And the Iterators Gonna Iterate-ate-ate

Step 1

As you may have noticed by now, although many data structures have similar functionality (e.g. insert, remove, find, etc.) and similar purpose (i.e., to store data in a structured fashion), they are implemented in fundamentally different ways. The differences in their implementations are essentially the causes of their differing costs/benefits. However, as data structure wizards, we want to code our data structures in a fashion such that the user doesn't have to worry about the implementation details. For example, take the following segment of code:

```
for(auto element : data) {
    cout << element << endl;
}
```

What data structure is `data`? Is it a `set`? `unordered_set`? `vector`? It could be any of these, or maybe something else altogether! Nonetheless, we are able to iterate through the elements of `data` in a clean fashion. In this section, we will be covering the C++ Standard Template Library (STL), with our attention focused on iterators, the mystical entities that make this functionality possible.

The **iterator pattern** of Object-Oriented design permits access to the data in some data structure in sequential order, without exposing the container's underlying representation. Basically, it **allows us to iterate** (hence the name "iterator") **through the elements of a data structure in a uniform and simple manner**.

You may be familiar with the **for-each** loop (shown above), where we iterate through the elements of a data structure in a clean fashion: the programming language actually uses an iterator to perform the for-each loop. Note that iterators exist in numerous languages, not just C++.

Step 2

You may recall that Java has many standard data structures (e.g. lists, sets, maps, etc.) already implemented and ready to use in the Collections API in the `java.util` package. The Java Collections API defines an `Iterator` interface that can be used to traverse collections (each collection is responsible for defining its own iterator class), which only has three methods: `next`, `hasNext`, and `remove`.

C++ has a package with similar functionality to Java's Collections API: the C++ **Standard Template Library (STL)**. With regard to iterators, which are what we are focusing on in this section, the specifics of C++ iterators are slightly different in comparison to Java iterators. The syntax is messy to describe verbally, so the best way to teach how to use iterators is to provide you with a step-by-step example:

```
vector<string> c;
/* populate the set with data */

vector<string>::iterator itr = c.begin(); // get an iterator pointing to c's first element
vector<string>::iterator end = c.end();   // get an iterator pointing past c's last element
while(itr != end) {                       // loop over the elements (overloaded !=)
    cout << *itr << endl;                 // dereference the iterator to get element
    ++itr;                                // point to next element (overloaded ++ pre-increment)
}
```

- `*itr` returns a reference to the container object that `itr` is pointing to
- `itr1 == itr2` returns `true` if `itr1` and `itr2` refer to the same position in the same container (`false` otherwise)
- `itr1 != itr2` is equivalent to `!(itr1 == itr2)`

- `c.begin()` returns an iterator positioned at the first item in container `c`
- `c.end()` returns an iterator positioned *after* the last item in container `c` (typically `NULL` or some equivalent)

Step 3

EXERCISE BREAK: Which of the following statements about C++ iterators are true? (Select all that apply)

To solve this problem please visit https://stepik.org/lesson/26105/step/3

Step 4

As mentioned before, if our collection has an iterator class implemented (which all of the C++ STL data structures do), we can easily iterate over the elements of our collection using a **for-each loop**. The syntax is as follows:

```
void printElements(vector<string> & c) {
    for(string s : c) {    // "for-each" string 's' in 'c':
        cout << s << endl; // print 's'
    }
}
```

You may have noticed that the above code functions identically to the example code we showed you when we first introduced C++ iterator syntax. In addition to looking similar and resulting in identical output, the two methods are actually functionally identical. As we mentioned, for-each loops use iterators behind the scenes. When we call the for-each loop, C++ actually creates an iterator object pointing to the first element, keeps incrementing this iterator (performing the contents of our loop's body on each element), and stops the moment it reaches the end. It simply hides these details from us to keep our code looking clean.

Step 5

EXERCISE BREAK: What will be outputted by the code below?

```
vector<char> data;
data.push_back('H');
data.push_back('e');
data.push_back('l');
data.push_back('l');
data.push_back('o');
data.push_back('!');

for(auto element : data) {
    cout << element;
}
```

To solve this problem please visit https://stepik.org/lesson/26105/step/5

Step 6

CODE CHALLENGE: Using a C++ Iterator

In this challenge, you will be given a `set` consisting of randomly-generated `string` objects. Your task will be to use an `iterator` to output the elements of the `set` in ascending alphabetical order. Print out the elements to `cout`, one element per line.

Hint: Feel free to revisit step 2 to remind yourself of the syntax of the iterator class.

Hint: Notice that you are given a C++ `set` object. When an iterator iterates over the elements of a C++ `set`, in what order are the elements visited?

Sample Input:

```
CCC
TTT
AAA
GGG
```

Sample Output:

```
AAA
CCC
GGG
TTT
```

To solve this problem please visit https://stepik.org/lesson/26105/step/6

Step 7

Of course, being the hard-working and thorough computer scientists that we are, if our goal in this course is to implement data structures, we will want to implement the iterator pattern for our data structures so that users can easily iterate through the container's elements! If we want to implement the iterator pattern, we will we need to implement the following functions/operators:

Functions in the Data Structure Class:

- **`begin()`** - returns an `iterator` object "pointing to" the first element of the container
- **`end()`** - returns an `iterator` object "pointing just past" the last element of the container

Operators in the Data Structure's Iterator Class:

- **`==`** (equal) - returns true (1) if the two iterators are pointing to the same element, otherwise false (0)
- **`!=`** (not equal) - returns false (0) if the two iterators are pointing to the same element, otherwise true (1)
- **`*`** (dereference) - returns a reference to the data item contained in the element the iterator is pointing to
- **`++`** (pre-increment) - causes the iterator to point to the next element of the container and returns the object pointed to AFTER the increment
- **`++`** (post-increment) - causes the iterator to point to the next element of the container, but returns the object pointed to BEFORE the increment

In the next step, we will provide an example of implementing an iterator.

Step 8

Below is an example of implementing an iterator for a **Linked List**. Note that we have omitted the actual **Linked List** functions (e.g. `find`, `insert`, `remove`, etc.) for the sake of keeping the example clean and focused on iterators.

```
// Helper Node class
class Node {
    public:
        int value;
```

```
        int value;
        Node* next;
};

// Linked List class
class LinkedList {
    public:
        Node* root; // root node

        // Iterator class
        class iterator : public std::iterator<std::forward_iterator_tag, int> {
            public:
                friend class LinkedList; // declare Linked List class as a friend class
                Node* curr;              // the Node this iterator is pointing to

                // the following typedefs are needed for the iterator to play nicely with C++ STL
                typedef int value_type;
                typedef int& reference;
                typedef int* pointer;
                typedef int difference_type;
                typedef std::forward_iterator_tag iterator_category;

                // iterator constructor
                iterator(Node* x=0):curr(x){}

                // overload the == operator of the iterator class
                bool operator==(const iterator& x) const {
                    return curr == x.curr; // compare curr pointers for equality
                }

                // overload the != operator of the iterator class
                bool operator!=(const iterator& x) const {
                    return curr != x.curr; // compare curr pointers for inequality
                }

                // overload the * operator of the iterator class
                reference operator*() const {
                    return curr->value;    // return curr's value
                }

                // overload the ++ (pre-increment) operator of the iterator class
                iterator& operator++() {
                    curr = curr->next;     // move to next element
                    return *this;          // return after the move
                }

                // overload the ++ (post-increment) operator of the iterator class
                iterator operator++(int) {
                    iterator tmp(curr);    // create a temporary iterator to current element
                    curr = curr->next;     // move to next element
                    return tmp;            // return iterator to previous element
                }
        };

        // return iterator to first element
        iterator begin() {
            return iterator(root);
        }

        // return iterator to JUST AFTER the last element
        iterator end() {
            return iterator(NULL);
        }
};
```

Thus, if we were to want to iterate over the elements of this **Linked List**, we could easily do the following:

```
for(auto it = l.begin(); it != l.end(); it++) {
    cout << *it << endl;
}
```

In this example, because we were creating an iterator for a **Linked List**, which is a structure built around pointers to nodes when implemented in C++, the iterator objects had `Node` pointers (i.e., `Node*`) to keep track of the element to which they pointed. In the next example, we will explore how to implement an iterator for an **Array List**.

Step 9

Below is an example of implementing an iterator for an **Array List**. Note that we have omitted the actual **Array List** functions (e.g. `find`, `insert`, `remove`, etc.) for the sake of keeping the example clean and focused on iterators.

```
// Array List class
class ArrayList {
    public:
        int arr[10]; // backing array
        int size;    // number of elements that have been inserted

        // Iterator class
        class iterator : public std::iterator<std::forward_iterator_tag, int> {
            public:
                friend class ArrayList;  // declare Array List class as a friend class
                int* curr;               // the element this iterator is pointing to

                // the following typedefs are needed for the iterator to play nicely with C++ STL
                typedef int value_type;
                typedef int& reference;
                typedef int* pointer;
                typedef int difference_type;
                typedef std::forward_iterator_tag iterator_category;

                // iterator constructor
                iterator(int* x=0):curr(x){}

                // overload the == operator of the iterator class
                bool operator==(const iterator& x) const {
                    return curr == x.curr; // compare curr pointers for equality
                }

                // overload the != operator of the iterator class
                bool operator!=(const iterator& x) const {
                    return curr != x.curr; // compare curr pointers for inequality
                }

                // overload the * operator of the iterator class
                reference operator*() const {
                    return *curr;          // return curr's value
                }

                // overload the ++ (pre-increment) operator of the iterator class
                iterator& operator++() {
                    curr++;                // move to next slot of array
                    return *this;          // return after the move
                }

                // overload the ++ (post-increment) operator of the iterator class
                iterator operator++(int) {
                    iterator tmp(curr);    // create a temporary iterator to current element
                    curr++;                // move to next slot of array
                    return tmp;            // return iterator to previous element
                }
        };

        // return iterator to first element
        iterator begin() {
            return iterator(&arr[0]);
        }

        // return iterator to JUST AFTER the last element
        iterator end() {
            return iterator(&arr[size]);
        }
};
```

Thus, if we were to want to iterate over the elements of this **Array List**, we could easily do the following:

```
for(auto it = l.begin(); it != l.end(); it++) {
    cout << *it << endl;
}
```

As can be seen, similar to a **Linked List**, we still use pointers to point to our elements when implementing an iterator for an **Array List**. Note that in the **Array List** iterator, iterators point directly to elements in the array (i.e., the actual integers we are storing). However, in the **Linked List** iterator, iterators pointed to nodes and we had to use the nodes to access the elements we were storing. This is a direct consequence of a **Linked List** property: memory is allocated as nodes are created dynamically. Consequently, nodes end up all over the place in memory, and thus we need to use the nodes' forward pointers to iterate over them. With an **Array List**, however, because the slots of the array are all allocated at once and are contiguous in memory, we can simply move across cells of the array.

Step 10

We started this chapter by learning *about* different data structures, without worrying too much about actual implementation. Then, we learned about some specifics regarding implementation of the data structures themselves. Now, we have learned about a key feature that we can implement into our data structures: iterators. Iterators are powerful entities that allow the user to iterate over the elements of *any* data structure in a clean and uniform syntax, without knowing how the data structure actually works on the inside.

Of course, as you noticed in the last portions of this section, iterators can be a *pain* to implement from scratch! Lucky for us, in practically all programming languages, some pretty smart people have already gone through and implemented all of the fundamental data structures *with* iterators! In practice, you will always want to use some existing implementation of a data structure (and its iterator implementation). In C++ specifically, you want to use data structures already implemented in the STL, which are implemented extremely efficiently.

Nevertheless, it is important to know *how* to go about implementing things from scratch, as you never know what you will see in life! Perhaps you will enter an extremely niche field of research, and data structures you need have not yet been implemented in the language of your preference, or perhaps you are secretly a genius who will one day design a groundbreaking super efficient data structure all on your own, for which you will of course give credit to the authors of this amazing online textbook that sparked your interest in data structures in the first place. Either way, you may one day find yourself needing to know the good practices of data structure implementation, and the iterator pattern is a key feature you will want to implement.