The following is an excerpt from Scott Meyers' new book, *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library.*

Item 2: Beware the illusion of container-independent code.

The STL is based on generalization. Arrays are generalized into containers and parameterized on the types of objects they contain. Functions are generalized into algorithms and parameterized on the types of iterators they use. Pointers are generalized into iterators and parameterized on the type of objects they point to.

That's just the beginning. Individual container types are generalized into sequence and associative containers, and similar containers are given similar functionality. Standard contiguous-memory containers (see Item 1) offer random-access iterators, while standard node-based containers (again, see Item 1) provide bidirectional iterators. Sequence containers support push_front and/or push_back, while associative containers don't. Associative containers offer logarithmic-time lower_bound, upper_bound, and equal_range member functions, but sequence containers don't.

With all this generalization going on, it's natural to want to join the movement. This sentiment is laudable, and when you write your own containers, iterators, and algorithms, you'll certainly want to pursue it. Alas, many programmers try to pursue it in a different manner. Instead of committing to particular types of containers in their software, they try to generalize the notion of a container so that they can use, say, a vector, but still preserve the option of replacing it with something like a deque or a list later — all without changing the code that uses it. That is, they strive to write *container-independent code*. This kind of generalization, well-intentioned though it is, is almost always misguided.

Even the most ardent advocate of container-independent code soon realizes that it makes little sense to try to write software that will work with both sequence and associative containers. Many member functions exist for only one category of container, e.g., only sequence containers support push_front or push_back, and only associative containers support count and lower_bound, etc. Even such basics as insert and erase have signatures and semantics that vary from category to category. For example, when you insert an object into a sequence container, it stays where you put it, but if you insert an object into an associative container, the container moves the object to where it belongs in the container's sort order. For another example, the form of erase taking an iterator returns a new iterator when invoked on a sequence container, but it returns nothing when invoked on an associative container. (Item 9 gives an example of how this can affect the code you write.)

Suppose, then, you aspire to write code that can be used with the most common sequence containers: vector, deque, and list. Clearly, you must program to the intersection of their capabilities, and that means no uses of reserve or capacity (see Item 14), because deque and list don't offer them. The presence of list also means you give up operator[], and you limit yourself to the capabilities of bidirectional iterators. That, in turn, means you must stay away from algorithms that demand random access iterators, including sort, stable_sort, partial_sort, and nth_element (see Item 31).

On the other hand, your desire to support vector rules out use of push_front and pop_front, and both vector and deque put the kibosh on splice and the member form of sort. In conjunction with the constraints above, this latter prohibition means that there is no form of sort you can call on your "generalized sequence container."

That's the obvious stuff. If you violate any of those restrictions, your code will fail to compile with at least one of the containers you want to be able to use. The code that *will* compile is more insidious.

The main culprit is the different rules for invalidation of iterators, pointers, and references that apply to different sequence containers. To write code that will work correctly with vector, deque, and list, you must assume that any operation invalidating iterators, pointers, or references in any of those containers invalidates them in the container you're using. Thus, you must assume that every call to insert invalidates everything, because deque::insert invalidates all iterators and, lacking the ability to call capacity, vector::insert must be assumed to invalidate all pointers and references. (Item 1 explains that deque is unique in sometimes invalidating its iterators without invalidating its pointers and references.) Similar reasoning leads to the conclusion that every call to erase must be assumed to invalidate everything.

Want more? You can't pass the data in the container to a C interface, because only vector supports that (see Item 16). You can't instantiate your container with bool as the type of objects to be stored, because, as Item 18 explains, vector<bool> doesn't always behave like a vector, and it never actually stores bools. You can't assume list's constanttime insertions and erasures, because vector and deque take linear time to perform those operations.

When all is said and done, you're left with a "generalized sequence container" where you can't call reserve, capacity, operator[], push_front, pop_front, splice, or any algorithm requiring random access iterators; a container where every call to insert and erase takes linear time and invalidates all iterators, pointers, and references; and a container incompatible with C where bools can't be stored. Is that really the kind of container you want to use in your applications? I suspect not.

If you rein in your ambition and decide you're willing to drop support for list, you still give up reserve, capacity, push_front, and pop_front; you still must assume that all calls to insert and erase take linear time and invalidate everything; you still lose layout compatibility with C; and you still can't store bools.

If you abandon the sequence containers and shoot instead for code that can work with different associative containers, the situation isn't much better. Writing for both set and map is close to impossible, because sets store single objects while maps store pairs of objects. Even writing for both set and multiset (or map and multimap) is tough. The insert member function taking only a value has different return types for sets/maps than for their multi cousins, and you must religiously avoid making any assumptions about how many copies of a value are stored in a container. With map and multimap, you must avoid using operator[], because that member function exists only for map.

Face the truth: it's not worth it. The different containers are *different*, and they have strengths and weaknesses that vary in significant ways. They're not designed to be interchangeable, and there's little you can do to paper that over. If you try, you're merely tempting fate, and fate doesn't like to be tempted.

Still, the day will dawn when you'll realize that a container choice you made was, er, suboptimal, and you'll need to use a different container type. You now know that when you change container types, you'll not only need to fix whatever problems your compilers diagnose, you'll also need to examine all the code using the container to see what needs to be changed in light of the new container's performance characteristics and rules for invalidation of iterators, pointers, and references. If you switch from a vector to something else, you'll also have to make sure you're no longer relying on vector's C-compatible memory layout, and if you switch to a vector, you'll have to ensure that you're not using it to store bools.

Given the inevitability of having to change container types from time to time, you can facilitate such changes in the usual manner: by encapsulating, encapsulating, encapsulating. One of the easiest ways to do this is through the liberal use of typedefs for container and iterator types. Hence, instead of writing this,

```
class Widget { ... };
vector<Widget> vw;
Widget bestWidget;
...                                          // give bestWidget a value
vector<Widget>::iterator i =                 // find a Widget with the
  find(vw.begin(), vw.end(), bestWidget);    // same value as bestWidget
```

write this:

```
class Widget { ... };
typedef vector<Widget> WidgetContainer;
typedef WidgetContainer::iterator WCIterator;
WidgetContainer vw;
Widget bestWidget;
...
WCIterator i = find(vw.begin(), vw.end(), bestWidget);
```

This makes it a lot easier to change container types, something that's especially convenient if the change in question is simply to add a custom allocator. (Such a change doesn't affect the rules for iterator/pointer/reference invalidation.)

```
class Widget { ... };
template<typename T>                          // see Item 10 for why this
SpecialAllocator { ... };                     // needs to be a template
typedef vector<Widget, SpecialAllocator<Widget> > WidgetContainer;
typedef WidgetContainer::iterator WCIterator;
WidgetContainer vw;                                     // still works
Widget bestWidget;
...
WCIterator i = find(vw.begin(), vw.end(), bestWidget);  // still works
```

If the encapsulating aspects of typedefs mean nothing to you, you're still likely to appreciate the work they can save. For example, if you have an object of type

```
map< string,
        vector<Widget>::iterator,
        CIStringCompare>                    // CIStringCompare is "case-
                                            // insensitive string compare;"
                                            // Item 19 describes it
```

and you want to walk through the map using const_iterators, do you really want to spell out

```
map<string, vector<Widget>::iterator, CIStringCompare>::const_iterator
```

more than once? Once you've used the STL a little while, you'll realize that typedefs are your friends.

A typedef is just a synonym for some other type, so the encapsulation it affords is purely lexical. A typedef doesn't prevent a client from doing (or depending on) anything they couldn't already do (or depend on). You need bigger ammunition if you want to limit client exposure to the container choices you've made. You need classes.

To limit the code that may require modification if you replace one container type with another, hide the container in a class, and limit the amount of container-specific information visible through the class interface. For example, if you need to create a customer list, don't use a list directly. Instead, create a CustomerList class, and hide a list in its private section:

```
class CustomerList {
private:
  typedef list<Customer> CustomerContainer;
  typedef CustomerContainer::iterator CCIterator;

  CustomerContainer customers;

public:                                  // limit the amount of list-specific
  ...                                    // information visible through
};                                       // this interface
```

At first, this may seem silly. After all a customer list is a *list*, right? Well, maybe. Later you may discover that you don't need to insert or erase customers from the middle of the list as often as you'd anticipated, but you do need to quickly identify the top 20% of your customers — a task tailor-made for the nth_element algorithm (see Item 31). But nth_element requires random access iterators. It won't work with a list. In that case, your customer "list" might be better implemented as a vector or a deque.

When you consider this kind of change, you still have to check every CustomerList member function and every friend to see how they'll be affected (in terms of performance and iterator/pointer/reference invalidation, etc.), but if you've done a good job of encapsulating CustomerList's implementation details, the impact on CustomerList clients should be small. *You* can't write container-independent code, but *they* might be able to.