Index Based Ranges

Document: N3752
Date: 2013-08-30
Authors: Arno Schödl (schoedl@think-cell.com), Fabio Fracassi(f.fracassi@gmx.net)

Authors Note

This document is proposed as a basis for continuing work and discussion. The design choices implied in this work are meant as a starting point. Better ideas, as well as comments and critiques will be gratefully received.

Key Points

In contrast to most currently popular range implementations, which define range iterations in terms of (pairs of) iterators we propose to use range adaptors and indices (which is similar to N1873 - Cursor/Property Maps). The most important consequence of this is that we can avoid "Fat Iterators" when stacking range adaptors, in particular range filters.

Fat Iterators

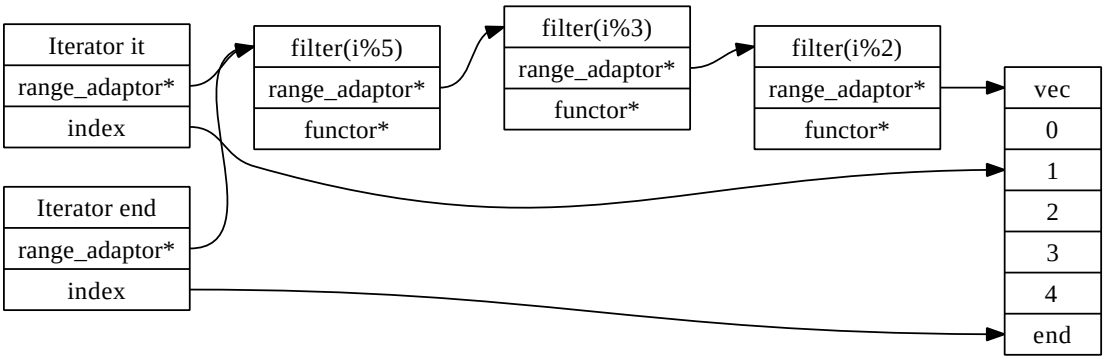
Consider the following (contrived) example which uses a stack of filters:

```
1  std::vector<int> vec = {1,2,3,4,5,6,7,8,9,10};
2  auto sieve = vec | boost::adaptors::filtered([](int i){ return i%2!=0; })
3                   | boost::adaptors::filtered([](int i){ return i%3!=0; })
4                   | boost::adaptors::filtered([](int i){ return i%5!=0; });
5
6  auto it = std::begin(sieve);
7  auto end = std::end(sieve);
```

Now consider what information the iterator `it` does have to carry if it is implemented in terms of the iterators of the underlying range. At the very least each has to store (by value, as references would be far to brittle) the underlying position iterator, the predicate and the underlying end iterator. Of course the underlying iterators in this example are in turn iterators of a filtering range, so the size of the fully filtered iterator grows exponentially with the number of stacked ranges.

Range adaptors and indices

We can prevent that by separating per-range data and per-iterator data into a range adaptor and an index. Consider the above example again, but this time the iterator saves a range adaptor by reference and a special index, which serves to indicate a position in the original range. The index by itself can do no operations. For it to be meaningful it has to be combined with the matching range adaptor. By separating the pure position into the index, and the operations into the range adapter, we are now able to use the *original* ranges iterator (which are usually slim) throughout the adapter stack.



Separating the index is mainly an implementation detail, the ranges we propose are mix and match perfectly iterator based implementations. However this design drives some of the decision about how to handle the details of lifetime management or constness.

Lifetime

Range adaptors usually store a reference to their base range, and thus are susceptible to dangling references if the base range goes out of scope before the adapter does. Range adaptors do however aggregate their base range (i.e. store it by value) iff the base range is an rvalue. This enables us to have a optimal size of range adapters. It is either constant size or grows linearly with the number of adapted ranges.

To implement this we use a `reference_or_value` template which might be of more general interest as a basic building block.

Constness

A range will transitively ”inherit” the constness from its base range or range adaptor. That means that the elements of a `Range const&` can never be modified.

Generator Ranges

This proposal can be extended to support a further range category below the one that provides input iterators. We propose to call ranges that fit into this category generator ranges. A generator range does not provide iterators but simply calls a functor on each of its elements. The functor is passed to the ranges `operator()`.

The generator interface is enough to implement a great number of algorithms (e.g. `for_each`, `all_of`, `any_of`, `none_of`, ...) and is trivially implementable for all input ranges. Furthermore for certain Ranges like join Ranges or tree traversals it can be implemented with less overhead than iterator based categories.

This makes it very easy to create on-the-fly ranges:

```
struct generator_range {
   template< typename Func >
   void operator()( Func func ) {
      for(int i=0;i<50;++i) {
         func(i);
      }
   }
};

for_each( make_filter_range( generator_range(), [](int i){ return i%2==0; } ),
          [](int i)
{
   std::cout << i << ", ";
});
```

To support unlimitied generator ranges we propose that the functor to be passed to the generated range supports a protocol to break out of an iteration. We call this `break_or_continue` and it works like this.

```
struct generator_range_break {
   template< typename Func >
   break_or_continue operator()( Func func ) {
      for(int i=0;i<5000;++i) {
         if (func(i)==break_) { return break_; }
      }
      return continue_;
   }

```

```
10  };
11
12  for_each( make_filter_range( generator_range_break(), [](int i){ return i%2==0; } ),
13            [](int i) -> break_or_continue
14  {
15     std::cout << i << ", ";
16     return (i>=50)? break_ : continue_;
17  });
```