ECE 223: Computer Systems Engineering **Spring 2014**
Machine Problem 1 **Due: 11:59 pm, Thursday, January 16**

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. 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 examines incoming packets and directs those packets that might be associated with an attack to our module. Our project collects key data from the packets and stores the data in a sorted order. When directed, your program scans the list and determines if there are a significant number of packets addressed to the same destination address, and if so prints the records. This is a first indication that there may be an attack and that other parts of the system (which we will not implement) can evaluate these packets for further testing. For this project we will write the ADT for storing the packet information, and a function to evaluate the information for a sign of an attack.

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.
honeypot.c	– contains the ADT code for our sequential list. The interface functions must be exactly defined as described below.
honeypot.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 packet. A key parameter is the size of the array (and thus the number of records that can be stored), and this value is read from the command line as specified below.

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

```
struct honeypot_t {
      int pot_size;
      int pot_entries;
      struct packet_t **hpot_ptr;
};
struct packet_t {
      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 hop_count           /* number of routers in route */
      int protocol;           /* TCP=1, UDP=2, SSL=3, RTP=4 */
      float threat_score;     /* rating of source host */
      int time_received;      /* time in seconds packet received */
};
```

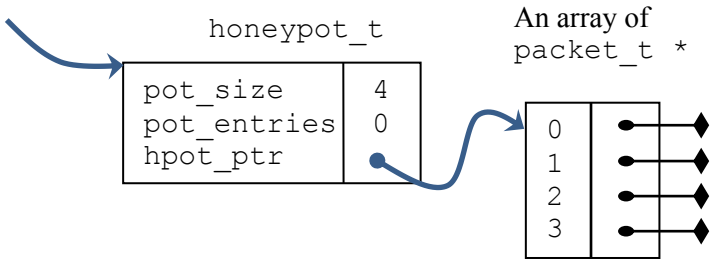
The sequential list ADT must have the following interface:

```
struct honeypot_t *hpot_construct(int);
void hpot_destruct(struct honeypot_t *);
int hpot_add(struct honeypot_t *, struct packet_t *);
int hpot_lookup(struct honeypot_t *, int);
struct packet_t *hpot_access(struct honeypot_t *, int);
struct packet_t *hpot_remove(struct honeypot_t *, int);
int hpot_empty(struct honeypot_t *);
int hpot_count(struct honeypot_t *);
void hpot_record_fill(struct packet_t *);
void hpot_print_rec(struct packet_t *);
```

The above functions **must** be found in the `honeypot.c` file, and their function prototypes **must** be listed in the file `honeypot.h`. Code in `lab1.c` can only call the above functions from `honeypot.c` **only** if its prototype is listed in `honeypot.h`. You can include additional functions in `lab1.c` (such as a function to support the `SCAN` command – see below). You can also add other "private" functions to `honeypot.c`, however, these private functions can only be called from within other functions in `honeypot.c`. The prototypes for your private functions **cannot** be listed in `honeypot.h`. Code in `lab1.c` **cannot** call any of your private functions. Code in `lab1.c` is **not** permitted to access any of the members in `struct honeypot_t` (i.e., `pot_size` or `hpot_ptr`), instead code in `lab1.c` **must** use the sequential list functions `hpot_` as defined in `honeypot.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 `honeypot.c`. The only information that `lab1.c` has about the honeypot list data structure is found in `honeypot.h` (and any "private" functions you add to `honeypot.c` are not available to `lab1.c`). The fact that `honeypot.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 `honeypot.c` does need to read <u>one</u> member of the `packet_t` structure (i.e., `dest_ip_addr`). If we decide to store different types of records, we have to re-write the part of `honeypot.c` that uses `dest_ip_addr`. 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.

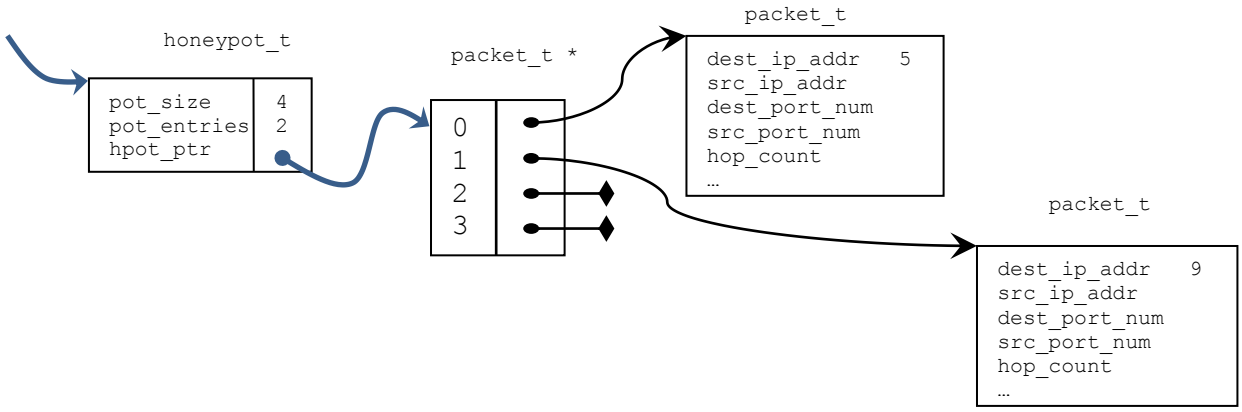
`hpot_construct` should return a pointer to a structure that includes an array with size determined by the parameter to this function. Each element in the array is defined as a pointer to a structure of type `packet_t`. Each pointer in the array should be initialized to NULL.



`hpot_destruct` should free all `packet_t` memory blocks in the list, free the array of type `packet_t *`, and finally free the memory block of type `honeypot_t`.

`hpot_add` should take a `packet_t` memory block (that is already populated with input information) and insert it into the list such that the list is ordered using the `dest_ip_addr` and the list is sequential with no empty gaps between entries in the list. That is, the record with the lowest `dest_ip_addr` should be found at index position 0, the next lowest `dest_ip_addr` at index position 1, etc. If the list already contains one or more packet records with the same `dest_ip_addr`, the new record should be placed after all of the matching records. If the list is full when this function is called then first double the size of the array of `packet_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 packet 9 and then packet 5 were added to the list.



`hpot_lookup` should find the first `packet_t` memory block in the list with the specified `dest_ip_addr` and return the index position of the record within the list. If the `dest_ip_addr` is not found, then return -1.

`hpot_access` should return a pointer to the `packet_t` memory block that is found in the list index position specified as a parameter. The memory block should not be removed from the list. If the index is out of bounds or if no packet record is found at the index position, the function returns NULL.

`hpot_remove` should return a pointer to the `packet_t` memory block that is found in the list index position specified as a parameter. The record should 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 packet 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 `hpot_construct`.

`hpot_empty` should return 1 if the list is empty, and 0 otherwise.
`hpot_count` should return the number of entries stored in the sequential list..
`hpot_record_fill` is provided to prompt the user for input data and should not be changed.
`hpot_print_rec` is provided to print a record and should not be changed.

In the `lab1.c` file you will write a program to test the sequential list of packet information. For purposing of testing our ADT, the program should read from the command line one of these commands:

```
INSERT ip-address
LIST ip-address
REMOVE ip-address
SCAN threshold
PRINT
QUIT
```

The first word of each command must be formatted exactly as shown. You program should be able to handle a line that contains any number of strings. If the line does not exactly match the format of one of the above commands, the line should be simply printed (precede the printing of the line with the character # to indicate the line was not evaluated as a command). The `INSERT` command should create a dynamic memory block for the `packet_t` structure using `malloc()` and then prompt for each field of the record, one field on each line and in the order listed in the structure. If the packet information is added to the sequential list, then print "`Added: x`". If the list was full when the packet was added then also print "`doubled list size`". The `LIST` command must print the information for each packet for which the `dest_ip_addr` matches the `ip-address`. Print all fields of each matching packet record, and after the last packet print "`Found n packets matching x`" where `n` is the number of matching packets. Unless no packets were found, in which case print the message "`Did not find: x`". The `REMOVE` command removes all the corresponding records from the list, frees the memory, and prints "`Removed n packets matching x`" unless the item was not found, in which case it prints "`Did not remove: x`". The `SCAN` command searches the list for all sets of packets for which there are `threshold` or more packet records with the same `dest_ip_addr`. For each set, print the `dest_ip_addr` and the number of packets in the set. After the report for each set is printed, print the number of sets that were found. 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 in the `lab1.c` template file.

Notes

1. 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 < 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
```

2. Use 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. For example:

```
#include <stdio.h>
#define MAXLINE 256

char line[MAXLINE];
char command[MAXLINE];
char restofline[MAXLINE];
int num_items;
int ip_add;

while (fgets(line, MAXLINE, stdin) != NULL) {
       num_items = sscanf(line, "%s %d %s", command, &ip_add, restofline);
       if (num_items == 1 && strcmp(command, "QUIT") == 0) {
              /* found exit */
       } else if (num_items == 2) {
              /* more tests on command */
       } else { /* did not match any other test */
              printf("# %s", line);
       }
}
```

You do not need to check for errors in the information the user is prompted to input for the `packet_t` information. However, you must extensively test your code that it can handle any possible combinations of INSERT, LIST, REMOVE, SCAN, 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 honeypot.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 > testoutput
```

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 they are using the latest version of gcc or make a final check on a machine running Ubuntu.)

4. All code, a test script, and a test log must be turned in by email to assign@assign.ece.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. The email program you use to send your code needs to use your Clemson address as the "from email address" (and not your google address). WebMail works well. When you submit to the assign server, verify that you get an automatically

generated confirmation email within a few minutes. If you don't get a confirmation email, your submission was not successful.

5. Turn in a paper copy of all files at the start of the first class meeting following your submission to the assign server.

See the ECE 223 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.