Date: February 22, 1999
Project: ISO C++
Doc. No.: J16/99-0008 = WG21 N1185
Reply To: Herb Sutter
(*hsutter@peerdirect.com*)

# vector<bool> Is Nonconforming, and Forces Optimization Choice

## Discussion

Consider the following code:

```
// Example 1: Works for every T except bool
//
template<class T>
void g( vector<T>& v ) {
  T& r = v.front();
  T* p = &*v.begin();
  // ... do something with r and *p ...
}
```

The reason for the anomaly is that the `vector<bool>` specialization (**23.2.5, lib.vector.bool**) does not meet the standard's container or sequence requirements. Nevertheless, the specialization appears in Clause 23, with no warning that it is in fact neither a container nor a sequence.

The `vector<bool>` specialization was put into the standard to provide an example of how to write a proxied container. It's clear that proxied collections can be useful; the usual example is a disk-based collection. So the idea was to show how to make such a proxied collection meet the library's container requirements.

Unfortunately, the container and iterator requirements do not allow proxied containers: a `container<T>::reference` must be a true lvalue of type `T`. Iterators have similar requirements: dereferencing a forward, bidirectional, or random-access iterator must yield a `T&`. Proxy-based collections are a good and useful technique; such collections just can't be "containers" in the sense required by the standard, that's all. Further, although it is strongly implied that `vector<bool>::iterator` is a random-access iterator, it is not; this means that it is possible to write a conforming implementation of standard library algorithms that will nevertheless not work with `vector<bool>`.

Note that the original STL's container requirements and the C++ standard's container requirements are based on the implicit assumption (among others) that dereferencing an iterator both is a constant-time operation and requires negligible time compared to other operations. As James Kanze correctly pointed out on the *comp.std.c++* newsgroup in early 1997, neither assumption holds for a disk-based container or a packed-representation container. [For all the gory details, do a DejaNews search for Subject="vector and bool" and Forum="*c++*". The discussions took place in Jan/Feb 1997. Note also the more recent discussions from users asking how to turn off the `vector<bool>` specialization.] Consider that one would be unlikely to apply a standard algorithm like `std::find()` to a disk-based container; the performance would be abysmal compared to a special-purpose replacement, largely because the fundamental performance assumptions for in-memory containers do not apply to disk-based containers. (For similar reasons, even in-memory containers like `std::map` provide their own `find` as a member function.) For a packed-representation container like `vector<bool>`, access through the proxy object requires bitwise operations, which are generally much slower than manipulation of a native type like an `int`. Further, whenever the proxy object construction and destruction can't be completely optimized away by the compiler, managing the proxies themselves adds further overhead.

Finally, `vector<bool>` was intended as an optimization. Unfortunately, is not a pure optimization, but a tradeoff that favors "less space" at the expense of "potentially slower speed." Normally, we teach users to optimize only after empirical evidence (e.g., a profiler's output) demonstrates the need in their particular application; otherwise, the optimization is probably premature. The most premature optimization of all is an optimization that's enshrined in the standard. In this case, `vector<bool>` forces the "favour less space at the expense of potentially slower speed" optimization choice on all programs. The implicit assumption is that virtually all users of a `vector` of `bool`s will prefer "less space" at the expense of "potentially slower speed," that they will be more space-constrained than performance-constrained. This is clearly untrue.

## Summary of Problems and Issues

In summary, the problems/issues are:

1. `vector<bool>` does not meet the container or sequence requirements.
2. `vector<bool>::iterator` does not meet the requirements of a forward, bidirectional, or random-access iterator, although the last is strongly implied by the specialization's naming and position. This means that it may not work with a conforming implementation of a standard library algorithm.
3. It is misleading that `vector<bool>` appears in Clause 23 without a note to indicate that it's really neither a container nor a sequence.
4. `vector<bool>` attempts to illustrate how to write standard-conforming proxied containers. Unfortunately, that appears not to be a sound idea, for two reasons:
   - Although a proxied collection can be an important and useful tool, by definition it must violate the standard's current container requirements and therefore can never be a conforming container. (See #1.)
   - The main reason to conform to the standard container requirements is to be used with the standard algorithms, yet the standard algorithms are typically inappropriate for proxied containers because proxied containers have different performance characteristics than plain-and-in-memory containers.
5. `vector<bool>`'s name is misleading because the things inside aren't `bool`s.

6. `vector<bool>` forces a specific optimization choice on all users by enshrining it in the standard. That's probably not a good idea, even if the actual performance overhead turns out to be negligible for a given compiler for most applications; different users have different requirements.

## Proposed Resolution

1. Deprecate 23.2.5 [lib.vector.bool]. (The more desirable solution is to remove 23.2.5 entirely, but this is probably not possible until the next revision of the standard.)
2. Add a warning to 23.2.5 that `vector<bool>` does not meet the container or sequence requirements, and that `vector<bool>::iterator` does not meet the requirements of a forward, bidirectional, or random-access iterator.

*end of text*