The Boost.Iterator Library Boost

Authors: David Abrahams, Jeremy Siek, Thomas Witt

Contact: dave@boost-consulting.com, jsiek@osl.iu.edu, witt@styleadvisor.com

organizations: Boost Consulting, Indiana University Open Systems Lab, Zephyr Associates, Inc.

date: $Date: 2004/01/27 04:05:33 $

Abstract: The Boost Iterator Library contains two parts. The first is a system of concepts which extend the C++ standard iterator requirements. The second is a framework of components for building iterators based on these extended concepts and includes several useful iterator adaptors. The extended iterator concepts have been carefully designed so that so that old-style iterators can fit in the new concepts and so that new-style iterators will be compatible with old-style algorithms, though algorithms may need to be updated if they want to take full advantage of the new-style iterator capabilities. Several components of this library have been accepted into the C++ standard technical report. The components of the Boost Iterator Library replace the older Boost Iterator Adaptor Library.

Table of Contents

New-Style Iterators

The iterator categories defined in C++98 are extremely limiting because they bind together two orthogonal concepts: traversal and element access. For example, because a random access iterator is required to return a reference (and not a proxy) when dereferenced, it is impossible to capture the capabilities of `vector<bool>::iterator` using the C++98 categories. This is the infamous "`vector<bool>` is not a container, and its iterators aren't random access iterators", debacle about which Herb Sutter wrote two papers for the standards comittee (n1185 and n1211), and a Guru of the Week. New-style iterators go well beyond patching up `vector<bool>`, though: there are lots of other iterators already in use which can't be adequately represented by the existing concepts. For details about the new iterator concepts, see our

- Standard Proposal For New-Style Iterators (PDF)

Iterator Facade and Adaptor

Writing standard-conforming iterators is tricky, but the need comes up often. In order to ease the implementation of new iterators, the Boost.Iterator library provides the `iterator_facade` class template, which implements many useful defaults and compile-time checks designed to help the iterator author ensure that his iterator is correct.

It is also common to define a new iterator that is similar to some underlying iterator or iterator-like type, but that modifies some aspect of the underlying type's behavior. For that purpose, the library supplies the `iterator_adaptor` class template, which is specially designed to take advantage of as much of the underlying type's behavior as possible.

The documentation for these two classes can be found at the following web pages:

- `iterator_facade` (PDF)
- `iterator_adaptor` (PDF)

Both `iterator_facade` and `iterator_adaptor` as well as many of the specialized adaptors mentioned below have been proposed for standardization, and accepted into the first C++ technical report; see our

Standard Proposal For Iterator Facade and Adaptor (PDF)

for more details.

Specialized Adaptors

The iterator library supplies a useful suite of standard-conforming iterator templates based on the Boost iterator facade and adaptor.

- `counting_iterator` (PDF): an iterator over a sequence of consecutive values. Implements a "lazy sequence"
- `filter_iterator` (PDF): an iterator over the subset of elements of some sequence which satisfy a given predicate
- `indirect_iterator` (PDF): an iterator over the objects *pointed-to* by the elements of some sequence.
- `permutation_iterator` (PDF): an iterator over the elements of some random-access sequence, rearranged according to some sequence of integer indices.
- `reverse_iterator` (PDF): an iterator which traverses the elements of some bidirectional sequence in reverse. Corrects many of the shortcomings of C++98's `std::reverse_iterator`.
- `transform_iterator` (PDF): an iterator over elements which are the result of applying some functional transformation to the elements of an underlying sequence. This component also replaces the old `projection_iterator_adaptor`.
- `zip_iterator` (PDF): an iterator over tuples of the elements at corresponding positions of heterogeneous underlying iterators.

Iterator Utilities

Traits

- `pointee.hpp` (PDF): Provides the capability to deduce the referent types of pointers, smart pointers and iterators in generic code. Used in `indirect_iterator`.
- `iterator_traits.hpp` (PDF): Provides MPL-compatible metafunctions which retrieve an iterator's traits. Also corrects for the deficiencies of broken implementations of `std::iterator_traits`.

Testing and Concept Checking

- `iterator_concepts.hpp` (PDF): Concept checking classes for the new iterator concepts.
- `iterator_archetypes.hpp` (PDF): Concept archetype classes for the new iterators concepts.

Upgrading from the old Boost Iterator Adaptor Library

If you have been using the old Boost Iterator Adaptor library to implement iterators, you probably wrote a `Policies` class which captures the core operations of your iterator. In the new library design, you'll move those same core operations into the body of the iterator class itself. If you were writing a family of iterators, you probably wrote a type generator to build the `iterator_adaptor` specialization you needed; in the new library design you don't need a type generator (though may want to keep it around as a compatibility aid for older code) because, due to the use of the Curiously Recurring Template Pattern (CRTP) [Cop95], you can now define the iterator class yourself and acquire functionality through inheritance from `iterator_facade` or `iterator_adaptor`. As a result, you also get much finer control over how your iterator works: you can add additional constructors, or even override the iterator functionality provided by the library.

If you're looking for the old `projection_iterator` component, its functionality has been merged into `transform_iterator`: as long as the function object's `result_type` (or the `Reference` template argument, if explicitly specified) is a true reference type, `transform_iterator` will behave like `projection_iterator` used to.

History

In 2000 Dave Abrahams was writing an iterator for a container of pointers, which would access the pointed-to elements when dereferenced. Naturally, being a library writer, he decided to generalize the idea and the Boost Iterator Adaptor library was born. Dave was inspired by some writings of Andrei Alexandrescu and chose a policy based design (though he probably didn't capture Andrei's idea very well - there was only one policy class for all the iterator's orthogonal properties). Soon Jeremy Siek realized he would need the library and they worked together to produce a "Boostified" version, which was reviewed and accepted into the library. They wrote a paper and made several important revisions of the code.

Eventually, several shortcomings of the older library began to make the need for a rewrite apparent. Dave and Jeremy started working at the Santa Cruz C++ committee meeting in 2002, and had quickly generated a working prototype. At the urging of Mat Marcus, they decided to use the GenVoca/CRTP pattern approach, and moved the policies into the iterator class itself. Thomas Witt expressed interest and became the voice of strict compile-time checking for the project, adding uses of the SFINAE technique to eliminate false converting constructors and operators from the overload set. He also recognized the need for a separate `iterator_facade`, and factored it out of `iterator_adaptor`. Finally, after a near-complete rewrite of the prototype, they came up with the library you see today.

[Cop95] [Coplien, 1995] Coplien, J., Curiously Recurring Template Patterns, C++ Report, February 1995, pp. 24-27.