Macros

Think of a function which can take a variable number of arguments. Did you find one? Great! Now let's think how you can pass variable arguments to macros.

Here is an example: `#define LOG(\dots) printf(_VA_ARGS_);`

The following statement will print the argument string exactly like `printf`. `LOG("This is a sample log")` prints "This is a sample log".

Using the above hint, implement a multi-level logging infrastructure using macros.

If `ERROR` is defined - the string to be printed is appended to the word "Error: " e.g `LOG("Some message")` would print "Error: Some message".

Otherwise, if `VERBOSE` is defined `LOG()` - the macro prints the current file name, function name, line number, and the string passed to it. For example, `LOG("Some message");` would print "`foo.c:my_function`: Some message").

If none of the above are defined, `LOG()` just prints the string. `LOG("Some message")` would print "Some message".

Linked List and Function Pointers

This puzzle has been designed to introduce you to a typical data structure which is used widely today, the singly linked list. Also, this activity requires you to understand the concept of function pointers and how they are used.

You are provided with a tar file called `linked_list.tar` . Untar it using the command `tar -xvf linked_list.tar`. This will generate the following files:

1. `list.c`
   Contains the main function.
2. `helper.c`
   Contains a bunch of helper functions along with the 4 functions you are expected to write. Also defines the globally defined variable head.
3. `helper.h`
   Header file which contains the declaration of the structure used as a node for the linked list along with the declarations of various other functions.
4. `validation.o`
   This object file contains the definition of the `create_list` and validate functions. `create_list` simply creates the linked list you need to work with, which is pointed to by head. The function validate simply validates the output provided after the list has been operated on according to the user input.
5. `Makefile`
   This file has been provided so that you can recompile your list program by simply using the command 'make'.

Requirement:

You have been provided with a linked list which is pointed to by head. This linked list was created using the `create_list` function.

You are required to define 4 functions in the helper.c file whose function prototypes are already present in that file. Each function also has a description of what that function is supposed to do.

These 4 functions are:

1. `operate_on_list`

   This function takes in as parameters a head pointer to a list and a function pointer. You are required to use the function pointer and call the function that this function pointer points to. Ideally, the function pointer you pass to this function should point to one of the following functions which you will define. This function returns the head of the list after the list has been operated on.

   You will need to understand what to do in this function and implement this before you can check whether your other functions have been implemented properly or not.

2. `reverse`

   This function takes the head of a list as a parameter, reverses that linked list and returns the head of this reversed linked list.

3. `sort_by_id` /*Sort in ascending order*/

   This function takes the head of a list as a parameter, sorts that linked list based on the data member 'id' of each node, and then returns the head of this sorted linked list.

4. `sort_by_val` /*Sort in ascending order*/

   This function takes the head of a list as a parameter, sorts that linked list based on the data member 'val' of each node, and then returns the head of this sorted linked list.

A few helper functions have been defined in the same header file. Please go through these functions to understand how nodes are created, inserted into a linked list and how a linked list can be displayed.

Attack Plan

1. After you untar all the files, and try make for the first time, you will get an error that `func_ptr` is undeclared. First and foremost, declare this function pointer appropriately.

2. Once you have done that, you can run make again. Once the program compiles successfully, it will produce an executable called list. Run this executable using the command `./list`. You should be able to see the singly linked list that has been provided to you pointed to by head. This program expects an integer input from the user.

3. Add relevant code in the switch statement present in the main function to manipulate the function pointer defined above according to the user input.

4. Define the 4 functions present in the helper.c file as described above.

5. Successfully complete your mission. You should see the message `****Mission: Successful****` for every valid user input to successfully complete the assignment.

Data Structures and Interfaces

You've probably heard of binary search trees before. Provided below is a basic interface for a generic BST implementation.

```
#ifndef BST_H
#define BST_H

typedef struct bst* bst;

bst bst_new();
int insertINTOtheBST(bst tree, int element);
int bst_remove(bst tree, int element);
int is_it_in_the_bst(bst tree, int element);

#endif
```

You notice that this interface ignores several of the good practices that we talked about. The lack of comments, missing free function, and bad naming of several of the functions stand out to you. Fix these issues!

Now that you've commented the interface, you realize that this BST is rather limited, as it can only interact with integers. You should probably change this to make it more generic...

A good place to start would be creating a function pointer type that can return the order of an object, and pass in elements as void*. You will want to pass this sorting function in when the tree is created, so you should modify the `bst_new` constructor. In pseudocode, the sort function should act as follows:

```
sort_function(element a, element b):
    if a < b: return -1
    if a == b: return 0
    if a > b: return 1
```

Now that you have the interface done, create an implementation! Just to inspire you, a sample type for bst follows:

```
struct bst {
    void* element;
    void* left_child;
    void* right_child;
};
```

If you're feeling confident after this, write an implementation that uses a balanced tree. (Hint: a red-black tree is a very efficient for implementing malloc)