# Iterators

## 1 Introduction

When you need to visit all of the elements of a vector `vec` from the first to the last, you could use a loop such as

```
for ( int i = 0; i < vec.size(); i++ )
    // do something with vec[i]
```

This works, and is a fine solution. It will also work with C++ strings. But what if you needed to visit all elements of a C++ `list` object? How could you do that? The short answer is that, without iterators, you cannot. Iterators make it possible. These notes answer the question, "What is an iterator, and how do you use it?"

***Iterators*** are a generalization of pointers in C++. They allow a program to navigate through different types of containers in a uniform manner. Just as pointers can be used to traverse a linked list or a binary tree, and subscripts can be used to traverse the elements of a vector, iterators can be used to sequence through the elements of any standard C++ container class. Iterators are specialized to "know" how to sequence through particular containers; an iterator that can sequence through a `vector` is a different type than one that can iterate through a `list`. An example follows.

Listing 1: A function to iterate through a list.

```
void addsuffixto_list(  strlist & names, char* str )
{
    // Declare an iterator that can process lists of strings
    strlist::iterator l_iter;

    // iterate through the list
    for ( l_iter = names.begin(); l_iter != names.end(); ++l_iter )
        l_iter->append(str);
}
```

This example demonstrates several concepts:

- An iterator is specific to a container. In Listing 1, `l_iter` is of type `list<string>::iterator` because we can only use an iterator defined within the `list` template class to sequence through a `list` class instance.
- An iterator is advanced through a container by applying the pre-increment operator to it. This is just like the semantics of pointers. In this example, `++l_iter` advances it through the list.
- An iterator is dereferenced by using the same dereference operator that is used with pointers. In this example, `l_iter->append()` is the member function called on the string pointed to by the iterator `l_iter`.

- All containers provide a set of iterator operations. In particular, the `begin()` method returns an iterator that references the first element in the container, viewed as a sequence, and the `end()` method returns a one-past-the-last-element iterator, which can be used to check if an iterator has reached the end of the container.

You do not need to know how iterators work to use them. That is what makes them convenient.

# 2 More Details

## 2.1 Iterator Types

There are different types of iterators, even for a single container class. There are ***constant iterators***, of C++ type `const_iterator`, which cannot be used to modify the objects to which they refer. There are also ***reverse iterators***, declared to be of type `reverse_iterator`, which are just what their name implies – iterators that "go in reverse." When a car is in reverse and you accelerate, it moves backwards. When a reverse iterator is incremented, it goes backwards too. Soon you will see some examples. There are also ***constant reverse iterators***, which are what their name implies – reverse iterators that cannot be used to modify the objects to which they refer. They are declared to be type `const_reverse_iterator`. Because there are reverse iterators, ordinary iterators are called ***forward iterators*** for clarity.

Containers such as vectors and lists provide the `begin()` and `end()` functions as described above, but these are not the only functions that return some type of iterator. The others include

`rbegin()` returns a `reverse_iterator`, i.e., one that starts at the last element and travels towards the first element as it is advanced.

`rend()` returns a `reverse_iterator` pointing to the element one before the first element in the container. This element does not exist, of course, but the iterator is used as a sentinel in the same way that the one returned by `end()` is used.

`cbegin()` like `begin()`, but returns a `const_iterator`.

`cend()` like `end()`, but returns a `const_iterator` that points to one-past-the-last-element.

`crbegin()` like `rbegin()`, but returns a `const_reverse_iterator` that starts at the last element.

`crend()` like `rend()`, but returns a `const_reverse_iterator` pointing to the element one before the first element in the container.

The following containers provide all of the above iterator-returning functions:

- `array`
- `list`
- `map`
- `set`
- `string`
- `vector`

The `string` class is not technically a container but is included because it does have these methods. There are other containers such as stacks and queues that do not provide these methods, because as abstractions, they are not supposed to provide method for traversing their contents.

## 2.2 Iterator Operations

Not all iterators support the same set of operations, but for the above set of classes, the following operations are supported by valid iterators:

Dereference and read `= *iter`

Dereference and modify `*iter =`

Comparison `iter1 == iter2`

Advance `++iter`

Dereference and access `iter->`

An iterator is ***valid*** if it points to an element. It might be invalid because it was never initialized, because the element to which it pointed was removed, or the container into which it points was resized or destroyed, or because it points past the end of the sequence.