Date: October 20, 1999
Project: ISO C++
Doc. No.: J16/99-0035 = WG21 N1211
Reply To: Herb Sutter
(*hsutter@peerdirect.com*)

vector<bool>: More Problems, Better Solutions

Summary of J16/99-0008 = WG21 N1185

The major issues covered in the earlier paper were:

1. Although the standard lists `vector<bool>` in Clause 23, `vector<bool>` is not a container and `vector<bool>::iterator` is not a random-access iterator (or even a forward or bidirectional iterator either, for that matter). This has already broken user code in the field in mysterious ways. The motivating example from the previous paper was:

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

2. `vector<bool>` forces a specific (and potentially bad) optimization choice on all users by enshrining it in the standard. The optimization is premature; different users have different requirements. This too has already hurt users who have been forced to implement workarounds to disable the 'optimization' (e.g., by using a `vector<char>` and manually casting to/from `bool`).
3. Yet if we try to fix Problem #1 or #2 we risk breaking code that relies on `vector<bool>`'s space or performance characteristics and/or its `flip()` interface.

New Information

Further discussions have brought the following problems to light:

4. Curiously, `vector<bool>` is not actually specified, so no current use of it invokes well specified behavior. Its declaration appears in the standard, but not a single function is specified. Note that the argument "it's just the same as `vector`" fails because a `vector<bool>` is demonstrably not a vector: it has a different interface (i.e., `flip()`), a different structure (e.g., `reference` is a class, not a typedef for `T&`), does not meet the same requirements (e.g., container and iterator requirements), etc.

   There are only two ways to fix this particular problem: Complete the specification of `vector<bool>`, or remove (not just deprecate) it.
5. Following from the previous point: Since in particular `vector<bool>::flip()` and `vector<bool>::reference::flip()` are actually not specified, it could be argued that Problem #3 is not a problem at all; that is, even if we removed `vector<bool>` outright it could only break user code that relied on incompletely specified behavior anyway.

Discussion

The container requirements do not allow proxied containers, and a packed representation for `vector<bool>` is possible only with proxies. Therefore I think the only known resolutions to this problem fall into the following categories:

1. *Status quo.* I now think this is not possible, because `vector<bool>` shouldn't be left underspecified.
2. *Document the status quo for `std::vector<bool>`, and finish specifying it.* As noted in Dublin, adding a note that documents Problem #1 is equivalent to saying we made a mistake and we're proud of it. Also, we would still have to actually specify `vector<bool>`. Finally, this does not solve Problem #2.
3. *Fix Problem #1 but keep the packed-representation `std::vector<bool>` under that name, and finish specifying it.* This is not possible, because the only way to do it is to loosen the container and iterator requirements in a way that will break much more existing code (e.g., drop the requirement that `container<T>::reference` be a `T&`). Although it was proposed in Dublin that we might create new requirements for a "proxyable container," even if this definition existed it wouldn't solve Example 1, where the fundamental problem is subtly breaking user code because "not all vectors are containers" and which problem would be unchanged.
4. *Keep the packed-representation `std::vector<bool>` under another name and document that it's not a container, and finish specifying it.* This would solve all problems. It was proposed in Dublin that we might rename `vector<bool>` to something else (e.g., `bitstring`). I think I have a better solution: Move it into a nested namespace within namespace `std`, say `std::packed`. The code in Example 1 would now work for all types, without surprises. This technique also solves Problem #3, because any existing code that might want the packed representation or the `flip()` interface still works as written after writing `using namespace std::packed`.
5. *Deprecate `std::vector<bool>`, and finish specifying it.* This is a "punt," and puts us in the odd position of admitting we made a mistake while at the same time finishing the mistake's specification so that it can be used in the meantime. I don't think this option is viable.
6. *Eliminate `std::vector<bool>`.* This solves all problems, including #3 which is addressed by #5.

I think that only variants on Options #4 and #6 are implementable and otherwise viable.

Proposed Resolutions

I propose one of the following two alternative solutions, either of which solves all of the problems. The first represents the minimum change.

1. Remove 23.2.5 [lib.vector.bool], on the basis that it is broken and breaks user code, and that any existing code that can tell the difference (i.e., uses `flip()` or any other part of `vector<bool>`) is relying on underspecified behavior anyway.
2. Move `vector<bool>` into a nested namespace within namespace std (e.g., `std::packed`), and finish specifying it. Any existing code that might want the packed representation or the `flip()` interface still works as written after writing `using namespace std::packed`.