Iterators in C++

1 Introduction

When a program needs to visit all of the elements of a vector named `myvec` starting with the first and ending with the last, you could use an *iterative loop* such as the following:

```
for ( int i = 0; i < myvec.size(); i++ )
    // the visit, i.e., do something with myvec[i]
```

This code works and is a perfectly fine solution. This same type of iterative loop can visit each character in a C++ string. But what if you need code that visits all elements of a C++ `list` object? Is there a way to use an iterative loop to do this? To be clear, an iterative loop is one that has the form

> *"for var = start value; var compared to end value; update* var; *{do something with node at specific index}"*

In other words, an iterative loop is a counting loop, as opposed to one that tests an arbitrary condition, like a while loop. The short answer to the question is that, in C++, without iterators, you cannot do this. Iterators make it possible to iterate through arbitrary containers. This set of notes answers the question, "What is an iterator, and how do you use it?"

Iterators are a generalization of pointers in C++ and have similar semantics. 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 class than one that can iterate through a `list`. We illustrate with an example.

Listing 1: A function to iterate through a list.

```
typedef list<string> strlist;     // strlist is a list of C++ strings

void addsuffixto_list( strlist & names, char* str )
{
    // Declare an iterator that can process lists of strings.
    // The list template class defines an iterator:
    strlist::iterator l_iter;

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

void print_list( const strlist & names )
{
    // Declare a constant iterator that can process a list of
    // const strings.
    // A const_iterator cannot be used to change what it points to
```

```
    strlist::const_iterator l_iter;

    // iterate through the list
    for ( l_iter = names.begin(); l_iter != names.end(); l_iter++ )
        cout << *l_iter << endl;
}
```

This example introduces and demonstrates several concepts:

- An iterator is specific to a container. In Listing 1, `l_iter` is declared to be of type

  ```
  list<string>::iterator
  ```

  This is because the only type of iterator that can sequence through an instance of a `list` template class is one that is specifically defined in the `list` template class. If the base type is `string`, then the iterator must be declared as one that works with the class `list<string>`.

- An iterator is advanced through a container by applying the increment operator (`++`) to it. This is just like the semantics of pointers. In this example, `++l_iter` advances it through the list. Either the pre- or post-increment operator can be used.

- Certain container classes, like the `list` class used here, have member functions such as `begin()` and `end()`. These return iterators. In particular, `begin()` returns an iterator that points to the first element of the list, and `end()` returns an iterator that points to an imaginary element one past the end of the list, which is called the ***past-the-end*** value.

- Iterators can be compared to each other, as in the condition `l_iter != names.end()`.

- Iterators can be assigned the values of other iterators, as in `l_iter = names.begin()`.

- In `print_list()`, the iterator is dereferenced by using the same dereference operator that is used with pointers. In this example, `*l_iter` is the string referenced by `l_iter`. In `addsuffixto_list()`, `l_iter->append()` is the member function called on the string pointed to by the iterator `l_iter`.

- There are iterators that can be used to change what they reference, and those that cannot. Constant iterators point to objects but cannot be used to change them.

- All containers provide a set of iterator operations. In these examples, 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 are implemented in order to use them, but you do need to understand what they point to, how they can be moved around, and what operations can be performed upon them. There is much to learn about iterators, but these notes are not intended to be comprehensive. They concentrate on the basics only.

2 The Basics

C++ defines five different categories of iterators: input iterators, output iterators, forward iterators, bi-directional iterators, and random access iterators. They form a class hierarchy, with successive categories inheriting the allowed operations from previous ones. Here, discussion is limited to bi-directional iterators, which are iterators that can move forward and backward incrementally through the range of elements in the container to which they are bound. (In contrast, a random access iterator can jump across its range using iterator arithmetic.)

An iterator `itr2` is called ***reachable*** from an iterator `itr1` if and only if there is a finite sequence of applications of the expression `++itr1` that makes `itr1 == itr2`. If `itr2` is reachable from `itr1`, they refer to elements of the same sequence. If `i` and `j` are two iterators and `j` is reachable from `i`, we define the ***range*** `[i,j)` to be the sequence of elements in the data structure starting with the element pointed to by `i` and up to but not including the element pointed to by `j`.

2.1 Bi-directional Iterator Types

There are different types of bi-directional iterators, even for a single container class. Basically, an iterator can be constant, meaning that it cannot be used to modify the object to which it points, or non-constant, in which case it can modify the object. Independently, an iterator can move in one of two directions: forward or backward. A backward-moving iterator is called a reverse iterator; incrementing it moves it backwards in its range. This leads to four different combinations of iterators, declared as the following C++ types:

`iterator` Increment moves it forward and it can modify referenced object

`const_iterator` Increment moves it forward but it cannot modify referenced object

`reverse_iterator` Increment moves it backward, and it can modify referenced object

`const_reverse_iterator` Increment moves it backward but it cannot modify referenced object

Standard containers, such as vectors, sets, lists, maps, and others, provide all of the above iterator types, as well as member functions that create these iterators. In addition, C++ strings, although not containers, provide iterators too. The following methods create iterators of various types and exist for all of the standard containers (like the ones just mentioned) that provide iterator types:

`begin()` returns an `iterator` that points to the first element of the sequence. If the container is a `const` object, the iterator returned is a `const_iterator`.

`end()` returns an iterator that is one element past the end of the sequence. If the container is a `const` object, the iterator returned is a `const_iterator`.

`rbegin()` returns a `reverse_iterator`, i.e., one that points to the last element of the sequence, and travels towards the first element as it is incremented. If the container is a `const` object, the iterator returned is a `const_reverse_iterator`.

`rend()` returns a `reverse_iterator` pointing to the element one before the first element in the sequence. 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. If the container is a `const` object, the iterator returned is a `const_reverse_iterator`.

`cbegin()` like `begin()`, but returns a `const_iterator` regardless of whether or not the container object is `const`-qualified.

`cend()` like `end()`, but returns a `const_iterator` that points to one-past-the-last-element regardless of whether or not the container object is `const`-qualified.

`crbegin()` like `rbegin()`, but returns a `const_reverse_iterator` that starts at the last element regardless of whether or not the container object is `const`-qualified.

`crend()` like `rend()`, but returns a `const_reverse_iterator` pointing to the element one before the first element in the container regardless of whether or not the container object is `const`-qualified.

There are containers such as stacks and queues that do not provide these methods, because as abstractions, they are not supposed to provide methods for traversing their contents. They are opaque to the client.

2.2 Iterator Operations

You saw above that iterators, like pointers, can be compared to each other, can be assigned the values of other iterators of the same container, and can be incremented. Bi-directional iterators can also be decremented. Not all iterators support the same set of operations, but for the above set of classes, the following operations are supported by all valid iterators. In the examples, assume that the iterators are declared to be members of a container object X. To be precise, the following definitions hold:

```
X::iterator iter1;
X::iterator iter2;
```

- To dereference an iterator `iter`, we use the same notation as dereferencing pointers, namely `*iter`. So for example, to copy the value referenced by `iter2` into the object referenced by `iter1`,

  ```
  *iter1 = *iter2;
  ```

  provided that the object to which `iter1` points allows modification (is non-const-qualified).

- To dereference and access a member, use `iter1->`. If `*iter1` has a member named `data`, then

  ```
  iter1->data
  ```

  is a dereference followed by a member access to `data`, just as is done with pointers.

- To advance an iterator, use either `++iter1` or `iter1++` .

- To advance it backwards, assuming `iter1` is bi-directional, use `--iter1` or `iter1--`.

- To dereference an iterator and then advance it, use `*iter1++` or to send it backward, use `*iter1--`.

2.3 Container Operations with Iterator Parameters

Some containers have member functions that require iterators as their parameters. For example, the method of the list class template (and the vector class template) to insert a single value into the list has the prototype

```
iterator insert (const_iterator position, const value_type& val);
```

where `position` is a bidirectional iterator at the position in the list where the new element is inserted and `val` is the value to be copied into the inserted element. `value_type` must match the type of the elements in the list. This function returns an iterator that points to the inserted element. The insert() method is overloaded and has many other forms. Consult the documentation for details.

Some other member functions that require iterators are the `erase()` methods, the `splice()` methods, and the `emplace()` methods.

2.4 Valid and Invalid Iterators

An iterator is ***valid*** if it references an element. An iterator that does not reference an exact element is said to be ***invalid*** (actually the C++ standard calls this state ***singular***.) There are several reasons that an iterator could be invalid:

- It was declared but not yet initialized.
- The element to which it pointed was removed as a result of some method such as `remove()` or `erase()`.

- The container within which it points was resized or destroyed.
- It points to the past-the-end value of a sequence.

This is the rule to remember: in general, when a container's member function modifies the container's structure, iterators, pointers and references referring to elements removed by the function are invalidated. All other iterators, pointers and references keep their validity. The following code illustrates this. It moves an iterator so that it lies within a range of list elements that are about to be deleted with the list's `erase()` method. The code dos not attemot to dereference the invalid iterator, but you can try modifying it and running it to see the error you will get. The example also points out that the `erase()` overload using iterators to specify a range makes the iterator at the bottom of the range invalid but not the one at the top of the range, since that element is not erased.

Listing 2: Example showing how an iterator becomes invalid.

```
#include <iostream>
#include <list>

int main ()
{
    std::list<int> mylist;
    std::list<int>::iterator iter1, iter2, iter3;

    // initialize some values:
    for (int i = 1; i < 10; ++i)
        mylist.push_back(i*10);
    // list looks like: 10 20 30 40 50 60 70 80 90

    iter1 = iter2 = iter3 = mylist.begin();

    for ( int i = 1; i <= 4; i++)
        iter2++;
    // Now iter2 points to 50

    for ( int i = 1; i <= 6; i++)
        iter3++;
    // Now iter3 points to 70

    mylist.erase (iter1,iter3);
    // erases 10,...,60 leaving list = 70 80 90
    // Now iter2 is invalid because it pointed to 50 which
    // was deleted, iter1 is invalid because it pointed to 10,
    // but iter3 is still valid.
    // As proof:
    std::cout << "mylist contains:";
    for (iter1 = iter3; iter1 != mylist.end(); ++iter1)
        std::cout << ' ' << *iter1;
    std::cout << '\n';

    // But we could also use this, because the list beginning
    // is automatically updated:
    std::cout << "mylist contains:";
    for (iter1 = mylist.begin(); iter1 != mylist.end(); ++iter1)
        std::cout << ' ' << *iter1;

    std::cout << '\n';
```

```
    return 0;
}
```

In the preceding listing, we advanced `iter2` in a fairly obvious way, by using a loop and incrementing it within the loop. We could have used the `advance()` function instead:

```
advance(iter2, 4);
```

would have moved `iter2` ahead by 4 elements in the sequence. If the second parameter would cause the iterator to go past the last element of the sequence, its behavior is undefined. Other functions that you might find useful are `next()`, `prev()`, `begin()`, and `end()`.