ECE 2230: Computer Systems Engineering **Fall 2020**
Machine Problem 1 **Due: 11:59 pm, Thursday, August 27**

The goal of this machine problem is to familiarize you with the basics of maintaining information with dynamic memory allocation and pointers. This assignment provides a simple example of data abstraction for organizing information in a sorted list. We will develop a simple abstraction with a few key interfaces and a simple underlying implementation using sequential arrays. In a subsequent programming assignment we will expand upon the interfaces and explore alternative implementations. We will refer to this abstract data type (ADT) as a *sequential list*.

A Honeypot is one strategy used by computer network administrators to monitor and isolate unauthorized use of information systems. One specific mechanism is to locate the honeypot on the router at an ingress connection to the organization's network. The honeypot examines incoming packets and monitors those packets with destination network addresses that are assigned to the organization but are not currently being utilized. A popular tool for intrusion detection is Snort. For example, packets that are destined for unutilized addresses may indicate an attacker is scanning all addresses looking for vulnerabilities.

For this project, we model one portion of the honeypot. Assume another process like Snort examines incoming packets and creates an alert message if a possible threat has been identified. Our project collects key data from the alert records and stores the data in a sorted order. When directed, your program scans the list and determines if there are a significant number of alerts of the same type, and if so prints the records. This supports an analyst that can evaluate these alerts for further testing. For this project we will write the ADT for storing the alert record information, and a function to search for attacks that target specific IP addresses.

You are to write a C program that must consist of the following three files:

lab1.c	– contains the main() function, menu code for handling simple input and output used to test our ADT, and any other functions that are not part of the ADT.
ids.c	– contains the ADT code for our sequential list. The interface functions must be exactly defined as described below.
ids.h	– contains the interface declarations for the ADT (e.g., constants, structure definitions, and prototypes).

Your program must use an array of pointers to C structures that contain information extracted from each record.

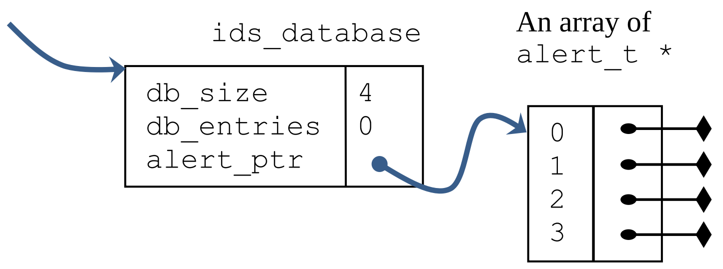
A record of alert information that is stored in your list is represented with a C structure as follows. (You may add additional members to the `ids_database` structure if needed.)

```
struct ids_database {
      int db_size;
      int db_entries;
      struct alert_t **alert_ptr;
};
struct alert_t {
    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
};
```

The sequential list ADT must have the following interface. You cannot change the parameters or add new functions to ids.h:

```
struct ids_database *ids_construct(int);
void ids_destruct(struct ids_database *);
int ids_add(struct ids_database *, struct alert_t *);
struct alert_t *ids_access(struct ids_database *, int);
struct alert_t *ids_remove(struct ids_database *, int);
int ids_empty(struct ids_database *);
int ids_count(struct ids_database *);
int ids_size(struct ids_database *);
void ids_record_fill(struct alert_t *);
void ids_print_rec(struct alert_t *);
```

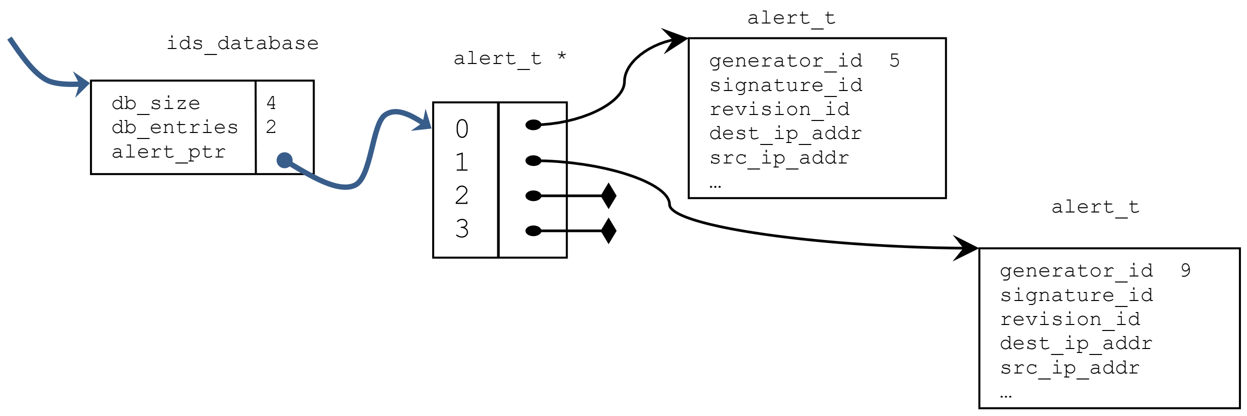
`ids_construct` should return a pointer to a structure that includes an array with size determined by the parameter to this function. Each element in the array is defined as a pointer to a structure of type `alert_t`. Each pointer in the array should be initialized to NULL.



`ids_destruct` should free all `alert_t` memory blocks in the list, free the array of type `alert_t *`, and finally free the memory block of type `ids_database`.

`ids_add` should take a `alert_t` memory block (that is already populated with input information) and insert it into the list such that the list is ordered using the `generator_id` and the list is sequential with no empty gaps between entries in the list. That is, the record with the lowest `generator_id` should be found at index position 0, the next lowest `generator_id` at index position 1, etc. If the list already contains one or more records with the same `generator_id`, the new record should be placed after all the matching records. If the list is full when this function is called then first double the size of the array of `alert_t` pointers (hint: use the `realloc()` library function). If during the function call, the size of the array is doubled, then return 1. Otherwise return 0.

For example, the figure below shows the state of the list after alerts with generator ID 9 and then generator ID 5 were added to the list.



`ids_access` returns a pointer to the `alert_t` memory block that is found in the list index position specified as a parameter. The memory block <u>should</u> <u>not</u> be removed from the list. If the index is out of bounds or if no alert record is found at the index position, the function returns NULL. For example, if the index is 1, the function returns a point to the memory block with `generator_id` 9.

`ids_remove` returns a pointer to the `alert_t` memory block that is found in the list index position specified as a parameter. The record <u>should</u> be removed from the list, and the resulting list should still be sequential with no gaps between entries in the list. If the index is out of bounds or if no record is found at the index position, the function returns NULL. If after a record is removed, the number of entries in the list is less than 20% of the size of the list, then reduce the size of the array in half, returning the excess memory back to the memory heap (hint: use `realloc()`). However, do not allow the size of the array to ever become smaller than the initial size that was defined during the call to `ids_construct`. Hint: store the initial size in the `ids_database` header block.

`ids_empty` returns 1 if the list is empty, and 0 otherwise.
`ids_count` returns the number of entries stored in the sequential list.
`ids_size` returns the size of the database list.
`ids_record_fill` is provided to prompt the user for input data and should not be changed.
`ids_print_rec` is provided to print a record and should not be changed.

The template for the file `lab1.c` provides the framework for input and output and testing the sequential list of alert information. The code reads from standard input the commands listed below. The template contains the only prints to standard output that you are permitted to use. You will expand the template code in `lab1.c` to call functions found in **`ids.c`**. Based on the return information you will call the appropriate print statements. It is critical that you do not change the format of either the input or output as our grading depends on your program reading this exact input pattern and producing exactly the output defined in `lab1.c` and nothing else. These are the input commands:

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

The ALERT command creates a dynamic memory block for the alert_t structure using malloc() and then prompts for each field of the record, one field on each line and in the order listed in the structure. The alert information is added to the sequential list so that the list is sorted based on the generator_id. After the record is inserted use the supplied print to output "Added: x" where x is the Generator ID for the alert. If the list was full when the alert was added then also use the print that includes "doubled list size". The LISTGEN command must print the information for each alert for which the generator_id matches the input value gen-id. Use the ids_print_rec function to print all fields of each matching alert record, and after the last alert use the print in the lab.c file to print the number of records found and the gen-id. Unless no alerts were found, in which case use the print that reports no matching alerts for gen-id were found. The LISTIP command operates the same way except looks for records for which the input value ip-address matches the dest_ip_addr in the alert records. The DELGEN command removes all the corresponding records for which the generator_id matches the input value gen-id from the list, frees the memory, and uses the supplied print to report the number of alerts removed. However, if the item was not found, in which case use the print the reports no matches were found. The DELIP command operates the same way except looks for records for which the input value ip-address matches the dest_ip_addr in the alert records. The SCANGEN command searches the list for all sets of alerts for which there are threshold or more alert records with the same generator_id. For each set, print the generator_id and the number of alerts in the set. After the report for each set is printed, print the number of sets that were found. The PRINT command outputs the count of alerts in the list and the current size of the database followed by the contents of each alert. Finally, the QUIT command frees all the dynamic memory and ends the program.

To facilitate grading the output for each command must be formatted exactly as specified by the **printf()** commands in the lab1.c template file.

Notes

1. The ten ids_* functions **must** be found in the ids.c file, and their function prototypes **must** be listed in the file ids.h. The template code provided has stubs for these functions. Code in lab1.c can call a function defined in ids.c **only** if its prototype is listed in ids.h. You can include additional functions in lab1.c (such as a function to support the SCANGEN command). You can also add other "private" functions to ids.c, however, these private functions can only be called from within other functions in ids.c. The prototypes for your private functions **cannot** be listed in ids.h. Code in lab1.c **cannot** call any private functions in ids.c. Code in lab1.c is **not** permitted to access any of the members in struct ids_database (i.e., db_size or alert_ptr), instead code in lab1.c **must** use the sequential list functions ids_* as defined in ids.h as to **only** way to access details of the list.

Note we are using the principle of *information hiding*: code in lab1.c does not "see" any of the details of the data structure used in ids.c. The only information that lab1.c has about the alert list data structure is found in ids.h (and any "private" functions you add to ids.c are not available to lab1.c). The fact that ids.c uses an array of pointers is unimportant to lab1.c, and if we redesign the data structure no changes are required in lab1.c (including PRINT). However, notice that ids.c does need to read one member of the alert_t structure (i.e., generator_id). If we decide to store different types of records, we must re-write the part of ids.c that uses generator_id. In future machine problems we will study designs that allow us to hide the details of the records from the data structure, so we can reuse the data structure for any type of record.

2. The file `lab1.c` uses the C functions `fgets()` and `sscanf()` to read input data. Do **not** use `scanf()` for any input because it can lead to buffer overflow problems and causes problems with the end-of-line character.

You do not need to check for errors in the information the user is prompted to input for the `alert_t` information. However, you must extensively test your code that it can handle any possible combinations of ALERT, LISTGEN, LISTIP, DELGEN, DELIP, SCANGEN, and PRINT. For example, you code must handle a request to delete, print, or look in an empty list, adding to a full list, and other possible actions on the list.

3. Recall that you compile your code using:

```
        gcc -Wall -g lab1.c ids.c -o lab1
```

Your code must be able to pipe my example test scripts as input using <. Collect output in a file using > For example, to run with a list size of 10 do

```
        ./lab1 10 < testinput.txt > testoutput.txt
```

The code you submit must compile using the `-Wall` flag and **no** compiler errors or warnings should be printed. (OS X users must verify that there are no warnings on a machine running Ubuntu.)

An example `testinput.txt` and `expectedoutput.txt` files are provided. When you run your code on the `testinput.txt` file, you output must be identical to the file `expectedoutput.txt`. You can verify this using

```
        meld testoutput.txt expectedoutput.txt
```

However, the tests in `testinput.txt` are incomplete! You must develop more thorough tests (e.g., attempt to delete from an empty list, or insert into a list that is already full).

Make sure you can use meld. But if there are problems a temporary work around is

```
        diff -w testoutput expectedoutput
```

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.

4. Be sure that your program does not have any memory leaks. That is, all dynamically allocated memory should be freed before the program ends.

We will test for memory leaks with `valgrind`. You execute `valgrind` using

```
        valgrind --leak-check=yes ./lab1 123 < testinput
```

The last line of output from `valgrind` must be:

```
        ERROR SUMMARY: 0 errors from 0 contexts (suppressed: x from y)
```

You can ignore the values *x* and *y* because suppressed errors are not important and are hidden from you. In addition the summary of the memory heap must show

```
        All heap blocks were freed -- no leaks are possible
```

5. All code and a test script that you develop must be submitted to the ECE assign server. You submit by email to ece_assign@clemson.edu. Use as subject header ECE223-1,#1. The 223-1 identifies you as a member of Section 1 (the only section this semester). The #1 identifies this as the first 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.

See the ECE 2230 Programming Guide for additional requirements that apply to all programming assignments.

Work must be completed by each individual student. See the course syllabus for additional policies.