How Inserters Work

David Kieras
Prepared for EECS 381, Fall 2004

A back_inserter allows you to copy into an empty vector using the copy algorithm as follows:

```
#include <vector>
#include <iterator>
#include <algorithm>

vector<int> src, dest;
// fill src with some values

// copy the src into the empty vector dest
copy(src.begin(), src.end(), back_inserter(dest));
```

This works because the `back_inserter` results in an object that looks like an iterator, and so can participate in the algorithm like an ordinary iterator, but when the dereferenced iterator is assigned to, the result is that a `push_back` operation is performed on the container instead of the usual assignment to the place the iterator points to. How is this done? This explanation is in terms of `back_inserter` for a `vector`, but the same principles apply to the other inserters and containers.

Let's start with the an implementation of the copy algorithm function template:

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

Clearly all this does is assign each element pointed to by iterator `first`, to the dereferenced iterator `result`, incrementing both `first` and result each time around the loop. If `result` is pointing into a container with space already existing at each iterator position, the algorithm behaves very simply. But if the destination container is empty, the result is undefined and probably will crash. Doing a `push_back` on the container should be used instead of the assignment. The key is a class that provides definitions of `operator*`, `operator++`, and `operator=` that are compatible with being used as an iterator, and is initialized with the container to be filled with a `push_back`. In this implementation, the class is called `back_insert_iterator`, and has the following declaration (somewhat simplified):

```
template <class Container>
class back_insert_iterator {
public:
explicit back_insert_iterator(Container& x) : container(&x) {}
back_insert_iterator& operator=(typename Container::const_reference value)
	{
		container->push_back(value);
		return *this;
	}
```

```
back_insert_iterator& operator*()
        {return *this;}
back_insert_iterator& operator++()
        {return *this;}
back_insert_iterator& operator++(int)
        {return *this;}
private:
        Container* container;
};
```

There is only one template parameter, `Container`, for the container type (`vector<int>` in this example). The constructor for this class simply stores a pointer to the supplied container. The dereference operator and increment operators couldn't be simpler - they simply return a reference to this `back_insert_iterator` object. The assignment operator takes a r.h.s. parameter whose type is supplied by one of the standard container typedefs, for a const reference to the type of value stored in the container (`const int&` in this example). The assignment operator simply does a `push_back` of the r.h.s. value into the container whose pointer has been stored.

You could create a `back_insert_iterator` object using this template directly, with the template argument being the container type, and the constructor argument being the destination container, and then use it as an iterator in the copy algorithm:

```
back_insert_iterator<vector<int> >bi_it(dest),
copy(src.begin(), src.end(), bi_it);
```

The copy function template will get instantiated as:

```
back_insert_iterator<vector<int> >
copy(vector<int>::iterator first, vector<int>::iterator last,
back_insert_iterator<vector<int> > result)
{
        for (; first != last; ++first, ++result)
                *result = *first;
        return result;
}
```

See how the `++result` does nothing. Likewise, the `*result` just becomes `result`. But the assignment `*result = *first` executes as `dest.push_back(*first)`. How about that! Amazing what can be accomplished with code that does almost nothing!

One last bit to simplify using the template. Note how there is some redundancy in how we create the `back_insert_iterator` object - we have to specify both the specific container, `dest`, which is a `vector<int>`, and also the type of the container, which is `vector<int>`. This clunkiness is solved with Template Magic Trick #1: *Use a function template to create and return a class template object.* The compiler infers the template type parameter from the type of the function argument. The function template is `back_inserter`, defined as:

```
template <class Container>
inline
back_insert_iterator<Container>
back_inserter(Container& x)
{
        return back_insert_iterator<Container>(x);
}
```

This function template is instantiated with the type of the parameter, `Container`, and simply creates and returns a `back_insert_iterator` object, instantiating that template with the same type. Now instead of the tedious direct invocation of the `back_insert_iterator` template, you simply write:

```
back_inserter(dest);
```

The compiler sees that `dest` is a `vector<int>`, so it knows that the template parameter `Container` is `vector<int>`, so it creates the same tedious code that you didn't want to write. Finally, because the function template is also declared inline, the function call goes "poof", meaning that the final result is exactly as if you had written:

```
copy(src.begin(), src.end(), back_insert_iterator<vector<int> >(dest));
```

Note: This explanation is based on somewhat simplified declarations from the Metrowerks version of the C++ Standard Library. Other implementation may differ in details or specifics, but the overall principle is the same.