Lecture Summary

In this lecture, we learned about iterators. The idea of an iterator is to provide a functionality to go through all elements of a data structure (such as a List, Linked List, or something else), without exposing how the internals are stored. We also saw how to implement an iterator in C++, and discussed some of the bigger picture associated with it.

1 Iterators

When we store a set of items in a data structure, one of the things that we often want to do is be able to output all of them. For instance, in the case of our social network, we want to be able to output all wall posts of a user, or all users in the system. How should we do that?

1.1 Some attempts

One solution would be to give the programmer (who wants to output things) access to the internals of our data structure. For instance, if it is known that we used a linked list, and we return the head of the linked list, then starting from the head, it is possible to go through all items of the linked list. But of course, this is a terrible solution: it violates everything we are trying to do by hiding the implementation; the user of our data structure should only interact with specified functions, and not know what goes on internally.

At the other extreme would be to simply include in our implementation of the data structure another function `output()`, which prints all items to `cout`. But what if we want to print them to a file instead? Well, we could print them into a string, which could then be written to either `cout` or a file. But now what if someone would like them output in a different format? Or someone only wants to output all the entries that satisfy certain criteria? We might end up writing a huge number of different output functions. And then additional functions that might actually process the items. Clearly, that solution does not work, either.

A solution suggested in class was to essentially provide the interface of the `List` class: allow access to all elements by index, via a function `get(int i)`. This works better, but it is not very efficient: if the underlying implementation is via a linked list, then this takes time $\Theta(i)$ for element number $i$, for a total of $\Theta(\sum_{i=1}^{n} i) = \Theta(n^2)$. Whereas if we had just traversed the linked list, it would have been just $\Theta(n)$. It’s also overkill: we don’t need to be able to access elements by index; we just want to be able to go through all of them.

The solution that we arrived at is to have some kind of a “marker” that gets pushed through the data, and remembers where it was before. If the elements of the data structure were in some particular meaningful order (such as for a `List` data structure), then our marker would traverse them in that order. If the order of the elements was not meaningful (for instance, for a `Bag` data structure), then they could be output in any order.

The “marker” may be implemented differently, depending on how the data structure itself is implemented. If the data are stored in an array, then the marker may just be an integer, the index in the array that we’re currently at. If the data are stored in a linked list, then the marker may be a pointer to the current element. Ideally, the implementation of the marker should hide these kinds of detail, and just expose an interface whereby another function can retrieve the actual stored elements one by one, without ever having to know about the marker.
The next question we asked ourselves was where the marker should be stored. The first “obvious” choice was that it may be stored inside the data structure itself. So things may look something like the following for our new modified Bag class:

```cpp
class Bag<T> {
    ... // prior Bag functionality.
    T start(); // starts the traversal, returns first element
    T next(); // advances the marker internally, returns next element
}
```

Bag<T> bag; // contains some element already, somehow

for(T elt = bag.start(); end not reached; elt = bag.next())
    { // process elt, e.g., by printing it;
        }

We’d have to add a way to test if the end is reached, which is not very difficult. The problem with the implementation so far is that we might want to have multiple markers in use at once. Then, we couldn’t just use start() and next(), but would have to somehow index them. So the data structure would have to internally store a set of markers (e.g., as a bag or something). That’s workable, but the more “standard” solution is to define a separate class, whose objects have the ability of traversing the elements of the data structure.

### 1.2 Iterators

Such a class is traditionally called an *iterator*, so we will use that term instead of “marker” from now on. Since an iterator is a class/object encompassing one marker, it must provide functionality to advance, to dereference (i.e., read the data), and compare for equality or inequality, in order to terminate a loop. For readability (by analogy with traditional for loops), these functions are usually implemented with operator overloading.

The class which is being iterated over must have the following functionality:

- Return an iterator initialized to the beginning. This method is usually called begin(), and returns an iterator.
- Return an iterator initialized to the end, so that we can test when we are done. This method is usually called end(), and returns an iterator.

Suppose that for our Bag class, implemented, say, as a LinkedListBag (see earlier lectures), we may call the corresponding iterator a BagIterator. Then, in the main() function, our use of an iterator may look as follows:

```cpp
int main()
{
    IntLinkedListBag LLB;
    for (int i=2; i<10; i++)
        {
            LLB.add(i);
        }

    LLB.remove(7);

    for (BagIterator bi = LLB.begin(); bi != LLB.end(); ++bi)
        {
            // process bi, e.g., by printing it;
        }
}
```
cout << *bi << " ";
}
return 0;
}

Here, as you can see, we use comparison to the end() iterator to test whether we have passed the last element. To advance the iterator bi, we use operator overloading, so that it looks as much as possible to a typical loop in which we would increment an integer. Similarly, we use *bi to dereference the current value of the iterator, just like we do with pointers.

One important thing to keep in mind about iterators is that in general, we want to avoid modifying the underlying data structure while iterating over it. Most data structures do not guarantee what will happen if you try that, and it’s always a bad idea to execute operations whose results are unpredictable. For instance, some implementations of an iterator may have copied all of the data from the data structure over (so they won’t be affected by any change), while others may look at the data structure in real time (and will be affected).

1.3 Implementing an Iterator

Let us first see how we would implement a BagIterator class. First, we want to think about what data need to be stored internally. We need to have the marker; and since we are implementing the Bag using a linked list, that marker will be a pointer to an IntListElement. But we also need another field: we need to remember which data structure the marker is traversing. For instance, two markers both of which point to element 7 may not be the same if they are pointing to element 7 of different arrays. So we also need to store a pointer to the data structure being traversed. The signature thus has two private fields:

class BagIterator {
    public: // functions listed below
    private:
        const IntLinkedListBag *whatIAmTraversing;
        IntListElement *current;
}

The implementation of the methods now looks as follows:

BagIterator::BagIterator(const IntLinkedListBag *bag, IntListElement *p)
{
    whatIAmTraversing = bag; current = p;
}

int BagIterator::operator* () const
{ return current->data; }

bool BagIterator::operator== (const BagIterator &bagIt) const
{ return (current == bagIt.current && whatIAmTraversing == bagIt.whatIAmTraversing); }

bool BagIterator::operator!= (const BagIterator &bagIt) const
{ return (!this->operator== (bagIt)); }

BagIterator BagIterator::operator++ () const
{ current = current->next;
    return *this;
}
A few notes are in order about the implementation:

- For the $==$ operator, we compare the pointers (current vs. bagIt.current), rather than their data (current->data vs. bagIt.current->data). This makes sense because we might have a list that contains the same number multiple times; just because it does does not make a marker on the 5th element equal to a marker on the 8th. Also notice that we are comparing to make sure that both iterators are traversing the same data structure.

- For the $!=$ operator, we took the easy way out. Instead, we could have implemented it from scratch (using DeMorgan’s Theorem).

- For the increment operator, we not only want to move the marker, but also have to declare it of type BagIterator (rather than, say void). The reason is that people may want to write something like cout << *(++bi), where the iterator is moved and immediately dereferenced. While some people (including me) think that this is not good coding style, you should still support it.

  Also notice that we only implemented a pre-increment and no post-increment, so we can write ++bi but not bi++.

Now that we have implemented the BagIterator class, we can augment our IntLinkedListBag with two functions to return a start and end iterator:

```cpp
BagIterator IntLinkedListBag::begin() {
    return BagIterator(this, head); }

BagIterator IntLinkedListBag::end() {
    return BagIterator(this, NULL); }
```

Again, a few notes on what we did here:

- Remember that the C++ keyword this is determined at runtime and always contains a pointer to the object from which it is called. In other words, an object can use the word this to find out where in memory it resides.

  In our case, this is used to tell the iterator the identity of the object it is traversing. The other argument is just the head, i.e., where the traversal should start.

- For the end() function, we give the NULL pointer as the current element of the iterator. The reason is that the end() iterator should encode the case when the for loop has reached the end of the linked list, which happens when the pointer is a NULL pointer. If instead, we wrote return BagIterator(this, tail);, then by looking at what happens in our main() function above, you’ll notice that the last element of the bag wouldn’t be processed.

Again, notice how iterators let us go through all elements of a container without knowing any internal details about it. Only the LinkedListBag and the BagIterator need to know how Bag is actually implemented.

The implementation of an iterator for a Bag based on a linked list was pretty straightforward. If we think about how we would implement an iterator for a Bag built using an internal array, we would quickly discover that the iterator will store internally a pointer to the bag, as well as an integer to be used as an array index. Let’s say that internally, our IntArrayBag stores its data in a private array int *a. Now, the dereferencing would have to be of the form

```cpp
int BagIterator::operator* () const {
    return whatIAmTraversing->a[current]; }
```
This means that at the moment we want to return an item, we need access to a private variable in the data structure, which the iterator does not have. To solve this problem, C++ offers us the keyword friend. By declaring BagIterator a friend of IntArrayBag, the class IntArrayBag gives BagIterator access to all its private (and protected) fields and functions: in other words, if one class is a friend of another, it overrides traditional access rules. We would accomplish this by adding the following to the definition of IntArrayBag:

```cpp
class IntArrayBag {
    ... // old stuff
    public:
        friend class BagIterator;
}
```

While it is not ideal to expose private fields and functions to another class, exposing them to just one or a few friends is a decent compromise, given the functionality that we get out of it while still hiding the information from most other classes.

## 2 The Bigger Picture

### 2.1 Design Patterns

So far in this class, we had seen pretty standard ways for objects of different types to interact: inheritance, one class containing fields of another class’s type, and that’s about it. This is our first example of classes interacting in more complex ways: we have one class whose “job” it is to traverse another class and provide some functionality (access to elements) that is not naturally handled within the original class. In other words, we have abstracted one particular functionality out of one class into another.

This kind of paradigm, one class “visiting” all elements of another, is sometimes called a Visitor Pattern. It’s an example of what is called Design Patterns: interesting (and frequently recurring) ways in which different classes can interact with each other. Once we absorb the ideas of object-oriented programming more fully, it becomes really fun to think about objects as each having their own identity, filling a role in a larger system of components, and interacting with each other in particular ways. Design patterns are then an important way of identifying frequently occurring ways in which objects interact, and they help in designing well-structured larger software systems. As such, they are an important part of the field of Software Engineering. For the student interested in learning more, we recommend the classic book on “Design Patterns” by Gamma, Helm, Johnson, and Vlissides.

### 2.2 Foreach, Map, Functional Programming, and Parallelization

We have seen that for essentially all data structures, we will now want to use iterators and write loops of the following form:

```cpp
for (BagIterator bi = LLB.begin(); bi != LLB.end(); ++bi)
{
    // process *bi
}
```

Since the code will always have the exact same form, except for different ways of processing the elements, we could try to shortcut it even more. By implementing the processing of *bi into a function f, we could implement a function:

```cpp
void for_each (Iterator start, Iterator end, Function f)
{
    for (Iterator current = start; current != end; ++current)
        f(*current);
}
```
f(*current);
}

In fact, C++ implements an almost identical function that has only a few small differences. To use C++’s function, your iterator has to inherit from the abstract class iterator (for which you #include<iterator>). You’ll have to implement your function, and pass it in.

While using the for_each function would only save us a little bit of typing, it has a conceptual advantage: it clarifies what is going on in your code. For instance, by using this function, you are making it clear that you are simply applying the same function to each element, and there is no state transferred from one iteration to the next.

This kind of construction, where you pass a function as a parameter to another function and then apply it inside, is one of the key constructs of functional programming. The particular case in which one function is applied to each element of a collection is called the map paradigm. In other words, for_each implements the map paradigm in C++.

Thinking a little further, the map paradigm is half of the map-reduce paradigm of parallel processing. This is one of the really popular techniques for massively parallelizing Big Data computations over large numbers (hundreds of thousands) of machines, at the heart of what Google and other companies do. The idea is to farm out a lot of parallel computation on individual elements to machines (via map), and then combine the results into the final output (called reduce). If you use constructs such as for_each, then you make it much easier for your code to be transferred into map-reduce code, and thus parallelized later. Notice that this is where it comes in handy that the different iterations of the loop do not interact with each other, and we instead have each element processed by itself; by doing this, we ensure that the different parallel machines do not need to communicate with each other. (Network usage is slow.)

In addition to the for_each loop, if you use iterators, you also have some other standard functionality available: you can add up all elements, find the maximum, and a few others, simply by passing iterators into some pre-programmed functions. Check the textbook for some examples.