Encapsulation and Iterators

Encapsulation

- ***Encapsulation*** is one of the most important things to strive for when we're developing code that will be used by other developers e.g. implementations of general-purpose data structures.

- Encapsulation is specifically a mechanism through which we hide the internal details of a data type from the user of that data type, instead exposing only a simplified interface through which the user interacts with the data type.
    - To be clear, the term "user" here refers to another developer who will be using the code we've written.

- We've seen examples of encapsulation in action in some of the assignment code we've worked on already in this course.

- For example, the linked list implementation we've used has hidden the details of the list implementation behind a simplified interface. Specifically, only the name of the linked list data type was exposed to the user in the linked list header file `list.h`, but the actual details of that data type remained hidden:

  ```
  struct list;
  ```

- Because of the way this data type was encapsulated, if the user actually tried to access any of the internal fields of the list struct (e.g. directly accessing `list->head`), they would receive compiler errors saying something about "dereferencing a pointer of incomplete type."
    - Many of you, in fact, experienced exactly this error!

- In this case, encapsulation was enforced using a mechanism specific to the C language. Other languages have various other mechanisms for enforcing encapsulation. For example, in C++, we can enforce encapsulation by applying the `private` access specifier to specific class members.

    - Other languages, like Python and JavaScript, don't really have mechanisms for strictly enforcing encapsulation.

- There are many reasons why encapsulation is a good thing to strive for in your code. For example, when a user does not (and cannot) know the internal structure of a data type and must instead interact with that data type through a simplified interface, it reduces the cognitive overhead required for the user to understand and reason about that data type.
    - In other words, encapsulated data types are easier to use because they don't bog users down with lots of details.

- In addition, when a user does not (and cannot) know the internal structure of a data type, they can't (intentionally or unintentionally) misuse (and possibly break) the data type by manipulating its internals.
    - E.g. the user of our linked list implementation can't directly set `list->head` to `NULL`, which could cause a memory leak.

- This makes it easier for us to implement the data type. Specifically, by constraining the user to interact with the data type via its interface instead of via its internals, we can more easily control how the data type is manipulated, what possible values can be assigned to its internal fields, etc. In turn, this means, for example, that we don't have to do as much tedious error checking on our data type's internals, etc.

- However, encapsulation can also introduce its own challenges. One such problem arises when the user of a well-encapsulated collection wants to ***iterate*** through the elements in that collection, e.g. to visit each element in the collection within a loop.

- Take our linked list implementation, for example. How can a user of this linked list iterate through it when they can't access the internals of the list (e.g. the head of the list or its links)?

- The solution to this problem lies in an abstraction known as an ***iterator***.

Iterators

- An iterator is a data type that acts as a companion to a collection and provides a mechanism to iterate through that collection.

- An iterator is implemented in such a way as to have access to the internals of the collection, which is how it's able to iterate through the collection. Thus, each specific kind of collection will have its own iterator data type.
    - For example, a linked list implementation will have its own iterator, as will a stack, a queue, a deque, etc.

- All iterators typically provide two common functions in the interface they offer:
    - `next()` – moves the iterator to the next element in the collection and returns the value of that element.
    - `has_next()` – returns true or false to indicate whether or not there is another element in the collection to which to iterate.

- Many iterators also offer additional functions to do things like remove a value from a collection, update the value of an element, insert a new element into the collection, etc.

- The most basic and common way to use an iterator looks like this (assuming that we already have an iterator `iter` over a specific collection):

```
while (has_next(iter)) {
  value = next(iter);
  ... /* Do something with value. */
}
```

Linked list iterator

- We can start to understand a little better how iterators work by exploring how we might implement one for a linked list.

- Remember, iterators must be implemented in such a way as to have access to the internals of the data type over which they're designed to iterate.

    - For example, in C, the iterator could be defined within the same file as the collection data type.
    - In C++, the iterator could be implemented as a nested class or as a friend of the collection class.

- Thus, our linked list iterator will need to have access to the internals of the linked list. Assume the list for which we want to implement an iterator is implemented with the following structures:

```
struct link {
  void* value;
  struct link* next;
};

struct list {
  struct link* head;
};
```

- Our first task will be to define a structure to represent the list iterator. To do this, we'll have to understand how exactly our iterator will move through the list.

- The typical way to iterate through a linked list is to maintain a pointer (e.g. `curr`) to represent the current link being visited. This pointer initially points to the head of the list and moves from one link to the next (e.g. `curr = curr->next`).

- For our linked list iterator, it will be sufficient to simply maintain a pointer like this:

```
struct list_iterator {
  struct link* curr;
};
```

- Next, we'll need some mechanism to create a new iterator and to associate it with a specific linked list over which to iterate. We can write a simple initialization function to allocate a new iterator and initialize it to point to the list's head:

```
struct list_iterator*
list_iterator_create(struct list* list) {
  struct list_iterator* iter =
    malloc(sizeof(struct list_iterator));
  iter->curr = list->head;
  return iter;
}
```

- Our iterator's `has_next()` function simply needs to check whether the iterator is currently pointing to `NULL`, which will be the case when it has reached the end of the list and there are no more elements to iterate to:

```
int has_next(struct list_iterator* iter) {
  return iter->curr != NULL;
}
```

- Finally, our iterator's `next()` function can simply return the value of the current link and update the iterator to point to the next link:

```
void* next(struct list_iterator* iter) {
  void* value = iter->curr->value;
  iter->curr = iter->curr->next;
  return value;
}
```

- Our iterator's implementation here could probably use a little more polish (e.g. error checking), but otherwise, it's complete, and it can be used as described above (e.g. in a while loop).