Programming Assignment #1

This assignment gives you some practice developing a simple program under Linux. The goal is to implement functions that will create, manipulate, and destroy linked lists. The goal is not to gain any great insight into linked-lists, but the process of coding on an operating system that is new to most students. To keep the complexity to a minimum, the lists will be singly-linked. This is the definition of the node that will be used by the single-linked list known as a *cachelist*:

```
#define LABEL_SIZE 16
typedef struct cl_node
{
  char label[LABEL_SIZE];
  int value;
  struct cl_node* next;
}cl_node;
```

Each node (*cl_node*) in a list contains a text label, an integer, and a pointer to another `cl_node`. This node structure is intentionally kept simple so you can focus on the list aspect of this assignment. This is the interface:

```
cl_node *cl_add_front(cl_node *list, int value, const char *label);
cl_node *cl_add_end(cl_node *list, int value, const char *label);
cl_node *cl_remove(cl_node *list, int search_value);
cl_node *cl_insert_before(cl_node *list, int search_value, int value, const char *label);
void cl_node *cl_insert_after(cl_node *list, int search_value, int value, const char *label);
cl_node *cl_find(cl_node *list, int search_value, bool cache, int *compares);
void cl_destroy(cl_node *list);
void cl_dump(const cl_node *list);
```

Since you will be programming in C, there are no classes, so all of the functions will take a list (a pointer to the first node) as their first parameter. Five of the functions will return the list that was passed in. At first glance, this may seem redundant but it is required as was explained in CS120. The reason is due to the fact that, if a function changes the first node (by an add, insert_before, insert, remove, or find) then the function must return this new "first" node to the caller. We also saw that an alternative approach is for the functions to take a pointer to a pointer to a node instead of returning a node.

Function	Description
`cl_add_front`	Adds a new node to front of the list. If this is the first function that is called, the list will be NULL. Return the head of the list.
`cl_add_end`	Adds a new node to the end of the list. If this is the first function that is called, the list will be NULL. Return the head of the list.
`cl_remove`	Given a value, find the **first** occurrence of it in the list and remove it. Be sure to free the memory that was allocated for the node. If the list does not contain the value, nothing is removed. Return the head of the list.
`cl_insert_before`	Given a value to search for, find the first occurrence of it in the list and insert a new node before it. If the list does not contain the value, nothing is inserted. Return the head of the list.
`cl_insert_after`	Given a value to search for, find the first occurrence of it in the list and insert a new node after it. If the list does not contain the value, nothing is inserted.
`cl_find`	Given a value to search for, find the **first** occurrence. This function is the only one that is slightly more interesting than the others. It will be explained in class and also later in this document. Returns the head of the list.
`cl_destroy`	Walk the list and free all of the nodes.
`cl_dump`	Print out the list. **This is implemented for you. Do not change it.**

The assignment is relatively straight-forward and I don't expect any third-semester Digipen student to struggle with this. This is kind of a practice assignment (however, it will be graded) for you to gain more familiarity with developing under the Linux environment. You will use singly-linked lists in a future assignment, so successful completion of this assignment will help you with that assignment. That future assignment has additional complexities that you will need to focus your attention on, not the linked-list part. So, successfully completing this assignment will give you a head-start on the future assignment.

More on back →

Other criteria

1. When you create helper functions (which you must), they must be tagged as **`static`.** In C++, you would put them either in the private section of a class, or in an unnamed namespace in the implementation file. Neither of those exist in C, so you must use the **`static`** keyword.
2. Be sure to free all of the memory that you allocate. This should be easy for you to verify because you are required to use *valgrind* to look for memory leaks. I will post a command line that you should use.
3. Be sure to check the return values of all of your library calls (e.g. malloc). Failure to do so will result in a lower score, regardless of whether or not the call was successful. Remember, this is a third-semester course and failing to check return values will not be tolerated at all. YOU HAVE BEEN WARNED. Hint: You should only have one place in your code where you call malloc.
4. With the exception of *cl_find*, all of the functions are straight-forward. What makes the find function slightly less obvious are the third and fourth parameters. The fourth (last) parameter is a pointer to an integer and represents the number of comparisons that were performed during the find. To find an item in the list, you must traverse from the front until you find it, counting the number of times you compare values. The farther from the front of the list an item is, the more comparisons that will be performed.
5. The third parameter to the find function is a flag (true/false) named *cache* that will indicate whether or not the found node will be cached. For this assignment, to cache a value in the list simply means that you will *move* it to the front of the list. This means that the list will be modified if an item is found. (The first parameter, *list*, is not marked as **const** because of this.) This caching mechanism can have a profound influence on the performance on the searching of the list. You don't have to understand the details about caching at this point, as I will explain caching many times throughout the semester. This just gives you a little taste of how a slightly modified algorithm can produce enormous benefits.
6. Note that the caching is only done during the find operation. In the Real World™ this would be done during adding/inserting as well.

Deliverables
You must submit the implementation file (`cachelist.c`), the `typescript` file, and the generated `refman.pdf` file. These files must be zipped up and submitted to the appropriate submission area. Notice that you are not submitting the header file, so don't make any changes to that file.

Files	**Description**
`cachelist.c`	The implementation file. All implementation for the functions goes here. You must document the file (file header comments) and all functions (function header comments) using the appropriate Doxygen tags.
`refman.pdf`	This is the PDF documentation that was generated by *Doxygen* and *pdflatex*.
`typescript`	This is the session that was captured with the `script` command.

Usual stuff
Your code must compile (using the compilers specified) to receive credit. The code must be formatted as per the documentation on the website.

Make sure your name and other info is on all documents.