STL, Iterators, and Generic Algorithms*

Jeffrey D. Oldham

2000 Feb 02

Contents

1 Introduction to Iterators

An *iterator* is any C++ thing that permits accessing items in a container using certain specified operations. Customary uses include walking through all of a container's items, printing each item or changing it to uppercase. An iterator is not a type and there is no special C++ syntax for an iterator. If it looks like an iterator, acts like an iterator, and quacks like an iterator, then it is an iterator.

Iterators were developed to permit writing code that works on many different types of containers. Many algorithms consist of just going through a container and looking at all its elements. For example, printing the elements of a container should not depend on the container; it only requires the ability to read all the elements in a container, printing each one. Other algorithms such as sorting are more complicated but again should not depend (much) on the container in which the items are stored. Most fast sorting algorithms require the ability to quickly access any element in a container. Thus, we need more powerful operations than just being able to visit all elements in a container.

2 Containers

Since iterators were designed to permit using any type of container, let us first make containers more concrete. A *container* is a collection of items. For example, a Tupperware container may hold a collection of peas or carrots or tofu. A sandwich bag container usually holds only sandwiches but can sometimes hold potato chips. A shish kebab stick is another type of container. Some containers such as Tupperware containers permit accessing any element in the container. Others do not; one can remove food from a shish kebob stick only at one end or the other. As the examples illustrate, a container is not a type nor is there special C++ syntax. If it looks like a container, acts like a container, and quacks like a container, then it is a container.

Standard Template Library containers we have already seen or will soon see include `vector`s, `string`s, `pair`s, and hash tables. One can also consider a stream as a container. By definition, a vector is an array that can change its size as needed. See the brief introduction, including the sample code, to `vector`s. A string is just a sequence of characters. The corresponding STL container is called `string`. Some examples of operations on `string`s are available.

One thing we frequently want to do to the elements in a container is print each element. It would be nice to be able to write code like

`for_each(container.begin(), container.end(), print_element)`

that would work regardless of which container is used. This function call starts at the container's beginning and invokes the `print_element` function on each of the container's elements. In fact, writing C++ code is almost that simple. Here is code to print all the elements in a vector of characters.

```
// Print Every Character in a Vector

// First, we will fill the vector with characters read from the
// standard input.  Then, we will print each character.

#include <iostream.h>
#include <stdlib.h>              // has EXIT_SUCCESS
#include <algorithm>             // has for_each()
#include <vector>


// Print the specified element.
void print_element(const char c) {
  cout << c;
  return;
}

int main()
{
  vector<char> v;                // make a vector of characters
```

```
  // Store characters from standard input in the vector.
  char c;
  while (cin.get(c))
    v.push_back(c);             // grow vector one larger, storing c
                                // in the last position

  // Print the vector's characters.
  for_each(v.begin(), v.end(), print_element);

  return EXIT_SUCCESS;
}
```

Here is code to print all the elements in a string (of characters).

```
// Print Every Character in a String

// First, we will fill the string with characters read from the
// standard input.  Then, we will print each character.

#include <iostream.h>
#include <stdlib.h>             // has EXIT_SUCCESS
#include <algorithm>            // has for_each()
#include <string>


// Print the specified element.
void print_element(const char c) {
  cout << c;
  return;
}

int main()
{
  string v;             // make a string of characters

  // Store characters from standard input in the string.
  char c;
  while (cin.get(c))
    v += c;                     // grow string one larger, storing c
                                // in the last position

  // Print the string's characters.
  for_each(v.begin(), v.end(), print_element);

  return EXIT_SUCCESS;
}
```

The two programs differ very little. In fact, I created the string code by replacing all `vector` occurrences with `string` and also changing the `push_back` function to `+=`. We do not need to know the type of `v.begin()` or `v.end()`. The compiler figures it out for us.

`++iter`	move to the "next" item in the container
`iter++`	move to the "next" item in the container
`iter1 == iter2`	compare two iterators to see if they point to the same place
`iter1 != iter2`	compare two iterators to see if they point to different places
`*iter`	return the item pointed to by the iterator (read only)
`iter->member`	provide read access to a member (if any) of the current item
`TYPE(iter)`	copy an iterator

Table 1: Input Iterator Operations

3 Input Iterators

In the previous example, the same `for_each` function worked for `vector` and `string` containers. Since every C++ function corresponds to a single piece of code, we need to have common notation for walking through the contents of both `vectors` and `strings`. Let's sneak inside the implementation of `for_each` to see the notation and what is required of its iterator parameters.

```
template <class InputIterator, class UnaryFunction>
UnaryFunction for_each(InputIterator begin, InputIterator end,
                       UnaryFunction f) {
  if (begin == end)
    return f;
  else {
    f(*begin);
    for_each(++begin, end, f);
  }
}
```

`for_each` takes two input iterator parameters and one function parameter. The use of the template means the compiler figures out the type of the arguments so we do not have to write them explicitly. The first two arguments indicate the beginning and end of the container. *We require that the end is exactly one past the last element in the container.* If the container is empty, i.e., `first == last`, then we return the function argument. (The reason for returning the function argument is obscure; I have never used `for_each`'s return value.) Otherwise, we apply the function argument each item in the container, starting with the first.

Whenever we have a templatized parameter, we should ask what operations the templatized parameter is required to support. The only use of the function parameter `f` is applying it to a container element. The input iterators are more interesting. We require three operations:

`++begin`	move to the "next" item in the container
`begin == end`	compare two iterators to see if they point to the same place
`*begin`	read the item pointed to by the iterator

We can specify anything we want as the first and second arguments to `for_each` as long as they can do these three operations. In other words, an input iterator "lets us walk through the container looking at each element." We use the word "input" because, like regular keyboard input, we can only look at items sequentially, and we can only read, not change, them.

Table 1 has a complete list of operations *input iterators* are required to support. We will not discuss the last two operations for a while so do not worry about them; I included them only for completeness.

`++`*iter*	move to the "next" item in the container
iter`++`	move to the "next" item in the container
`*`*iter* `=` *value*	write the *value* into the current position
TYPE`(`*iter*`)`	copy an iterator

Table 2: Output Iterator Operations

4 Output Iterators

When printing items to output streams, we perform two operations: we want to write items to output and we want to move to the next position in the stream so we do not overwrite what we have just written. *Output iterators* for containers are similar. The operations we require are listed in Table 2. We are not permitted to read from an output iterator.

The `copy` function uses an output iterator as it copies elements from one container to another. Again, we go "under the hood" to see how `copy` works.

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result)
  if (first == last)
    return result;
  else {
    *result = *first;
    copy(++first, last, ++result);
    return;
  }
}
```

The operations on the first two iterators are `==`, `*first`, and `++first` so these two parameters can be input iterators. The operations on the last parameter are `*result =` *value* and `++result`. These match the requirements we required for output iterators so this last parameter can be an output iterator. Note that we need only specify the beginning of the destination container; we just assume that it will be large enough to store the copies.

Here is code for copying from a vector of characters to a string of characters.

```
// Copy Characters from a Vector to a String

// First, we will fill the vector with characters read from the
// standard input.  Then, we will copy the characters.  Then, we print
// the string.

#include <iostream.h>
#include <stdlib.h>              // has EXIT_SUCCESS
#include <algorithm>             // has copy()
#include <vector>
#include <string>


int main()
{
  vector<char> v;                // make a vector of characters
```

```
  // Store characters from standard input in the vector.
  char c;
  while (cin.get(c))
    v.push_back(c);             // grow vector one larger, storing c
                                // in the last position

  // Copy the vector's characters.
  string s;
  s.resize(v.size());           // make the string large enough to
                                // hold the characters
  copy(v.begin(), v.end(), s.begin());// copy the characters

  // Print the string's characters.
  cout << s;

  return EXIT_SUCCESS;
}
```

Since `copy` assumes that the destination string container is large enough, we have to `resize` it. See Section 11 for ways to automatically grow the string as the characters are inserted.

To summarize, an output iterator lets us walk through a container, writing each element.

5 Forward Iterators

Sometimes we want to walk through a container changing each item in the container. For example, we may want to fill a vector with zeroes. Here is an implementation for the `fill` function. In this function, the operations required by the iterator are `==`, `*first = value`, and `++first`.

```
template <class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value) {
  if (first == last)
    return;
  else {
    *first = value;
    fill(++first, last, value);
    return;
  }
}
```

Table 3 contains the complete list of operations required for *forward iterators*. Basically, this list is the union of operations on input and output iterators. We need not worry about `->` and the last two operations.

6 Bidirectional Iterators

Bidirectional iterators are forward iterators that also permit moving backwards one position at a time in the container. The `reverse` function requires bidirectional iterators. Bidirectional iterators support all the operations in Table 3 plus the two additional operations in Table 4.

`++iter`	move to the "next" item in the container
`iter++`	move to the "next" item in the container
`iter1 == iter2`	compare two iterators to see if they point to the same place
`iter1 != iter2`	compare two iterators to see if they point to different places
`*iter`	read or write to the item pointed to
`iter->member`	provide read access to a member (if any) of the current item
`iter1 = iter2`	assign an iterator
`TYPE()`	create a default iterator
`TYPE(iter)`	copy an iterator

Table 3: Forward Iterator Operations

`--iter`	move to the "previous" item in the container
`iter--`	move to the "previous" item in the container

Table 4: Additional Operations of Bidirectional Iterators

7 Random Access Iterators

Random access iterators are bidirectional iterators that can access any point in the container. In a constant amount of time, one can move to any position in a container using a random access iterator. This is useful when sorting or performing binary search.

Table 5 contains the additional operations that random access iterators provide. Random access iterators provide no additional power over bidirectional iterators except possibly faster performance.

8 How Do I Remember All These Iterators?

Input iterator: Walk through the container left-to-right reading items.
Question to ask: Is it like reading from the keyboard?

Output iterator: Walk through the container left-to-right writing items.
Question to ask: Is it like writing to the screen?

`iter+n`	returns the iterator n positions forward
`n+iter`	returns the iterator n positions forward
`iter-n`	returns the iterator n positions backward
`iter+=n`	moves iterator n positions forward
`iter-=n`	moves iterator n positions backward
`iter1-iter2`	returns the signed distance between the two iterators
`iter1<iter2`	returns whether *iter1* is before *iter2*
`iter1<=iter2`	returns whether *iter1* is not after *iter2*
`iter1>=iter2`	returns whether *iter1* is not after *iter2*
`iter1>iter2`	returns whether *iter1* is before *iter2*
`iter[n]`	returns the element n positions to the right

Table 5: Additional Operations of Random Access Iterators

Forward iterator: Walk through the container left-to-right reading and writing items.
Question to ask: Are the items read and changed as we walk left-to-right?

Bidirectional iterator: Walk through the container in any order reading and writing items.
Question to ask: Are the items read and changed as we walk through the container both left-to-right and right-to-left?

Random access iterator: Jump around the container reading and writing items.
Question to ask: Do we jump around in the container?

9 How Do I Create and Use Iterators?

There are two easy ways to create iterators:

1. For any STL container `c`, `c.begin()` and `c.end()` yield iterators at its beginning and end. (Remember that `c.end()` points one position past the container's last item.) All STL containers are required to provide these functions.

2. *container*`::const_iterator pos1` and *container*`::iterator pos2` create iterator variables for the specified *container*.

In the latter case, we explicitly declare iterators called `pos1` and `pos2`. The type is the name of the container followed by `::const_iterator` or `::iterator`. Here is code using this notation.

```
// Print Every Character in a Vector

// First, we will fill the vector with characters read from the
// standard input.  Then, we will print each character.

#include <iostream.h>
#include <stdlib.h>             // has EXIT_SUCCESS
#include <algorithm>            // has for_each()
#include <vector>


// Print the specified element.
void print_element(const char c) {
  cout << c;
  return;
}

int main()
{
  vector<char> v;               // make a vector of characters

  // Store characters from standard input in the vector.
  char c;
  while (cin.get(c))
    v.push_back(c);             // grow vector one larger, storing c
                                // in the last position

  // Print the vector's characters.
  // explicitly use iterators, not "for_each(v.begin(), v.end(), print_element);"
```

```
  vector<char>::const_iterator pos;// explicit declaration of an iterator
  for (pos = v.begin(); pos != v.end(); ++pos)
    print_element(*pos);

  return EXIT_SUCCESS;
}
```

The code `vector<char>::const_iterator pos;` explicitly declares an iterator for vectors of `char`s. The for loop applies the function to each item in the container.

9.1 const_iterators and iterators

In the spirit of C++, we complicate the syntax to enable the programmer to write faster code. A *const_iterator* is an "constant" iterator. That is, it permits walking through a container but gives only read access to container items. For example, one can `++iter` and `*iter`, but not `*iter = value`.

Why use a const_iterator when an iterator will always work? First, we must use const_iterators to walk through constant containers. This frequently happens when a container is a `const` function parameter. Second, using `const`s can permit the compiler to write faster code. While code speed is not important for this class, it is a good habit to develop.

10 Istream and Ostream Iterators

Since input and output streams are just collections of data, we can consider them as containers under the "if it looks like a container, acts like a container, and quacks like a container, then it is a container" rule. We only need iterators for streams.

To treat an `istream in` as an iterator, use `istream_iterator<T>(in)`, where items of type T are to be read from the istream. The end-of-istream iterator is created using `istream_iterator<T>()`.

To treat an `ostream out` as an iterator, use `ostream_iterator<T>(out)`, where items of type T are to be read from the ostream. There is no end-of-ostream idea so there is not an analogous end-of-ostream iterator. To have a string *delim* printed after item is printed, use `ostream_iterator<T>(out, delim)`.

Be sure to `#include<iterator>` to use these iterator adaptors.

Here is code to copy `string`s from the standard input to the standard output, printing them one per line.

```
// Copy a List of Strings from Cin to Cout, one per line

#include <iostream.h>
#include <stdlib.h>             // has EXIT_SUCCESS
#include <string>
#include <iterator>             // has input_iterator
#include <algorithm>            // has copy

int main()
{
  copy(istream_iterator<string>(cin), istream_iterator<string>(),
       ostream_iterator<string>(cout, "\n"));
  return EXIT_SUCCESS;
}
```

To treat the input as a container, we specify its beginning and ending using the first two arguments to `copy`. The last argument specifies each `string` is sent to `cout` separated by a newline character. For more information, read more about istream iterators and ostream iterators.

11 Inserter Iterators

You are not responsible for knowing this material. In Section 4, we copied from a vector to a string, but we had to resize the string to ensure it was large enough to hold all the vector's characters. If the output iterator automatically enlarged the container every time a new item is added, we would not have to preallocate the container. In other words, we wish to insert elements using `push_back`. To do so, we can use an insert iterator like `back_inserter(`*`container`*`)`.

Here is some code I claim is correct, but the current compiler does not have `push_back` for `string`s. This shows how very close to the cutting edge of C++ programming we are. Instead, we can copy from a `string` to a `vector`.

```
// Copy Characters from a String to a Vector

// First, we will fill the vector with characters read from the
// standard input.  Then, we will copy the characters.  Then, we print
// the string.

#include <iostream.h>
#include <stdlib.h>             // has EXIT_SUCCESS
#include <algorithm>            // has copy()
#include <vector>
#include <string>


int main()
{
  string s;                     // make a string

  // Store characters from standard input in the vector.
  char c;
  while (cin.get(c))
    s += c;                     // grow string one larger, storing c
                                // in the last position

  // Copy the string's characters.
  vector<char> v;
  // unnecessary to v.resize(s.size());
  copy(s.begin(), s.end(), back_inserter(v));
                        // copy the characters, enlarging v as we go

  // Print the string's characters.
  copy(v.begin(), v.end(), ostream_iterator<char>(cout));

  return EXIT_SUCCESS;
}
```

After reading all the characters into a `string`, these are `copy`ied into the vector `v`, which is enlarged by the `back_inserter` as characters are added.

For containers having a `push_front` operation, one can use `front_inserter(`*`container`*`)`. To read more, see back inserter and front inserter WWW pages.

12 Read More About It

All other explanations of iterators known to me rely on readers already knowing about pointers even though it is not necessary to know about pointers to understand iterators. (In fact, I purposefully choose to introduce iterators before pointers.) The three best expositions I have found are:

- *The C++ Standard Library*, by Nicolai M. Josuttis, chapter 7.
- Introduction to the Standard Template Library by SGI.
- *C++ Ruminations*, by Koenig and Moo, chapter 18.

See the course syllabus for more information about the two books.