Index Based Ranges (Rev. 1)

Document: N3782 - revises N3752
Date: 2013-09-24
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.

Ranges and Traversables

This paper is intended to be fully compatible with N3763 - Traversable Arguments. We intedend to use the `Traversable` concept as a basis wherever applicable. N3763 sets a good foundation for making Traversables easy to work with. In this paper we would like to explore the design space beyond this basis to make working with them both easy and powerful for programmers.

Adaptors

One of the main advantages of using Traversables over ad-hoc pairs of iterators is the ability to stack several operations on them and be able to evaluate the results lazily. This can be achieved by using range adaptors. The most common operations that are usually provided in this way are transform, filter and sub_range. We think that it is crucial for us to not standardize a library that cannot do at least those operations as efficiently as possible. While it is possible to implement those adaptors with iterator based ranges, or even purely with iterators, those iterators do become what we dubbed "Fat Iterators". The problem is that iterators, because they have no notion of the underlying range or container, do have to carry duplicate information.

Fat Iterators

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

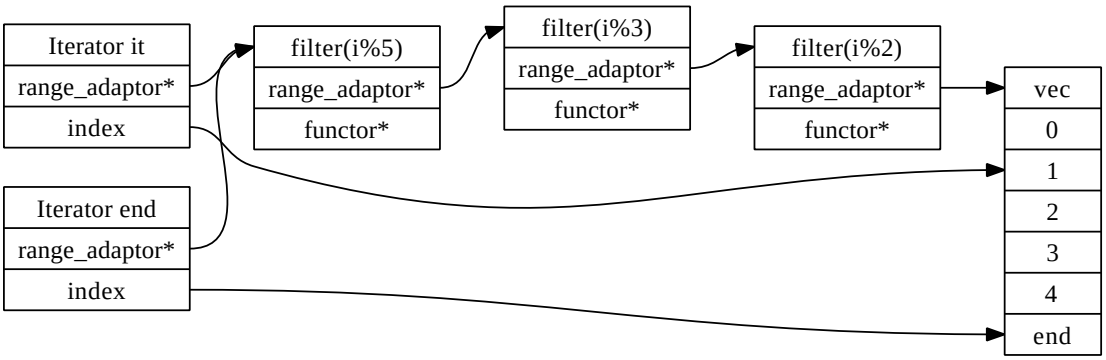
```
std::vector<int> vec = {1,2,3,4,5,6,7,8,9,10};
auto sieve = vec | boost::adaptors::filtered([](int i){ return i%2!=0; })
                 | boost::adaptors::filtered([](int i){ return i%3!=0; })
                 | boost::adaptors::filtered([](int i){ return i%5!=0; });

auto it = std::begin(sieve);
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.



At this point we do not propose to standardize the index protocol, or any public interface. However we do not see any drawbacks on using the index based protocol in an implementation, and we do get optimally small iterators out of doing so. As we do strive to make standard library algorithms to be as fast as hand-rolled loops, which would become more difficult if iterators grow linearly in size, we feel that we should be careful not to make a standardization decisions that makes this kind of optimization impossible to implement.

Design decision that could forbid this optimization might be those 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 in our implementation. 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:

```
1 struct generator_range {
2    template< typename Func >
```

```
3     void operator()( Func func ) {
4        for(int i=0;i<50;++i) {
5           func(i);
6        }
7     }
8  };
9
10 for_each( make_filter_range( generator_range(), [](int i){ return i%2==0; } ),
11           [](int i)
12 {
13    std::cout << i << ", ";
14 });
```

Generator ranges are already quite useful the way they are now, but they could really shine in conjunction with resumable functions (N3722) or coroutines (N3708).

Breaking

One drawback algorithms over Traversables (in contrast to hand-rolled loops) have is that they do not provide a way to break out of the loop. While this is somewhat optional with regard to Traversable Ranges, with generator Ranges, that might potentially be infinite, it becomes a requirement to be able to do so.

We propose that the functor to be passed to the generator ranges or algorithms like `for_each` supports a protocol to break out of an iteration. We call this `break_or_continue` and it works like this.

```
1  struct generator_range_break {
2     template< typename Func >
3     break_or_continue operator()( Func func ) {
4        for(int i=0;i<5000;++i) {
5           if (func(i)==break_) { return break_; }
6        }
7        return continue_;
8     }
9
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  });
```

Implementation Scope

We are currently working on publishing a reference implementation to showcase and test the issues raised here. The implementation is used extensively throughout our sizeable production codebase.

The basis of this implementation needs N3763 style `range_begin` to reliably detect a Traversable, and get the necessary dependent type information.

The first design decision we took was to concentrate on Traversables as a Concept. That means that we do not make any assumptions on whether a Traversable is lightweight or not, or whether or not it does own its values. We have found that in practice this is seldom a problem, and adaptors as well as algorithms can be implemented generically and efficiently without committing on any of those assumptions.

We implement the three basic adapters, `filter_range`, `transform_range` and `sub_range`. Our implementation is careful to work with both all kind of Traversables, and with generator ranges as well. Each of our range adaptors also implements an Iterator interface which falls trivially out of the index based interface. This enables seamless integration with other range based algorithms like range based for, or boost ranges.

We also implement basic algorithms `for_each` and `equal`, with support for the break protocol and generator ranges.

We think that those few features serve as a good building block as well as a test bed to evaluate the consequences of those foundational design decisions. We believe that the first step toward standardized ranges should focus on these decisions.

We have also implemented several more advanced algorithms like quantifiers (contians, all_of, ...) and several partitioning algorithms, and encountered no implementation difficulties that would hint at a unsound basic model.