

3 The Assignment

`print_tree()`, `print_node()`, `print_cpu_data()`, and `print_memory_data()` are written for you to have an example to look at. They show you how to cast (`void *`) to other structure pointers and how to call the functions specific to the data structures using the pointers in the `node` structure.

`print_tree()` calls the `print_node()` function to print the data of a node. `print_node()` calls the print function specified by the callback in the node structure (either `print_cpu_data` or `print_memory_data`). `print_cpu_data()` prints the data specific to the **cpu** structure (`model`, `manufacturer`, `speed`, `cores`), whereas `print_memory_data()` prints the data specific to the **memory** structure (`model`, `manufacturer`, `size`, `speed`, and `ddr_gen`).

Likewise, the `create`, `delete` and `compare` functions specific to each structure use the elements of the corresponding structure. Compare functions compare `cores` and `speed` (for cpu) and `model` and `manufacturer` (for memory).

Functions you will write

You will write the following functions:

~~`void create_node(struct node **, void *, void (*)(void *), void (*)(void **), int (*)(void *, void *);}`~~

Is that beautiful? Dynamically allocate a `struct node` and initialize the pointers of the node using arguments 2-5. The arguments are: pointer to pointer to a node, pointer to data structure, pointer to print function, pointer to delete function, and pointer to the compare function—respectively.

Assertions: pointer to pointer to node structure and arguments 2-5 should be non-NULL, pointer to node structure should be NULL.

~~`void delete_node(struct node **);`~~

Deallocate the data field by calling the delete function (using the delete function pointer in the node pointer passed in) and then free the node itself.

Assertions: pointer to pointer to node structure and pointer to node structure should be non-NULL. The left and right pointers of the node passed in should be NULL.

~~`void insert_node(struct node **, struct node *);`~~

The first argument is a pointer to pointer to the root element of a tree. The second argument is a pointer to a new element to insert into the tree. Use the compare function of the root pointer to make comparisons.

Assertions: pointer to pointer to the root and pointer to new element should be non-NULL.

~~`struct node **find_nodes(struct node *, void *, int *);`~~

~~The first argument points to the root node of a tree. The second argument is the pointer to the **data** structure of the cpu or memory to be searched for in the tree. The third ar~~gument is the integer value returned by the function that represents the number of nodes present in the tree with a matching comparison. (If this value is 1, there are no duplicates. If it is 2, there is one duplicate, etc).

This function will dynamically allocate memory for an array of N `struct node` pointers and fill it in with the addresses of the matching nodes. You will have to go through the tree twice, first to figure out how many matching nodes are present (used to allocate the required memory for the pointer array), and a second time to fill in the pointer array with the addresses of the matching nodes. Examine hw11_main.c carefully to get a better idea of the usage of the pointer array `array_of_dups`.

This function returns the array of pointers. Use the compare function of the root pointer to make comparisons.

~~Assertions: none of the arguments should be NULL.~~

~~`void remove_node(struct node **, struct node *);`~~

The first argument is a pointer to pointer to the root node of a tree. The second argument is the pointer to the node to be removed.

HINT: You will still have to do comparisons to decide whether to continue searching for the node you are looking for in the left or right subtree, but, what do you compare when you encounter duplicates? Duplicates with the same model and manufacturer?

Use the compare function of the root pointer to make comparisons.

Assertions: none of the arguments should be NULL.

~~`void delete_tree(struct node **);`~~

The argument is a pointer to pointer to the root node of a tree. Delete the whole tree and set the root pointer to NULL.

Assertions: pointer to pointer to the root should be non-NULL.

The following functions are data structure-specific functions. You will write them both for cpu and memory structures each with the proper signature matching each of its data:

~~`void create_cpu_data`~~`(void **, const char *, const char *, int, int);`

This is like the regular create functions that you have been writing for past homeworks. The first argument is a pointer to pointer to void. You will create a **cpu** node, allocate appropriate memory for the necessary fields, and populate the fields by copying the arguments into the structure.

You should then set the first argument's pointer to point to the cpu node. You will have to cast it to (`void *`).

See hw11_main.c

Assertions: pointer to pointer to void and the char pointers should be non-NULL. Pointer to void should be NULL.

~~`void delete_cpu_data(`~~`void **);`

Deallocate the memory for the relevant fields. Then deallocate the node itself. Set the pointer to NULL. You will need to cast the (`void *`) to a cpu structure (see the print function).

Assertions: pointer to pointer to void should be non-NULL. pointer should be non-NULL.

~~`int compare_cpu_data(void *`~~`, void *);`

This function will compare the overall speed of our cpus. For this you should simply multiply the `speed` and `cores` fields of the cpu structures and compare them.

Again, the pointers to the structures are not passed in as struct cpu pointers (`struct cpu *`), so that means you must cast the arguments to (`struct cpu *`) to be able to

reach the model and manufacturer fields. (`void` does not have a `speed` field nor a `cores` field, only `struct cpu` does).

The function returns an integer denoting the comparison of the two arguments passed in. The value it returns should be 1 for arg1 > arg2, 0 if arg1 == arg2, and -1 if arg1 < arg2.

For example, if you compare a cpu with 2 cores and a 3GHz clock rate with another cpu with 4 cores and 2GHz clock rate then the function should return -1, since $2 * 3 < 4 * 2$.

This function is used in insertion, search, and removal, so whatever it returns will be important in those functions.

Assertions: pointers to void should be non-NULL.

```
int compare_memory_data(void *, void *);
```

This function will compare the model and manufacturer for memories. For this you should first compare based on the model field and in the case of identical models then you should compare based on the manufacturer.

The rest of the function specification is similar to the specification for `compare_cpu_data()`.

You should write almost the same functions as the three above for the memory structure.

3.1 Input Files

There are no input files for this assignment.

3.2 Header Files

We provide a header file, hw11.h, for you. It contains prototypes for each of the functions that you will write as well as #definitions for the constants. You should not alter this file. We will replace it with the original when grading.

3.3 Error Codes

There are no error codes for this assignment.

These Standard Rules Apply

- You may add any `#includes` you need to the top of your hw11.c file;
- You may not create any global variables other than those that are provided for you. Creation of additional global variables will impact your style grade;

- Do not look at anyone else's source code. Do not work with any other students.

Submission

To submit your program for grading, type:

```
$ make submit
```

In your hw11 directory. You can do this as often as you wish. We encourage you to submit your code as often as possible. Only your final submission will be graded.

4 Grading

The operation of your functions will be graded out of 100 points. The point breakdown will be determined by the test program.

The test program will be run many times when grading. It is your job to do the same. The lowest score will be your final grade.

This homework will also have a style grade based on 20 points, with 2 points deducted for each code standard violation found.

Your code must compile successfully using `-Wall -Werror -std=c99` to receive any credit. Code that does not compile will be assigned an automatic score of 0.

If your program crashes (e.g., segmentation fault) at any point during the testing process, your score will be an automatic 0.

If your program exhibits any memory errors (e.g., corruption, double free) at any point during the testing process, your score will be an automatic 0.