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 16: Know how to pass vector and string data to legacy APIs.

Since C++ was standardized in 1998, the C++ elite haven't been terribly subtle in their attempt to nudge programmers away from arrays and towards vectors. They've been similarly overt in trying to get developers to shift from char* pointers to string objects. There are good reasons for making these changes, including the elimination of common programming errors (see Item 13) and the ability to take full advantage of the power of the STL algorithms (see, e.g., Item 31).

Still, obstacles remain, and one of the most common is the existence of legacy C APIs that traffic in arrays and char* pointers instead of vector and string objects. Such APIs will exist for a long time, so we must make peace with them if we are to use the STL effectively.

Fortunately, it's easy. If you have a vector v and you need to get a pointer to the data in v that can be viewed as an array, just use &v[0]. For a string s, the corresponding incantation is simply s.c_str(). But read on. As the fine print in advertising often points out, certain restrictions apply.

Given

```
vector<int> v;
```

the expression v[0] yields a reference to the first element in the vector, so &v[0] is a pointer to that first element. The elements in a vector are constrained by the C++ Standard to be stored in contiguous memory, just like an array, so if we wish to pass v to a C API that looks something like this,

```
void doSomething(const int* pInts, size_t numInts);
```

we can do it like this:

```
doSomething(&v[0], v.size());
```

Maybe. Probably. The only sticking point is if v is empty. If it is, v.size() is zero, and &v[0] attempts to produce a pointer to something that does not exist. Not good. Undefined results. A safer way to code the call is this:

```
if (!v.empty()) {
  doSomething(&v[0], v.size());
}
```

If you travel in the wrong circles, you may run across shady characters who will tell you that you can use v.begin() in place of &v[0], because (these loathsome creatures will tell you) begin returns an iterator into the vector, and for vectors, iterators are really pointers. That's often true, but as Item 50 reveals, it's not always true, and you should never rely on it. The return type of begin is an iterator, not a pointer, and you should never use begin when you need to get a pointer to the data in a vector. If you're determined to type v.begin() for some reason, type &*v.begin(), because that will yield the same pointer as &v[0], though it's more work for you as a typist and more obscure for people trying to make sense of your code. Frankly, if you're hanging out with people who tell you to use v.begin() instead of &v[0], you need to rethink your social circle.

The approach to getting a pointer to container data that works for vectors isn't reliable for strings, because (1) the data for strings are not guaranteed to be stored in contiguous memory, and (2) the internal representation of a string is not guaranteed to end with a null character. This explains the existence of the string member function c_str, which returns a pointer to the value of the string in a form designed for C. We can thus pass a string s to this function,

```
void doSomething(const char *pString);
```

like this:

```
doSomething(s.c_str());
```

This works even if the string is of length zero. In that case, c_str will return a pointer to a null character. It also works if the string has embedded nulls. If it does, however, doSomething is likely to interpret the first embedded null as the end of the string. string objects don't care if they contain null characters, but char*-based C APIs do.

Look again at the doSomething declarations:

```
void doSomething(const int* pInts, size_t numInts);
void doSomething(const char *pString);
```

In both cases, the pointers being passed are pointers to const. The vector or string data are being passed to an API that will *read* it, not modify it. This is by far the safest thing to do. For strings, it's the only thing to do, because there is no guarantee that c_str yields a pointer to the internal representation of the string data; it could return a pointer to an unmodifiable *copy* of the string's data, one that's correctly format-

ted for a C API. (If this makes the efficiency hairs on the back of your neck rise up in alarm, rest assured that the alarm is probably false. I don't know of any contemporary library implementation that takes advantage of this latitude.)

For a vector, you have a little more flexibility. If you pass v to a C API that modifies v's elements, that's typically okay, but the called routine must not attempt to change the number of elements in the vector. For example, it must not try to "create" new elements in a vector's unused capacity. If it does, v will become internally inconsistent, because it won't know its correct size any longer. v.size() will yield incorrect results. And if the called routine attempts to add data to a vector whose size and capacity (see Item 14) are the same, truly horrible things could happen. I don't even want to contemplate them. They're just too awful.

Did you notice my use of the word "typically" in the phrase "that's typically okay" in the preceding paragraph? Of course you did. Some vectors have extra constraints on their data, and if you pass a vector to an API that modifies the vector's data, you must ensure that the additional constraints continue to be satisfied. For example, Item 23 explains how sorted vectors can often be a viable alternative to associative containers, but it's important for such vectors to remain sorted. If you pass a sorted vector to an API that may modify the vector's data, you'll need to take into account that the vector may no longer be sorted after the call has returned.

If you have a vector that you'd like to initialize with elements from a C API, you can take advantage of the underlying layout compatibility of vectors and arrays by passing to the API the storage for the vector's elements:

```
// C API: this function takes a pointer to an array of at most arraySize
// doubles and writes data to it. It returns the number of doubles written,
// which is never more than maxNumDoubles.
size_t fillArray(double *pArray, size_t arraySize);

vector<double> vd(maxNumDoubles);           // create a vector whose
                                            // size is maxNumDoubles

vd.resize(fillArray(&vd[0], vd.size()));    // have fillArray write data
                                            // into vd, then resize vd
                                            // to the number of
                                            // elements fillArray wrote
```

This technique works only for vectors, because only vectors are guaranteed to have the same underlying memory layout as arrays. If you want to initialize a string with data from a C API, however, you can do

it easily enough. Just have the API put the data into a vector<char>, then copy the data from the vector to the string:

```
// C API: this function takes a pointer to an array of at most arraySize
// chars and writes data to it. It returns the number of chars written,
// which is never more than maxNumChars.
size_t fillString(char *pArray, size_t arraySize);

vector<char> vc(maxNumChars);                    // create a vector whose
                                                 // size is maxNumChars

size_t charsWritten = fillString(&vc[0], vc.size());   // have fillString write
                                                       // into vc

string s(vc.begin(), vc.begin()+charsWritten);   // copy data from vc to s
                                                 // via range constructor
                                                 // ( see Item 5)
```

In fact, the idea of having a C API put data into a vector and then copying the data into the STL container you really want it in always works:

```
size_t fillArray(double *pArray, size_t arraySize);      // as above

vector<double> vd(maxNumDoubles);                        // also as above
vd.resize(fillArray(&vd[0], vd.size());

deque<double> d(vd.begin(), vd.end());                   // copy data into
                                                         // deque

list<double> l(vd.begin(), vd.end());                    // copy data into list

set<double> s(vd.begin(), vd.end());                     // copy data into set
```

Furthermore, this hints at how STL containers other than vector or string can pass their data to C APIs. Just copy each container's data into a vector, then pass it to the API:

```
void doSomething(const int* pInts, size_t numInts);   // C API (from above)

set<int> intSet;                                       // set that will hold
...                                                    // data to pass to API

vector<int> v(intSet.begin(), intSet.end());           // copy set data into
                                                       // a vector

if (!v.empty()) doSomething(&v[0], v.size());          // pass the data to
                                                       // the API
```

You could copy the data into an array, too, then pass the array to the C API, but why would you want to? Unless you know the size of the container during compilation, you'd have to allocate the array dynamically, and Item 13 explains why you should prefer vectors to dynamically allocated arrays anyway.