Pset 3: Ahoy Matey! Part B

AKA Hookbook B

Due Friday, Feb 23 2024

Ahoy, matey! Let's extend your Hookbook from Hookbook A. In this assignment, you will be practicing **file I/O operations**, **structs**, **dynamic memory allocation**, **linked lists**, **sorting**, **function pointers**, and **software engineering** in C.

The Assignment

In Hookbook A, you created an application that reads in a list of pirate names and sorts them before printing out their names as a list.

In this document, your swashbuckling clientele has finally concluded their market research and now they are ready for full profiles for the pirates in their Hookbook. The information they need you to store includes: * Their current vessel of employ * Their current captain (who is another pirate!) * The number of treasures they've found * Their favorite port of call * All of the pirating skills they possess

As in the previous assignment, your job will be to display each pirate's profile in sorted order, this time with the full complement of profile information.

There are two additional requirements for this assignment, however.### Background: The `const` Type Qualifier

The `const` type qualifier in C identifies to the compiler that the object's value may not be changed. This has several implications, not all of which are straightforward.

For non-pointer variables, qualifying their type with `const` makes it impossible to modify their value. Directly attempting to modify the value of a variable with `const`-qualified type results in a compiler error; *indirect* modification attempts are undefined behavior.

In C, it is possible to explicitly cast a `const`-qualified object to a non-`const` object: this is a bad idea. Going the other way—casting a non-`const` object to `const`—is **implicit** and always safe.

For pointer types, the `const` qualifier refers to the object the pointer refers to, rather than the pointer itself. Because of this, you should avoid allocating `const`-qualified memory. Remember, `malloc` allocates memory but does not initialize it—as you can see below, it is an error to modify that memory if the pointer variable is `const`, thus rendering that memory essentially useless! Further, the `free` function expects a non-`const` pointer as its argument—you cannot free `const` memory!

A pointer to `const` should instead be initialized by *assigning* it to another (already-initialized) object. The restricts modification of the object only via access by the `const`-qualified pointer; the object can still be modified through access from the other (non-`const`) pointer. Most commonly, this happens when functions are declared with parameters having `const`-qualified types. Remember: in C, parameter passing to a function is nearly indistinguishable from assignment.

To declare a pointer variable *itself* as `const` (e.g., to force it always to point to the same address in memory), the `const` keyword should appear *after* the `*` pointer specifier. This is (perhaps surprisingly) an uncommon thing to do, particularly for the kinds of programs we wil be writing in this course.

See an example of several `const` and non-`const` variables below. Play around with it in `const_demo.c` in the starter code. (Yes, there are memory leaks in this example program.)

```c
int main(void)
{
    int x = 0;
    const int y = 0;
    int *p = (int*)&y;
```

```
    x = 42;  // OK: x is not const
    y = 42;  // ERROR: y is const
    *p = 42; // Undefined behavior

    int *a = malloc(sizeof(int));
    const int *b = malloc(sizeof(int));
    int *const c = malloc(sizeof(int));
    const int *const d = malloc(sizeof(int));

    *a = 42; // OK: a is not const
    *b = 42; // ERROR: *b is const
    b = a;   // OK: b is not const; implicit cast of *a (non-const to const)
    *c = 42; // OK: *c is not const
    c = b;   // ERROR: c is const
    *d = 42; // ERROR: *d is const
    d = a;   // ERROR: d is const
}
```

Variables with `const`-qualified type declarations are not necessarily constants in the same sense as constants defined by preprocessor `#define` directives. Whenever a value is required to be named as a constant in this document (and in this course, broadly speaking), you should use a `#define` directive unless specifically instructed otherwise in the document.

Background: The `restrict` Type Qualifier

Some of the functions you implement have parameters declared with the `restrict` type qualifier. This type qualifier is a hint o the compiler that for the life of this pointer, no other pointers will access the memory it is associated with. For this assignment, **there is nothing in particular you should do** to handle `restrict`-qualified pointers. It is needed to make the C++/C interoperability happy when we compile unit tests for your submission.

Files you are given

Files are reproduced here for your convenience. If there are any differences in these files and those in the starter code, defer to the version in the starter code.

- `pirate.h` (Interface), with the following (partial) type definition. You must complete the task marked by `TODO`. **You may not change anything else already in this file**, however you may add functions or other definitions to it if you like. “‘C /**
  - Type of a pirate, as a (non-opaque) struct */ typedef struct pirate_t pirate; struct pirate_t { char* name; // TODO: Expand this type definition };

  /**
  - Allocates enough memory for a pirate, and sets the pirate’s name to name.
  - Ownership of the name pointer is transferred to the pirate. Returns a
  - pointer to the newly-allocated pirate.
  -
  - @return a pointer to a new pirate with given name */ *pirate* pirate_create(char *name);

  /**
  - Reads a pirate profile from input. Assumes that the input cursor is at the
  - beginning of a pirate profile (otherwise, the behavior is undefined).
  -
  - Returns a pointer to a pirate created from the profile in input.

```
 –
 – This function consumes the blank line after the end of the current profile
 – before returning (that is, the first newline character and the second, if
 – there is one, or the EOF marker if not), meaning that after this function
 – is done, the input cursor is either at the end of the input file or at the
 – beginning of the next pirate profile.
 –
 – @param input the stream from which to read a pirate profile
 – @returns a new pirate based on the contents of input
 – @assumes input is an open and readable stream
 – @assumes the next character in input is the first character of a pirate's
 – profile, and that input is well-formed according to the input specification
 – in the README. / pirate pirate_read(FILE *restrict input);

/**
 – Prints a pirate's profile to output in conformance with the output
 – specification in the README.
 –
 – @param p the pirate to print
 – @param output the stream to which the pirate's profile should be printed
 – @does prints p's profile to output
 – @assumes p is not NULL
 – @assumes output is an open, writable stream / void pirate_print(const pirate p, FILE *restrict
   output);

/**
 – Compares two pirates, returning a negative number, zero, or a positive
 – number as pirate a "comes before", is equal to, or "comes after" pirate b.
 – For this function, we say a "comes before" b if the name of pirate a is
 – lexicographically less than the name of pirate b, and that they are equal
 – if they have the same name.
 –
 – A pirate with no name always comes after a pirate with a name. No order is
 – defined between two pirates with no name.
 –
 – @param a the first pirate
 – @param b the second pirate
 – @return a negative number, zero, or a positive number as a "comes before",
 – is equal to, or "comes after" b according to their names
 – @assumes neither a nor b are NULL / int pirate_compare_name(const pirate a, const pirate *b);

/**
 – Compares two pirates, returning a negative number, zero, or a positive
 – number as pirate a "comes before", is equal to, or "comes after" pirate b.
 – For this function, we say a "comes before" b if the vessel of pirate a is
 – lexicographically less than the vessel of pirate b, and that they are equal
 – if they have the same vessel.
 –
 – A pirate with no vessel always comes after a pirate with a vessel. Two
 – pirates with no vessels compare identically to pirate_compare_name(a,b)
 –
 – @param a the first pirate
 – @param b the second pirate
 – @return a negative number, zero, or a positive number as a "comes before",
```

– is equal to, or “comes after” b according to their vessels
– @assumes neither a nor b are NULL */ *int pirate_compare_vessel(const pirate* a, const pirate *b);

/**

– Compares two pirates, returning a negative number, zero, or a positive
– number as pirate a “comes before”, is equal to, or “comes after” pirate b.
– For this function, we say a “comes before” b if the treasure of pirate a is
– greater than the treasure of pirate b, and that they are equal if they have
– the same treasure.
–
– A pirate without a specified treasure always comes after a pirate with a
– specified treasure, even if that specified treasure is 0. Two pirates
– without specified treasure compare identically to pirate_compare_name(a,b)
–
– @param a the first pirate
– @param b the second pirate
– @return a negative number, zero, or a positive number as a “comes before”,
– is equal to, or “comes after” b according to their treasures
– @assumes neither a nor b are NULL */ *int pirate_compare_treasure(const pirate* a, const pirate *b);

/**

– Frees all memory owned by p. This function claims ownership of p.
–
– @param p the pirate to destroy
– @does frees all memory owned by p
– @assumes p is not NULL */ *void pirate_destroy(pirate* p); “‘

Note: The `pirate_t` struct includes a declaration of a single field, `char *name`. This must remain in your submission unchanged because the unit tests rely on the presence of a field with this name and type. The remainder of the `pirate_t` type definition is up to you.

Note: The `pirate` interface for Hookbook B defines a strict superset of the behavior of the `pirate` interface for Hookbook A. Therefore, you could implement only this version of the interface and use it for *both* Hookbook assignments!

- `pirate_list.h` (Interface), with the following declarations. **You may not change anything in this file**, however you may add functions or other definitions to it if you like.

```
/**
* Type of a comparison function (defined in starter/libhookbook.h)
*/
typedef int (*compare_fn)(const pirate *, const pirate *);

/**
* Type of a list of pirates, as an opaque struct.
*/
typedef struct pirate_list_instance_t pirate_list;

/**
* Allocates memory for a new pirate_list and returns a pointer to it. Sets the
*  comparison function for the new list to pirate_compare_name.
*
* @return a pointer to a newly-created pirate list that can be sorted by
*  pirate names.
*/
```

```
pirate_list *list_create();

/**
* Allocates memory for a new pirate_list and returns a pointer to it. Sets the
*  comparison function for the new list to cmp.
*
* @param cmp the comparison function to use when this list is sorted
* @return a pointer to a newly-created pirate list that can be sorted
*  according to the order computed by cmp
*/
pirate_list *list_create_with_cmp(compare_fn cmp);

/**
* Return the index in pirates of the pirate with the name, or a value greater
*  than or equal to the length of the list if the there is no such pirate.
*
* @param pirates the list to search
* @param name the name to search for
* @return the index of the pirate with name, or a value greater than or equal
*  to the length of the list if the there is no such pirate.
* @assumes neither pirates nor p are NULL
*/
size_t list_index_of(const pirate_list *pirates, const char *name);

/**
* Only if there is no pirate in the list with the same name as p, insert
*  pirate p into the list at index idx by copying the pointer, shifting the
*  latter part of the list one "slot" to the right.
*
* If idx = [length of pirates], then this function appends p to the end of
*  the list.
*
* If there is a pirate in the list with the same name as p, do nothing, and
*  return p (the pirate that was not inserted).
*
* If the pirate was inserted into the list, return NULL.
*
* If p is added to the list, the list claims ownership of the pirate pointed
*  to by p; otherwise ownership remains with p.
*
* @param pirates the list in which to insert p
* @param p the pirate to insert
* @param idx the index at which to insert p
* @return NULL or p, depending on whether p was successfully inserted
* @assumes neither pirates nor p are NULL, and that p has a name
* @assumes 0 <= idx <= [length of pirates]
*/
pirate *list_insert(pirate_list *pirates, pirate *p, size_t idx);

/**
* Remove the pirate from the list with name, and return a pointer to it.
*
* If name is NULL or there is no pirate in the list with name, return NULL.
```

```
*
* If this function does not return NULL, then the caller owns the returned
*  pirate.
*
* @param pirates the list from which to remove a pirate
* @param name the name of the pirate to remove
* @return the removed pirate, or NULL if there is no matching pirate in the
*  list
* @assumes pirates is not NULL
* @assumes every pirate in pirates has a name that is unique in the list
*/
pirate *list_remove(pirate_list *pirates, const char *name);

/**
* Return a pointer to the pirate pointed to by index idx in the list, or NULL
*  if idx is not a valid index (i.e., it is >= the length of the list).
*
* Ownership of the returned pirate remains with the list.
*
* @param pirates the list to access
* @param idx the index to access of the list
* @return a pointer to the pirate at index idx of pirates, or NULL
* @assumes pirates is not NULL
*/
const pirate *list_access(const pirate_list *pirates, size_t idx);

/**
* Sort the list of pirates in the order defined by the comparison function
*  for pirates, as previously set by list_set_cmp(pirates).
*
* @param pirates the list to sort
* @does sorts the list of pirates in the order defined by the comparison
*  function of pirates
* @assumes The comparison function for pirates exists
* @assumes pirates is not NULL and every pirate in the list has a name that is
*  unique in the list
*/
void list_sort(pirate_list *pirates);

/**
* Return the number of pirates in the list.
*
* @param pirates the list
* @return the number of pirates in pirates
* @assumes pirates is not NULL
*/
size_t list_length(const pirate_list *pirates);

/**
* Free all memory owned by the list, including the pirates in the list.
*
* @param pirates the list to destroy
*/
```

```
void list_destroy(pirate_list *pirates);
```

> **Note**: The `pirate_list` interface for Hookbook B defines a strict superset of the behavior of the `pirate_list` interface for Hookbook A. Therefore, you could implement only this version of the interface and use it for *both* Hookbook assignments!

- `libhookbook.h` (Interface) which contains the following definitions. **You may not change anything already in this file**, however you may add functions or other definitions to it if you like. “‘C /**
  - Type of a comparison function. */ typedef int (*compare_fn)(const void *, const void *);

  /**
  - Reads the next line of input from stream, discarding the trailing newline
  - character if there is one, and stores the line in the array pointed to by
  - str. If the next line of input is longer than count characters, this reads
  - and stores the next count characters from stream. In either case, str ends
  - with ‘\0’.
  -
  - If stream is at EOF when the function begins, returns NULL and leaves str
  - unchanged.
  -
  - @param str the array in which to store the characters
  - @param count the maximum number of meaningful characters to store in str
  - @param stream the file from which to read
  - @does read the next line of input from stream, discarding the trailing
  - newline character if there is one, and store those characters in str.
  - @return str, or NULL if stream is at EOF when the function begins
  - @assumes str points to enough memory to hold count+1 chars
  - @assumes stream is open and readable */ char *freadln(char *str, int count, FILE *stream); “‘
- `makefile` (Incomplete)
  - The provided makefile defines two variables, `CC` and `CFLAGS`. You must submit a makefile having identical values for those two variables: `CC = gcc` `CFLAGS = -std=c17 -Wall -pedantic -g`
  - The first target, `Hookbook`, is incomplete and you must complete it, however, you must not change its name, and it must generate an executable file named “`Hookbook`”.
  - The rest of `makefile` is up to you.
- `skills_list.h` (Interface), with the following type definition. **You may not change anything already in this file**, however you may add functions or other definitions to it if you like. This interface is your chance to flex your interface-designing creativity. Consider what you need to do with a list of skills, and define and implement functions that enable you to accomplish those tasks. A good interface design would include 5-10 function prototypes in this file. As you design the interface, consider whether you need to…
  - access a particular skill by name?
  - access a skill at a particular index?
  - remove a skill from the list?
  - handle several skills with the same name?
  - implement a custom type to track additional data about a skill in the list, or will a built-in type suffice?
  - put the skills in a particular order
  - print the list of skills?

  We suggest you use the various list-like interfaces used in class for inspiration. “‘C /**

- Type of a list of pirate skills, implemented as a linked list. */ typedef struct skills_list_instance_t skills_list; "'

Files you must complete

The starter code contains stubs/skeletons for the following files.

- `hookbookb.c` (Driver)
  - This will be the file where your `main` function is going to be. Your `main` function must:
    1. Take three command-line arguments: the path to a file containing the pirates' profiles, the path to a file containing pirate/captain pairs, and the sort order.
    2. Open the profiles file and read from it the list of pirate profiles, appearing as specified below, storing them in a `pirate_list*`
    3. Open the captains file and read from it the list of pirate/captain pairs, appearing as specified below
    4. Sort the list in the order defined by the third command-line argument
    5. Print the sorted list to `stdout`, conforming to the format described below in Output Format
    6. Release all resources (files, memory, *etc.*)
- `skills_list.c` (Implementation), containing a definition for every declaration or prototype in `skills_list.h`. **You are required to implement the list of skills as a singly-linked list.**
  - A linked list is arguably not the best tool for this job. We encourage you to think about what would be a better option. (But you are still required to implement a linked list.)
- `pirate.c` (Implementation), containing a definition for every declaration or prototype in `pirate.h`
- `pirate_list.c` (Implementation), containing a body for every function prototype in `pirate_list.h`, and a definition for `struct pirate_list_instance_t`
  - `pirate_list` must be implemented as an **array list**, *a.k.a.* **resizing array**
  - The initial capacity of the list must be defined as a constant named `INITIAL_CAPACITY`, having value `25`
  - The capacity of the list must never fall below `INITIAL_CAPACITY`
  - When the array capacity changes, it must change by a *factor* of `2`, which must be the value of a constant named `RESIZE_FACTOR`
  - Your array list implementation must avoid the oscillation problem discussed in class
  - In addition to functions declared in `pirate_list.h`, `pirate_list.c` must contain a declaration and implementation for each of the following two "helper" functions that are not declared in the header file: "'C /**
    * Check if the actual number of pirates in the array is "too large"; if it is,
    * increase the capacity of the array by a factor of RESIZE_FACTOR.
    *
    * Given N actual pirates in the array of capacity C, we say N is "too large"
    * whenever N >= C
    *
    * If the array capacity was changed, print to stderr the string "Expand to",
    * followed by the new capacity of the list and a newline. Here is a possible
    * print statement:
    *
    * `fprintf(stderr, "Expand to %zu\n", new_capacity);`
    *
    * If the capacity was not changed, print nothing.
    *
    * @param pirates the pirate list to expand, if necessary
    * @does increases the capacity of the array if there are too many pirates in
    * it
    * @assumes the pirate list is in the process of having a pirate added to it. */ *void list_expand_if_necessary(pirate_list* pirates);*
    /**

* Check if the actual number of pirates in the array is "too small"; if it is,
* decrease the capacity of the array by a factor of RESIZE_FACTOR.
*
* Given N actual pirates in the array of capacity C, we say N is "too small"
* whenever C > INITIAL_CAPACITY and N * RESIZE_FACTOR <= C / RESIZE_FACTOR
*
* If the array capacity was changed, print to stderr the string "Contract to"
* followed by the new capacity of the list. Here is a possible print
* statement:
*
* `fprintf(stderr, Contract to %zu\n, new_capacity);`
*
* If the capacity was not changed, print nothing.
*
* The capacity of the array must never fall below INITIAL_CAPACITY!
*
* @param pirates the pirate list to contract, if necessary
* @does decreases the capacity of the array if there are too few pirates in
* it.
* @assumes the pirate list is in the process of having a pirate added to it. */ *void list_contract_if_necessary(pirate_list* pirates); "'

- `libhookbook.c` (Implementation), containing a definition for every declaration or prototype in `libhookbook.h`

Other requirements

- The implementation of `skills_list` must be as a singly-linked list.
- Each header and source file must use `#include` directives to include things that are *directly* used by that file—**no more, no less**. Points will be deducted for clearly extraneous `#include` directives or directives that rely on transitive inclusions except between an implementation file and its corresponding header.
- You may not use the standard library's `qsort` function to sort anything, nor any other pre-implemented sorting function. You **must** implement every sorting algorithm that you use on your own
  - Sorting a `pirate_list` must take $O(n \log n)$ time and $O(\log n)$ additional memory regardless of the command-line sort flag
  - There is no required bound on sorting a `skills_list`

  Hint: take advantage of this relaxed efficiency requirement for `skills_list` as an opportunity to write code that is particularly easy to understand and reason about, and think about ways in which you can avoid a separate `sort` function altogether—perhaps by maintaining a sorted order as you insert each skill or by selecting the skills in order from an unsorted list when it comes time to print them.
- You must create *at most one* pirate for each profile in the input file—you may not create copies of pirates (copies of *pointers* to pirates is OK).
  - In other words, your program must use $O(n)$ memory for an profiles file with n pirates
  - The size of the captains file is unbounded relative to the profiles file; your program must use a constant amount of memory for each line in that file.
- It is possible that you will not use every function you implemented... but you still must implement every function identified in this README! We will run **unit tests** on your program, meaning we will test—in isolation—each of the functions identified in this document. For example, you can efficiently implement a solution to this assignment without using the `list_remove` function, although we will run

test cases that call `list_remove` on contrived input values and check that your implementation of that function conforms to its specification.

Here is a possible unit test for `list_remove` (note: this test case *does not* test `pirate_destroy` or `list_destroy`, as the the assertions are made before those functions are called; nor does it test the functionality of `list_create`, `pirate_create`, `list_insert`, or `list_length`—rather, it *assumes* those functions are correctly implemented):

```
/**
 * Tests list_remove for a list with one pirate in it.
 */
void test_remove_1()
{
    pirate_list* p = list_create();
    pirate* r = pirate_create("Foo");
    list_insert(p, r, 0);
    pirate* r2 = list_remove(p, r);
    if (list_length(p) == 0 && r2 != NULL && strcmp(r2->name, "Foo") == 0)
    {
        printf("test_remove_1 passed!\n");
    }
    else
    {
        printf("test_remove_1 failed!\n");
    }
    pirate_destroy(r2);
    list_destroy(p);
}
```

It could be called by a `main` function that does nothing but calls a bunch of similar unit test functions.

- Respond to the weekly surveys, with at least the following information
  - Your full name and netID
  - Your estimate of the time required to complete this assignment, selected *before* you start writing any code.
  - The total time you *actually* spent on the assignment
  - The names of all others (but not members of the teaching staff) with whom you discussed the assignment for more than 10 minutes
  - A brief discussion (~100 words) of the major conceptual and coding difficulties that you encountered in developing and debugging the program (and there will always be some).Your discussion in one of the weekly surveys (one of the surveys released on Feb 9, 16, or 23) must also address the following:
    * Suppose we asked you to allow a user to enter a pirate's name and see information about them. What might be a better (*i.e.*, more efficient) data structure than an array list to help you accomplish this task? Why?
    * What might a better (*i.e.*, more efficient) data structure be for a `skills_list` than a singly-linked list? Why?

      Important: There are specific prompts you must address in your discussion for one of the weekly surveys during this assignment:
      * Suppose we asked you to allow a user to enter a pirate's name and see information about them. What might be a better data structure than an array list to help you accomplish this task? Why?
      * What might a better data structure be for a `skills_list` than a singly-linked list? Why?

Memory Management

Now that you're managing memory yourself, we will test your submission with `valgrind`, a tool to help identify memory leaks. When run on valid input, **for full credit, your submission must produce a valgrind report with no memory leaks, no errors, and no warnings**.

> **Note**: Valgrind is not available for the most recent several generations of Apple computers, and its support level for modern Windows computers is iffy. It is however available, stable, and configured for use on the Zoo. This is yet another reason it is important to run your programs on the Zoo before submitting them (an easy way to make sure you do this is to develop them entirely on the Zoo).

Suggestions and Notes

- Start this assignment by copying your implementation of `pirate_list`, `pirate`, and `libhookbook` from your solution for Hookbook A.
  - You will, of course, have to make some changes to your solution for Hookbook A (otherwise, this would not be a very good assignment!) but at least some of it should be straightforward to reuse.
- Factor out the functionality of `main` into several well-named, easily-understood functions that you define in `libhookbook`.
- To help manage `#include` directives, draw a "map" of your project's files, with arrows marking *direct* dependencies, and use that to write your include directives
- `NULL` is a special pointer value that means "points to nothing". Attempting to dereference `NULL` will probably result in a segmentation fault, which your program must avoid, although it is technically undefined behavior (which your program also must avoid).
- If you find yourself struggling to accomplish a task in the course of your development, break that task down into smaller steps—chances are you'll know how to do some of the smaller steps!—and keep breaking it down into smaller and smaller steps that you know how to tackle. Then, compose those smaller steps into a function that solves your original problem.
  - Do not use ChatGPT or CoPilot for help. Generative AI is not permitted as a tool in this course.

Input Format

The input to the program is three command-line arguments, which represent—not necessarily in order—the name of the profiles file, the name of the captains file, and a flag indicating the method by which to sort the output. The format of the command-line arguments as follows:

- The sort method flag may appear anywhere in the command-line arguments, or not appear at all.
- The two non-sort-flag arguments are to be interpreted in left-to-right order: the first-appearing one is the list of pirate *profiles* and the second-appearing one is the list of pirates and their *captains*.

If present, the sort flag argument should be one of "`-n`", "`-v`", or "`-t`". Their meaning is as follows: * `-n`: sort the output in **ascending** order by the pirates' names * `-v`: sort the output in **ascending** order by the pirates' vessels * `-t`: sort the output in **descending** order by the pirates' treasures

If there is no sort flag present, the output should be sorted in **ascending** order by the pirates' names (as if the `-n` sort flag was provided). Your sorting algorithm must break ties by sorting tied pirates in ascending order of their names.

The profiles file is the organized as described in the previous assignment:

- The first line of the file is the name of a pirate, which may be empty
- The following consecutive non-empty lines hold pirate information, each of which is labeled with the kind of information it provides. Each line is one of the following:
  - "`r:[rank-name]`", where `[rank-name]` is the name of this pirate's rank on their vessel, such as "captain", "first mate", or "sailor"
  - "`v:[vessel-name]`", where `[vessel-name]` is the name of this pirate's current vessel of employ
  - "`p:[port-name]`", where `[port-name]` is the name of this pirate's favorite port of call

  - "`t:[treasure-amount]`", where `[treasure-amount]` is a nonnegative integer that counts the number of treasure pieces this pirate owns
  - "`s:[skill-name]`" where `[skill-name]` is a skill that a pirate might possess, such as "swashbuckling", "treasure hunting", or "navigating"
- Each pirate profile in the input is followed by exactly one empty line (the last profile in the file is followed by a *single* newline character; the last empty line of the file is terminated by the end of the file, not a second newline)

The captains file contains pirate/captain pairs, with one name per line, organized as follows:

```
underling-name-1
captain-name-1
underling-name-2
captain-name-2
pirate-name-3
captain-name-3
...
```

There are some features of the input files that are important:

- The name of either file does not begin with a hyphen (`'-'`)
- No line in either file is longer than 127 characters
- No pirate name begins or ends with whitespace.
- In the profiles file, the pirate's name will be on the first line of their profile, but the rest of the profile information can appear in any order, or may not be present at all
- There may be several pirates with the same name. In this case, only the *first* pirate with a given name should be persisted. Put another way, *in the generated* ***`pirate_list`****, every pirate should have a unique name.*
- There may be lines with repeated labels, even within a profile. Except in the case of repeated skills, only the *last* occurrence of a label in a profile should be persisted. (Repeated skills should *all* be persisted, with duplicate skills interpreted as increased proficiency by the pirate.)
- Not every pirate profile will contain a line for every piece of information; your method of storing pirate information must account for this possibility.
  - If a pirate does not have a line for the piece of information by which the user has requested the program to sort, then that pirate should appear **at the end** of the sorted list and multiple such pirates should be sorted in ascending order by name (just as if they were tied according to the initial sort parameter).
- In the captains file, every name is also a name of a pirate with a profile from the profiles file.
  - Beyond their mere existence, you may make no assumptions about the nature of a pirate and their captain:
    * It may be the case that one pirate's captain has a captain that is another pirate
    * It may be the case that one pirate's captain has as a captain the first pirate
    * It may be the case that a pirate is their own captain
    * A pirate and their captain need not be sailors on the same vessel
    * The "rank" of a captain need not be "captain"
    * Not every pirate with a profile will appear in the captains file

Output Format

The output from your program must be a list of pirate profiles, sorted in the order defined by the user at the command line. Each profile must begin with the pirate's name, which must be followed for a line identifying each additional piece of information you have available for this pirate, but *no* lines for pieces of information for which there was nothing in the profile.

The order in which the fields must be displayed is (note the capitalization): * "Captain" * "Rank" * "Vessel" * "Port" * "Treasure" * "Skills"

Additional pieces of information must appear each on their own line, indented by **4** spaces and beginning with the kind of information displayed ("Captain", "Rank", "Vessel", *etc.*) followed by a colon and a space (`": "`). In the case of a pirate with multiple skills, the label "Skills" should appear only once, with each *unique* skill on its own line, indented by **12** spaces (the first skill must appear on the same line as the "Skills" label). Skills must be sorted in ascending lexicographic order of their name, and skills must be followed on their line by a number of asterisks (`'*'`) equal to that skill's "rating" for the pirate (*i.e.*, the number of times that skill appeared in the pirate's profile in the input) minus 1. (See example 1 below for further clarification.)

If a pirate has a captain, is must be displayed in the output as follows:

```
Captain: <captain-name> (<captain-vessel>)
```

Examples

Here are several example runs of the program showing the output both to `stdout` and to `stderr`.

1. File with three pirates > File: `A_list.txt` > `text` > `jack sparrow` > `r:captain` > `v:the black pearl` > `p:tortuga` > `t:12` > > `william turner` > `p:port royal` > `s:smithing` > `s:smithing` > `s:fencing` > `s:fencing` > `s:fencing` > `s:fencing` > > `elizabeth swan` > `r:queen` > `t:367` > `v:the flying dutchman` > `p:port royal` > `s:leadership` > `s:subterfuge` > >

   File: `A_captains.txt`

   ```
   william turner
   jack sparrow
   jack sparrow
   elizabeth swan
   ```

   Command and output:

   ```
   $ ./Hookbook A_list.txt A_captains.txt -v
   jack sparrow
       Captain: elizabeth swan (the flying dutchman)
       Rank: captain
       Vessel: the black pearl
       Port: tortuga
       Treasure: 12
   elizabeth swan
       Rank: queen
       Vessel: the flying dutchman
       Port: port royal
       Treasure: 367
       Skills: leadership
               subterfuge
   william turner
       Captain: jack sparrow (the black pearl)
       Port: port royal
       Skills: fencing ***
               smithing *
   ```

2. File with 30 pirates, names only, and no captains > File: `B_list.txt` > `text` > `captain mansel alcantra` > > `john alexander` > > `algerine pirates` > > `ali basha` > > `captain alleston` > > `alwilda` > > `alexander ammand` > > `ibrahim ameer` > > `captain cornelius anderson` > > `captain androeas` > > `sultan of timor angora` > > `angria` > > `captain thomas anstis` > > `antonio` > > `john rose archer`

```
>     > james austin     >     > captain john avery     >     > captain bridgeman
>     > captain aylett     >     > job baily     >     > captain baker     >     >
roger ball     >     > john ballet     >     > captain baltizar     >     > captain
bannister     >     > barbarosa     >     > redbeard     >     > captain nicholas
barbe     >     > captain barnard     >     > captain barnes     >     >
```

File `B_captains.txt` is empty.

Command and output (note the line reading, "Expand to 50"!):

```
$ ./Hookbook B_list.txt B_captains.txt -n
Expand to 50
alexander ammand
algerine pirates
ali basha
alwilda
angria
antonio
barbarosa
captain alleston
captain androeas
captain aylett
captain baker
captain baltizar
captain bannister
captain barnard
captain barnes
captain bridgeman
captain cornelius anderson
captain john avery
captain mansel alcantra
captain nicholas barbe
captain thomas anstis
ibrahim ameer
james austin
job baily
john alexander
john ballet
john rose archer
redbeard
roger ball
sultan of timor angora
```

Assumptions

You may assume the following about the command-line arguments to the program:

- **Nothing!** In all cases where the command-line arguments are not as specified in the Input Format, your program must print a descriptive error message to `stderr` and exit with status code `1`.

To give you a better sense of the kinds of things you must handle, here are some *valid* commands for running your program, which must be handled correctly:

- `$ ./Hookbook profiles.txt captains.txt -n`
- `$ ./Hookbook profiles.txt -t captains.txt`
- `$ ./Hookbook -v profiles.txt captains.txt`
- `$ ./Hookbook profiles.txt captains.txt`

Here are some *invalid* commands, which your program must handle by printing a descriptive error message to `stderr` and exiting with status code `1` (the explanations shown are good "descriptive error messages"):

- `$ ./Hookbook` (not enough arguments)
- `$ ./Hookbook profiles.txt` (not enough arguments)
- `$ ./Hookbook -n -t profiles.txt captains.txt` (too many arguments)
- `$ ./Hookbook profiles.txt captains.txt -x` (invalid sort flag: `-x`)
- `$ ./Hookbook profiles.txt captains.txt foo.txt` (too many filenames)
- `$ ./Hookbook not-a-file.txt not-a-file-again.txt -n` (file can't be opened)
- `$ ./Hookbook unreadable-file.txt other-file.txt -n` (file can't be opened)

If you can open it for reading, you may assume the following about the file referred to by the first non-sort-flag command-line argument:

- Its name does not begin with a hyphen (`'-'`)
- It is a text file. That is, we will not try to open a binary file, *e.g.*, an image
- It conforms to the input specification above for the **profiles file**
- No line in the file is longer than 127 characters
- No line in the file contains any non-ASCII characters

If you can open it for reading, you may assume the following about the file referred to by the second non-sort-flag command-line argument:

- Its name does not begin with a hyphen (`'-'`)
- It is a text file
- It conforms to the input specification above for the **captains file**
- The contents of every non-empty line is the name of a pirate with a profile in the profiles file

  Think: Could the two filenames in a valid command refer to the same file under these assumptions? Why or why not?

  Think: Could a line in the captains file exceed 127 characters?

  Note: We will not test your submission with files that you *can* open that do not conform to the specified format, but we may test your program with files that you *cannot* open. Any file that cannot be opened should be treated as a non-existent file.

Testing and Grading

Grading for this assignment is automated. Unless otherwise requested, we will grade the *latest* submission on Gradescope. The autograder will perform three kinds of tests on your code:

- Compilation tests
- Diff tests
- Unit tests
- Performance tests

We will first ensure your program compiles without errors or warnings using the `makefile` that you submit. We will compile several versions of your executable, each with a different grid size, running `make clean` in between each.

> *IMPORTANT:* **You will receive no credit for this assignment if your program does not compile on Gradescope with your provided makefile, including `make clean`.** If your program compiles on the Zoo but not Gradescope, it is your responsibility to seek out clarification as to why and to modify your submission so that it *does* compile on Gradescope.
>
> The most common cause of a discrepancy is submitting incorrect files.

Next, we will run the compiled `Hookbook` executables on several test cases and check its output using the `diff` tool available on Unix systems including the Zoo and Gradescope. For each diff test, you will receive

full credit if there are **no differences** between the expected output and your program's actual output. If there are any differences between the two for a given test case, you will receive no credit on that test case.

We will also run **unit tests** on your code. These unit tests will exercise the required functions that you have implemented, in isolation from one another, to assess that each individual function is correct relative to its specification.

Here is a possible unit test for `list_remove` (note: this test case *does not* test `pirate_destroy` or `list_destroy`, as the the assertions are made before those functions are called; nor does it test the functionality of `pirate_create`, `list_length`, or `list_insert`—rather, it *assumes* those functions are correctly implemented):

```
void test_remove_1()
{
    pirate_list* p = list_create();
    pirate* r = pirate_create("Foo");
    list_insert(p, r, 0);
    pirate* r2 = list_remove(p, r);
    if (list_length(p) == 0 && r2 != NULL && strcmp(r2->name, "Foo") == 0)
    {
        printf("test_remove_1 passed!\n");
    }
    else
    {
        printf("test_remove_1 failed!\n");
    }
    pirate_destroy(r2);
    list_destroy(p);
}
```

It could be called by a `main` function that does nothing but calls a bunch of similar unit test functions.

> *IMPORTANT:* **You will not receive any credit for this assignment if the unit testing suite does not compile.** It is possible that your provided `makefile` may successfully compile your program, but that the unit test program does not compile (or vice versa!). If this happens, it is your responsibility to seek out clarification as to why and to modify your submission so that the unit testing script *does* compile.
>
> The most common cause is changing the required function prototypes or other declarations in header files. The unit tests (by necessity) assume the prototypes for all required functions are provided unchanged from the starter code. **Do not modify the prototype for any of the required functions.**
>
> This can also happen because the unit test program is written in C++, which (despite popular belief) is not actually a superset of C. That is, although many C programs are also valid C++ programs, *not all of them are*! Due to this property of the autograder, we require that your C programs use only the subset of C that is valid C++.

Finally, your code will be run against a variety of benchmarks to assess the performance of your submission. These benchmarks will include a script that decides the asymptotic complexity of your algorithms. The performance tests will also check that your executable produces no memory leaks or errors when run with `valgrind` on a variety of inputs.

Public vs. Private Test Cases

As with the previous assignment, there will be a reasonably complete suite of test cases that will run automatically on your submission when you upload it to Gradescope. You will only see feedback about your submission's performance on a subset of these cases (the "public" test cases).

Depending on how mean we're feeling, we may or may not release additional public test cases before the end of the submission period. We plan to release some downloadable diff tests during the first week of the assignment, which you will be able to use to judge the correctness of your program without needing to upload it to Gradescope.

Begging us for more public test cases is a good way to make us meaner ;-).

The output of passed test cases on Gradescope may not be easy to decipher because of how we had to name our test cases in the unit testing framework we are using. Ask a staff member if you're really curious about what a particular test case does (Alan wrote the script so if the first person you ask doesn't know, ask him). For failed test cases, however, the output should be somewhat more enlightening because it will display a message to you describing the test case along with the assertion that was violated and caused the failure.

Weekly Survey

Each week, you will be asked to fill out a survey asking for at least the following information:

- Your full name and netID
- Your estimate of the time required to complete this assignment, selected *before* you start writing any code.
- The total time you *actually* spent on the assignment
- The names of all others (but not members of the teaching staff) with whom you discussed the assignment for more than 10 minutes
- A brief discussion (~100 words) of the major conceptual and coding difficulties that you encountered in developing and debugging the program (and there will always be some).Your discussion in one of the weekly surveys (one of the surveys released on Feb 9, 16, or 23) must also address the following:
  - Suppose we asked you to allow a user to enter a pirate's name and see information about them. What might be a better (*i.e.*, more efficient) data structure than an array list to help you accomplish this task? Why?
  - What might a better (*i.e.*, more efficient) data structure be for a `skills_list` than a singly-linked list? Why?

    Important: There are specific prompts you must address in your discussion for one of the weekly surveys during this assignment:
    - Suppose we asked you to allow a user to enter a pirate's name and see information about them. What might be a better data structure than an array list to help you accomplish this task? Why?
    - What might a better data structure be for a `skills_list` than a singly-linked list? Why?

In total, the weekly surveys are worth 5% of your overall grade for the course.

Submission

Submit your project to Gradescope by uploading all files needed to run your program. It is due **11:59 PM NHT on Friday, February 23, 2024**.

You must submit the following files:

- `makefile` (or `Makefile`)
- `hookbookb.c`
- `pirate.h`
- `pirate.c`
- `pirate_list.h`
- `pirate_list.c`
- `libhookbook.h`
- `libhookbook.c`
- `skills_list.h`

- `skills_list.c`

Refer to the Syllabus for information on collaboration and late-assignment policy.

Optional Additional Activities

Completing these additional activities is at your discretion and for your learning benefit, but they do not count toward your grade on this assignment or in the course.

1. Check that the input files are actually valid
   - Check that the first command-line argument that's a file name conforms to the profiles file input format
   - Check that the second one conforms to the captains file input format
2. Implement `skills_list` such that one of the following is true:
   - Adding a skill to the list of skills takes $O(1)$ time and printing the list takes $O(n \log n)$ time (and sorts the list just before printing), or
   - Adding a skill to the list of skills takes $O(\log n)$ time (and maintains that the skills are always sorted) and printing the list of skills takes $O(n)$ time
     Note: A linked list cannot conform to these complexity bounds, so if you complete this second additional activity, make a clear note of it in your submission, so that your grader knows why `skills_list` is not implemented as a linked list.
3. Create a suite of unit tests for your `skills_list` interface, in a file named `unittest.c`, and add a target to your makefile to compile it into an executable that will automatically test your implementation of `skills_list`
   - If you're feeling extra adventurous, write a C++ program called `unittest.cpp` that uses the GoogleTest unit testing framework, which is something of an industry standard for unit testing C and C++ code (it is also the framework used for the unit tests on Gradescope!)