Week 4 Recitation Exercise

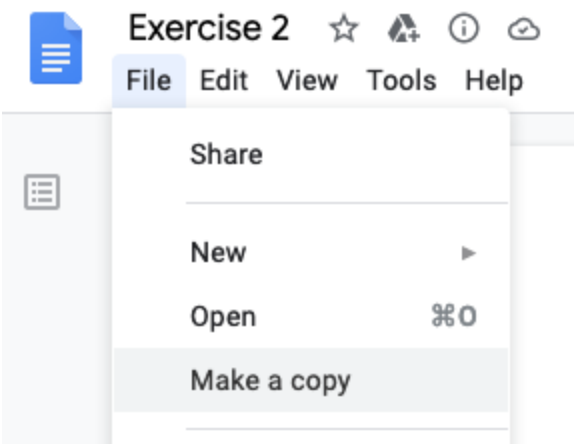
CS 261 – 10 points

In this exercise, you will explore the concepts of ***encapsulation*** and ***iteration***. Encapsulation is a principle by which access to the inner implementation details of a data structure are hidden or restricted, thereby preventing users of that data structure from manipulating (or even worrying about) those inner details. This makes data structures generally easier for users to work with and think about. However, it complicates some tasks, such as iterating through the collection of elements stored in a data structure. You'll examine this issue more closely here.

Follow the instructions below, and when you're finished, make sure to submit your completed exercise on Canvas. Please submit one copy per group (as long as all group members' names are listed below, it's fine if just one person submits). Remember that this exercise will be graded based on effort, not correctness, so don't worry if you don't get all the right answers. Do make sure you *try* to get the right answers though.

Step #1: Organize your group and create your submission document

Your TA will assign you to a small group of students. Among your group, select one group member to record your group's answers for this exercise. This group member should make a personal copy of this document so they are able to edit it:



Step #2: Add your names

Next, your group's recorder should add the names and OSU email addresses of all of your group's members in the table below:

Group member name	OSU email address

Step #3: Grab some code for a simple linked list implementation

Now, take a look at the code in the following GitHub repo:

https://github.com/osu-cs261-f21/recitation-4

Clone this repo onto your own machine, but note that you won't be able to push changes back to the repo, since your access to it is read-only. If you'd like to be able to push any changes you make to the code back to GitHub, you'll have to make a fork first and then clone and work with the fork.

The code in the repo above contains a very basic linked list implementation, similar to the one you worked with in assignment 1, which contains these files:

- **`list.h`** – This file contains the user-facing declarations for the linked list implementation. The user of the linked list will `#include` this file to use the linked list.
- **`list.c`** – This file contains definitions for the structures and functions associated with the linked list.
- **`main.c`** – This file contains a simple program that tries to iterate through a linked list to print its values.

Step #4: Try to compile and run the code

Try to compile and run the linked list code you just cloned to your machine. You can use the included makefile to compile:

```
$ make
```

What happens when you try to compile the code? You should see a few compiler errors that say something like “incomplete type” or “incomplete definition of type” in `main.c`. This happens because the linked list implementation in `list.h` and `list.c` is written to be “encapsulated”, i.e. to hide the internal implementation details of the list from the user of the list (here, the “user” of the list is `main.c`).

In particular, the user’s view of the linked list is specified in `list.h`. Here, only a ***forward declaration*** of the list structure is included:

```
struct list;
```

Importantly, the *fields* of the list structure are not defined in `list.h`, nor is the link structure (`struct link`). These things are defined *only* in `list.c`, and because the code in `list.c` is not “visible” to the user (specifically, its scope is limited to the file `list.c`), it is an error for the user to try to refer to or use any of these details. In particular, it is an error for the user to try to use the link structure or to try to access fields of the list structure, as they do in `main.c` (e.g. by creating and trying to use a variable of type `struct link*` or by trying to access the `head` field of a list).

This is all *purposeful*. In particular by restricting the user’s access to these internal implementation details, they are forced to interact with a linked list using only whatever functions are defined in `list.h`. This means two things:

- The user cannot manipulate (and possibly break) the internal state of a linked list. In particular, by restricting a user to interact with a linked list via the functions defined in `list.h`, we guarantee that their interactions will happen in a way that *we* control (since we are the ones who implemented those functions).
- The user doesn’t have to think about the internal details of the linked list, which in turn lightens the cognitive burden required to use the list.

In fact, this is very similar to making all of a class’s members private in C++ and providing only public member functions the user must use to interact with the class.

Step #5: Figure out how to let the user iterate through a linked list

Unfortunately, as we've seen, restricting access to the implementation details of the linked list, as we've done in this code, makes it impossible for the user to do something as simple as iterating through the list to print the values stored there. The following question arises:

> *How can we allow the user to iterate through a list to access the values stored there without exposing internal implementation details to the user?*

Discuss this question with your group, and, as a group, try to come up with a solution to the question. In particular, try to devise a way for the user to iterate through a list and access the values stored there that satisfies this criterion:

> *Your solution should not expose any fields of the list structure (`struct list`), and it should not expose the link structure (`struct link`) to the user at all. In other words, `list.h` (and `main.c`) should not mention the link structure and should still contain only the forward declaration of the list structure:*
>
> ```
> struct list;
> ```

In order to satisfy this criterion, try to imagine a solution that simply adds new functions to the linked list interface. Your solution might also introduce one or more new structures for the user to use. However, if you do introduce any new structures, make sure to hide their internal implementation details (i.e. by including only a forward declaration of the structure in `list.h` and fully defining the structure only in `list.c`).

Hint: Try to imagine a solution that fits into a typical loop structure, e.g.:

```
struct list* list = list_setup();

while (/* there are more elements in list */) {
  int val = /* the next value from list */;
  /* Do something with val, e.g. print it. */
}
```

Once your group has a solution to the question above, use the space below to describe your solution. You may include code below or simply provide a detailed explanation in words of how your solution works, or you may write a mix of code and words. Make sure to describe how to fix `main.c` using your solution.

Solution:

[Write your solution here.]

Submission and wrap-up

Once you've completed this exercise, download your completed version of this worksheet as a PDF and submit that PDF on Canvas (refer to the image below to help find how to download a PDF from Google Docs). Again, please submit one copy of your completed worksheet for your entire group. It's OK if just one member submits on Canvas. All group members will receive the same grade as long as their names and email addresses are included above.