ECE 2230: Computer Systems Engineering **Fall 2020**
Machine Problem 2 **Due: 11:59 pm, Thursday, September 10**

The goal of this machine problem is to build an implementation of a list ADT which includes a set of fundamental procedures to manipulate the list. You will implement the list using two-way linked lists, and define a general-purpose interface that will allow us to use this list ADT for the remainder of the semester. You will then re-write the MP1 honeypot application to utilize the new list ADT.

A key element in this assignment is to develop a well-designed abstraction for the list so that we can return to this interface and replace it with either (a) a different application that needs access to a list or (b) change the underlying mechanism for organizing information in a sorted list (e.g., other than a two-way linked list).

You are to write a C program that will maintain the **two** lists of packet information, one sorted and the other unsorted. The list ADT provides a general list package that is used for both lists. The code **must** consist of the following files:

`lab2.c`	– contains the main() function, menu code for handling simple input and output, and any other functions that are not part of the ADT.
`ids_support.c`	– contains subroutines that support handling of alert records.
`llist.c`	– The two-way linked list ADT. The interface functions must be exactly defined as described below. You can include additional private functions but the interface cannot be changed.
`ids_support.h`	– The data structures for the specific format of the alert records, and the prototype definitions.
`llist.h`	– The data structures and prototype definitions for the list ADT.
`datatypes.h`	– Key definitions of the snort alert structure
`makefile`	– Compiler commands for all code files

Part 1: The two-way linked list ADT

Your program must use a two-way linked list to implement the list ADT. A list is an object, and `llist_construct()` is used to generate a new list. There can be any number of lists active at one time. For a specific list, there is no limit on the maximum size, and elements can be added to the list until the system runs out of memory. Your program must not have a memory leak, that is, at no time should a block of memory that you allocated be inaccessible.

The list ADT is designed so that it does not depend on the type of data that is stored except through a very limited interface. Namely, to allow the list ADT to be used with a specific instance of a data structure, we define the data structure in a header file called `datatypes.h`. This allows the compiler to map a specific definition of a structure to a type we call `data_t`. All of the procedures for the list ADT operate on `data_t`. In future programming assignments we can reuse the list ADT by simply modifying the `datatypes.h` file to change the definition of `data_t` and the comparison procedure prototype, and then recompile.

```
/*
/* datatypes.h
 *
 * The data type that is stored in the list ADT is defined here.  We define a
```

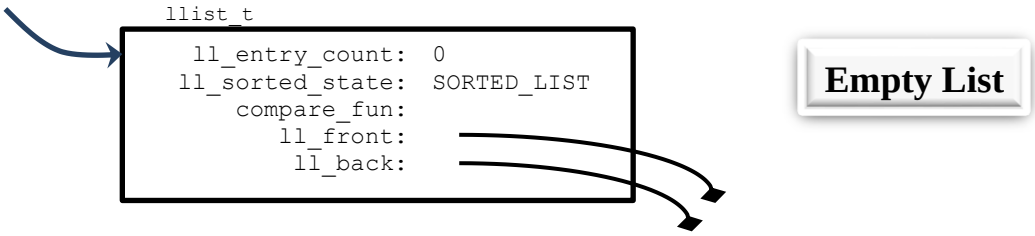
```
 * single mapping that allows the list ADT to be defined in terms of a generic
 * data_t.
 *
 * data_t: The type of data that we want to store in the list
 */
typedef struct alert_tag {
    int generator_id;   // ID of component generating alert
    int signature_id;   // ID of detection rule
    int revision_id;    // revision number of detection rule
    int dest_ip_addr;   // IP address of destination
    int src_ip_addr;    // IP address of source
    int dest_port_num;  // port number at destination
    int src_port_num;   // port number at source host
    int timestamp;      // time in seconds alert received
} alert_t;

/* the list ADT works on alert data of this type */
typedef alert_t data_t;
```

The list ADT must have the following interface, defined in the file `llist.h`.

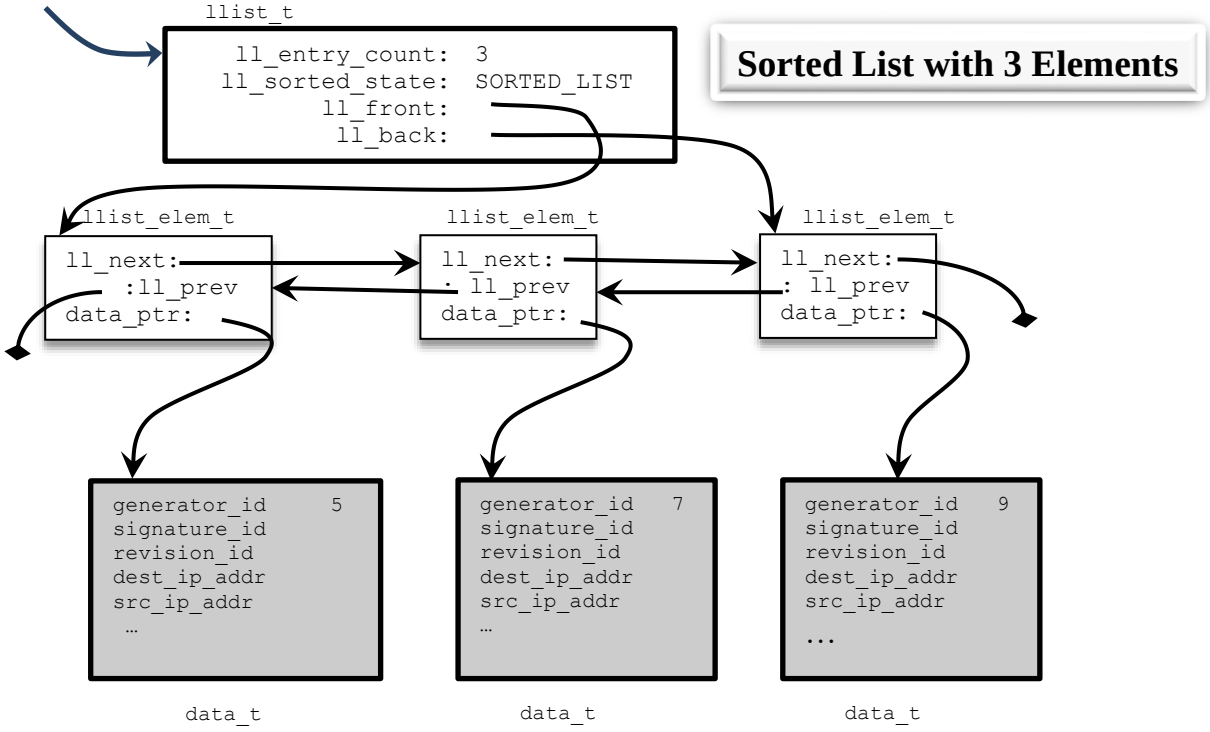
```
/* llist.h
 *
 * You should not need to change any of the code this file.  If you do, you
 * must get permission from the instructor.
 */

#define LLPOSITION_FRONT -987654
#define LLPOSITION_BACK  -234567

typedef struct llist_element_tag {
    // private members for llist.c only
    data_t *data_ptr;
    struct llist_element_tag *ll_prev;
    struct llist_element_tag *ll_next;
} llist_elem_t;

typedef struct llist_header_tag {
    // private members for llist.c only
    llist_elem_t *ll_front;
    llist_elem_t *ll_back;
    int ll_entry_count;
    int ll_sorted_state;
    // Private method for llist.c only
    int (*compare_fun)(const data_t *, const data_t *);
} llist_t;

/* prototype definitions for functions in llist.c */
data_t *  llist_access(llist_t *list_ptr, int pos_index);
llist_t * llist_construct(int (*fcomp)(const data_t *, const data_t *));
void      llist_destruct(llist_t *list_ptr);
data_t *  llist_elem_find(llist_t *list_ptr, data_t *elem_ptr, int *pos_index,
        int (*fcomp)(const data_t *, const data_t *));
int       llist_entries(llist_t *list_ptr);
void      llist_insert(llist_t *list_ptr, data_t *elem_ptr, int pos_index);
void      llist_insert_sorted(llist_t *list_ptr, data_t *elem_ptr);
data_t *  llist_remove(llist_t *list_ptr, int pos_index);
```

The details of the procedures for the list ADT are defined in the comment blocks of the `llist.c` template file. You can define additional *private* procedures to support the list ADT, but you must clearly document them as extensions.

When the list ADT is constructed, it must have the initial form as show in the figure below.



The list ADT supports two variations on a list. A list can be maintained in either sorted or unsorted order, depending on the procedures that are utilized. Multiple lists can be concurrently maintained. Here is an example of the list ADT data structure when three `alert_t`'s have been added to the sorted list.



It is critical that the `llist.c` procedures be designed to not depend on the details of our alert records except through the definitions contained in `datatypes.h`. In particular, the letters "ids_" , and the member names in alert_t *must not* be found in `llist.c`. It is also critical that the internal details of the data structures in the `llist.c` file are kept private and are not accessed outside of the `llist.c` file. The members of the structures `llist_t` and `llist_elem_t` are private and their details *must not* be found in code outside of the `llist.c` file. In particular, the letters "`->data_ptr`", "`->ll_next`", "`->ll_front`", "`->ll_entry_count`", or any variation on "`->ll_`" are considered private to the list ADT and *must not* be found in any `*.c` file except `llist.c`.

Hint: Begin this assignment by completing all of Part 1 only. All work for Part 1 is contained in `llist.c`. Test your `llist.c` module using `driver.c`. Only when Part 1 is completed and tested should you move on to Part 2.

Part 2: The extended functions for alert records

Implement the following procedures in a `ids_support.c` file. The procedures should be implemented using the list ADT as the mechanism to store the records. The following header file defines the example prototype definitions. You ***can*** modify the design in `ids_support.h` (but you cannot modify the design in `llist.h`).

```
/* ids_support.h */

/* prototype function definitions */
* function to compare packet records */
int ids_compare_genid(const alert_t *soc_a, const alert_t *soc_b);
int ids_match_destip(const alert_t *rec_a, const alert_t *rec_b);

/* functions to create and cleanup alert lists */
llist_t *ids_create(const char *);
void ids_cleanup(llist_t *);

/* Functions to get and print packet information */
void ids_print(llist_t *list_ptr, const char *); /* print list of records */
void ids_stats(llist_t *, llist_t *);            /* prints size of each list */

/* functions for sorted list */
void ids_add(llist_t *);
void ids_list_gen(llist_t *, int);
void ids_list_ip(llist_t *, int);
void ids_remove_gen(llist_t *, int);
void ids_remove_ip(llist_t *, int);
void ids_scan(llist_t *, int);

/* functions for unsorted list
 * inserts at the rear, removes at the front,
 * Duplicate packets with the same destination IP address are not
 * permitted.  If a new packet is inserted the old one must be
 * recycled
 */
void ids_add_rear(llist_t *);
void ids_remove_front(llist_t *);
```

Extended user interface for managing packet records

Implement the user functions from MP1 that control a sorted list. These commands should produce identical results compared to MP1. Unlike with MP1, there is no need to specify the size of the initial list. So you ***must not*** read a command line argument when the MP2 program is run.

```
ALERT
LISTGEN gen-id
LISTIP ip-address
DELGEN gen-id
DELIP ip-address
SCANGEN threshold
```

```
PRINT
QUIT
```

In addition add the following four user functions for a *second* list that is *unsorted* and does *not* have a limit on the number of elements in the list. However, there can never be more than one packet from a specific IP address in the list. So, the ADDREAR command first checks if a packet with the same destination ip_address is already in the list. If so, the packet is removed and freed. In any case, the new packet is appended to the end of the list (the new packet is **not** inserted into the position of the old packet). The RMFRONT command simply removes the record at the front of the list, if any.

```
ADDREAR
RMFRONT
PRINTQ
```

STATS prints the size of both the sorted and unsorted lists
QUIT ends the program and cleans up both lists.

Your program maintains two lists, one sorted and one unsorted. The first group of commands operate on the sorted list. The second three commands operate on the unsorted list.

To facilitate grading, the output for each command must be formatted exactly as provided in the supplemental template programs. The `printf()` commands in `ids_support.c` must not be changed.

See the ECE 223 Programming Guide for additional requirements that apply to all programming assignments. All code, a test script, and a test log must be submitted to the ECE assign server. You submit by email to ece_assign@clemson.edu. Use as subject header ECE223-1,#2. The 223-1 identifies you as a member of Section 1 (the only section this semester). The #2 identifies this as the second assignment. When you submit to the assign server, verify that you get an automatically generated confirmation email that does not report an error within a few minutes. If you do not get a confirmation email or the email reports an error, your submission was not successful. You must include all your files as an attachment. The email must be formatted as text only (no html features). You can make more than one submission but we will only grade the final submission. A re-submission must include all files. You cannot add files to an old submission.

To receive credit for this assignment your code must compile and at a minimum evaluate the provided `testinput.txt` and identically match the `expectedoutput.txt`. Code that does not compile or crashes before completing the example input will not be accepted or graded. If your code is not a perfect match you must contact the instructor or TA and fix your code. Code that correctly handles the example input but otherwise fails many other cases will be scored at 50% or less.

See the ECE 223 Programming Guide for additional requirements that apply to all programming assignments.

Work must be completed by each individual student, and see the course syllabus for additional policies.