Toward a concept-enabled standard library

Matt Austern, Gabriel Dos Reis, Eric Niebler, Bjarne Stroustrup,
Herb Sutter, Andrew Sutton, Jeffrey Yasskin
N4263
2014-11-04

Now that the Concepts Lite TS is (almost) done, the next step is a version of the standard library that uses Concepts. Here are some informal notes on what we might want from such a library and what some of the incremental steps are for getting there.

Overall goals:

- We want to define and apply concepts to all parts of the Standard Library where they make sense.
- We want to define a relatively small number of concepts, and we want constraints on individual classes and functions to be simple. We don't want to have algorithms where the constraint clause is longer than the implementation. We want to avoid all of the *has_** concepts that were present in the C++0x concepts attempt.
- Desirable property: it shouldn't be necessary to use any of the traits classes anywhere in a standard library implementation.
- We will not attempt to make the concept-enabled library fully compatible with the library described by the C++14 standard. The existing constraints are very complicated, a fact that becomes more obvious when the constraints are spelled out in code. Some features should be simplified or eliminated as necessary to reduce the number and complexity of concepts. Example: in a pre-concepts world it's natural to allow implicit conversion (in C++98 forbidding them would have required a lot of extra effort) but in a world with concepts allowing them is extra complexity.
- We do want to limit incompatibilities to areas where the damage to existing code is limited and the benefits significant. For example, the Palo Alto design (N3351) provides common_type to handle "well behaved" comparisons (etc.) that would otherwise require implicit conversions.

Delivery vehicle

- A concept-enabled standard library will be delivered as a TS by SG8 in consultation with the appropriate WGs.
- Since the TS will describe an incompatible version of the standard library, it's essentially certain that people will see "std2" as an opportunity to make design changes even if those fixes aren't directly related to concepts. We won't attempt to fight that temptation; trying to limit the scope will probably just lead to a lot of scope arguments. We'll just accept in advance that this TS will be open to a variety of proposals and we'll judge each on its merits.

Initial milestones

- The obvious place to start is the STL, i.e. algorithms, iterators, and containers. That's where people have thought the most about concepts.
- Although we're not attempting to make the concepts-enabled library 100% compatible with the C++14 standard library, we want a concepts-enabled STL that's at least broadly similar to the existing STL.
- We can use N3351, the "Palo Alto TR," as the starting point. N3351 defines concepts for the complete set of STL algorithms. It has been implemented.

Some work required to turn N3351 into a TS

- N3351 covers iterators and algorithms, but not containers. At the very least we need to say what containers' constraints are, and we may also want to define some container concepts.
- Verifying that N3351 completely addresses proxy iterators, e.g. N3851's `array_view`, or `vector<bool>::iterator`. (We may not want to have the `vector<bool>` specialization in a concept-enabled STL, but we probably still don't want to preclude something like it.)
- Verifying that N3351 has sufficient support for rvalue references and movable types.
- Stripping out the appendices, and perhaps making minor changes to take account of changes in Concepts Lite during the standardization process.

Some open technical issues

- Ranges. There are several active proposals for range-based algorithms, including N4128, which has considerable overlap with a concepts-enabled STL. Proposal: decouple these two pieces of work to the extent possible so that progress can be made on both independently. We believe that ranges will never replace iterators: we will layer range-based algorithms on top of iterator-based algorithms, so there is value in defining concept-enabled iterator-based algorithms as done in N3351. Similarly, there's value in defining a version of range-based algorithms that isn't blocked while waiting for concepts to be added to the whole library.
- Allocators. We won't be able to avoid thinking about them; they'll affect concepts as applied to containers, both constraints that containers obey and those that they impose on element types. We want the same thing for containers as we do for algorithms: the concepts shouldn't be too numerous or too complicated. One possibility is to leave allocators out entirely; another is keep them, but purely as run-time polymorphic machinery without support for alternative memory models and without any kind of interaction with containers' template parameter lists or element types.
- Container concepts. Container or container-like concepts may be useful for container-based algorithms, ranges, iterator adaptors, and container adaptors. It may, however, be possible to achieve many of these goals using weaker concepts that don't deal with element ownership; the range proposal in N4128, for example, hasn't needed container concepts yet.