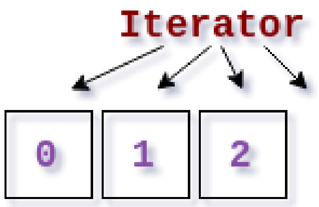
Iterators

Iterators traverse container classes (some Standard Template Library data structures). They can be thought of as a pointer as they behave similarly. Just like a pointer, an iterator points to a single element/node at a time and can be manipulated to point to other elements/nodes.



To declare an iterator, you must know which data structure you are going to use. For example, if you want an iterator for an **int vector** and another for a **char linked list**, you declare two separate iterators as such (note that **std::** is not required if you are **using namespace std;**):

```
std::vector<int>::iterator it; //it is iterator variable
std::list<char>::iterator it2; //it2 is iterator variable
```

Now, the iterators have to know where to point to. There are a couple of options you have. You can either set the iterator to the beginning of the data structure or at the end. Note that if you set the iterator to the end, it will point to **one past the last element**. This can be thought of as a sentinel value (like **nullptr**). Let's assume the vector variable **myVector** has been declared and initialized to some values and the list variable **myList** has also been declared and initialized to some values. The vector iterator will be set to the beginning and the list iterator will be set to the end:

```
it = myVector.begin(); //Points to first element
it2 = myList.end(); //Points to one past the last element
```

This can be done on the same line as the iterator declaration.

You are able to manipulate iterators by using the operators **++**, **--**, **==**, **!=**, *****. The **++** is for moving the iterator to the next element. The **--** is for moving the iterator to the previous element. The **==** tests for equality of two iterators. This will return true if both iterators are pointing to the same element. The **!=** tests to see if two iterators are not pointing to the same element. This will return true in that scenario. The ***** is the dereferencing operator. This is how you access the data where iterator is pointing. This is exactly the same as how a pointer accesses data.

If you would like to iterate through your whole container class, you are able to do so with iterators. Let's use **it** since it is already initialized to the beginning of the data structure:

```
for(it; it != myVector.end(); it++) //Loop terminates when iterator
    cout << *it << endl;           //reaches one past the end element
```

The example runs through the vector displaying each element with the dereferenced iterator.

There are three types of iterators to work with: **Forward**, **Bidirectional**, and **Random Access**. These are in the form of a hierarchy. The lowest is **Forward**, the next highest is **Bidirectional**, and the highest is **Random Access**. All higher-level iterators will have all the properties of the iterators that rank below it. For example, you will see **Bidirectional** iterators can do everything a **Forward** iterator can do plus more.

Forward iterators can only use the **++** operator. **Bidirectional** iterators can use both the **++** and **--** operators. **Random Access** can use both the **++** and **--** operators as well as the square bracket operator **[]** as seen in arrays. The specific type of iterator depends on the data structure you are using. For example, vector and deque use **Random Access** iterators, and list uses **Bidirectional** iterators. Keep in mind some data structures do not have iterators, such as queue and stack.

The iterator examples on the previous page were examples of mutable iterators. This means that once you dereference the iterator, you are able to manipulate data, just like a normal, dereferenced pointer. You are able to have a constant iterator that cannot alter the contents it points to. There are differences in the declaration as well as in the initialization:

```
std::vector<int>::iterator it; //Mutable iterator
std::vector<int>::const_iterator it; //Constant iterator
it = myVector.begin(); //Mutable iterator initialization
it = myVector.cbegin(); //Constant iterator initialization
```

In the declaration for a constant iterator, there is **const_** before **iterator**. In the initialization, the member function is **cbegin()** to denote a constant iterator. Remember that a constant iterator is able to move from element to element, but the iterator cannot alter what it points to.

If you have a container class with **bidirectional** iterator capabilities, you are able to work with the container in reverse. This makes the first element to work with the last element of the data structure. A special type of iterator is needed to make this happen (let's say we have a deque of type **char** called myDeque):

```
std::deque<char>::reverse_iterator it = myDeque.rbegin();
```

Here, we declared and initialized a reverse iterator. Notice there is **reverse_** before **iterator** in the declaration. In the initialization portion, the member function is **rbegin()**. In this case, **rbegin()** points to the last element and **rend()** points to **one past the beginning element**. Now you can use a **for** loop and traverse backward as such:

```
for(it; it != myDeque.rend(); it++)
```