Leveraging the range-based for loop

Matthäus Brandl

MUC++ on 2018-07-26

Leveraging the range-based for loop

Agenda

1. The range-based for loop
   1. Usage
   2. Ranges
   3. Limitations
2. Boost.Range
   1. Usage of range adaptors
   2. How to overcome the limitations of 1.3
   3. Integer ranges
3. Your own range adaptor
   1. Structure
   2. Implementation
   3. Example

The range-based for loop

The regular for loop

Traversing STL containers with C++03 is done using iterators and the regular for loop.

Example:

```
std::vector<std::string> container{ /* ... */ };
for (std::vector<std::string>::iterator iter = container.begin();
    iter != container.end();
    ++iter)
{
    std::cout << *iter << std::endl;
}
```

Problems with this approach:

- A lot of boilerplate code
- Not so easy to detect how the container is traversed
  - Does the loop really start with the first element and stop with the last one?
  - Hard to detect if the iterator is manipulated in the loop statement (e.g., `iter` might be incremented or decremented conditionally)
- Potential pitfalls
  - Iterator increment may be done with postfix instead of prefix operator
  - End condition is implemented using comparision operator instead of inequality

Similar problems arise with index-based for loops

The range-based for loop

The range-based for loop

Fortunately C++11 brought us the range-based for loop.

Example:

```
std::vector<std::string> container{ /* ... */ };
for (auto && item : container)
{
    std::cout << item << std::endl;
}
```

The range-based for loop is equivalent to a regular for loop, where the loop expressions are set up by the compiler automatically.

This fixes most of the problems of the regular for loop:

- No more boilerplate code, easy to understand what is going on
- No access to the underlying iterator
  Each element of the container is visited exactly once in the order defined by the container
- Less pitfalls
  - How to access the items correctly?
    By value or by reference
    - If by reference, lvalue, rvalue or forwarding?
      - If by lvalue, const or non-const?

The range-based for loop

How to access the items

How to access the items of a range correctly is a question often asked. (e.g. here or here on StackOverflow)

The answer is similar to that of how to capture function arguments, except that the usage of `auto` is preferred in this case:

- Access by value (`for (auto i : r)`) if the size of the type of the items is about the same as a pointer and copying is trivial (e.g., `int` or double) and you do not want to change the items of the range.
  Consider making the copy const (`for (auto const i : r)`) if you do not need to change it.
- Access by const reference (`for (auto const & i : r)`) if the size is considerably larger than a pointer or copying may be expensive (e.g., `std::string`) and you do not want to change the items of the range.
- Access by non-const reference (`for (auto & i : r)`) if you need to change the items of the range.
- It is always possible to use a forwarding reference (`for (auto && i : r)`), which will always compile and always avoid copies, but changing the items will only be possible if the range is non-const.

The range-based for loop

What is a range?

Usually we use containers, but it is the **range-based** for loop, so what is a range exactly (for the C++11 standard)?

A range is a sequence of items, denoted by a begin iterator pointing to the first item and an end iterator pointing past the last element. The iterator type must be at least incrementable, testable for inequality and dereferenceable.

There must exist a way for a given *range-expression* to retrieve the begin and the end iterator.
The following possibilities are tested in the given order, the first match is used as described:

1. If the *range-expression* is of array type (e.g., `int[10]`) the begin iterator is the pointer to `&range[0]` and the end iterator is the pointer to `&range[0] + range_size`
2. If the *range-expression* is of class type with members named `begin` and `end` (e.g. std::vector<int>) the begin iterator is the result of `range.begin()` and the end iterator is the result of `range.end()`.
   Note: this matches regardless of type and accessibility of these members
3. Otherwise the begin iterator is the result of `begin(range)` and the end iterator is the result of `end(range)`.
   Both need to be free functions which can be found by ADL.
   (simplified this means that the functions must be declared in the same namespace as the range type)

Note:

If the expression is a braced-init-list (e.g. `{ 1, 2, 3, 4, 5 }`) then it is deduced as `std::initializer_list<>&&` which has begin and end member functions, thus it matches case 2.

The range-based for loop

Example using case 3

The Boost.Filesystem library offers the class `boost::filesystem::directory_iterator` which allows to traverse all files in a directory. The begin iterator is obtained by constructing an instance with the directory as argument, the end iterator is obtained by default construction:

```
namespace fs = boost::filesystem;
for (auto iter = fs::directory_iterator{ "/path/to/directory" };
    iter != fs::directory_iterator{};
    ++iter)
{
    std::cout << iter->path() << std::endl;
}
```

However Boost also offers the following free functions which accept this iterator type:

- `boost::filesystem::begin` which simply returns the reference to the passed iterator
- `boost::filesystem::end` which always returns a default constructed `directory_iterator`

This allows using the range-based for loop in an equivalent manner:

```
for (auto const & entry : fs::directory_iterator{ "/path/to/directory" })
{
    std::cout << entry.path() << std::endl;
}
```

The range-based for loop

Limitations of the range-based for loop

Now you are excited about the range-based for loop and its simplicity and want to use it everywhere.
But what if

- you need to iterate your range in reversed order
- you need to iterate only a slice of the given range
- you need to iterate two (or more) ranges synchronously
- you need to iterate two ranges as if it were one long range
- you need also need the index, e.g. for output or logging
- you only need to visit every nth element
- your ranges contains consecutive duplicates which you do not want to visit (compare with `std::unique()`)
- you need a large range of integers for which you do not want to allocate memory

Granted, there exist work arounds for many of these problems, but we lose some of the guarantees.
Often we could manually keep track of the index, but then the index might get out of sync.

Solution: Boost.Range for the win!

Boost.Range

Range Adaptors

Boost.Range offers several range adaptors which give a different view of the adapted range. Note that the adapted range is not altered by the adaption. Furthermore the adapting ranges are lazy, the values of their elements are generated on demand.

To achieve this Boost.Range defines iterators which work on the adapted range:

- the adapting iterator might change the iterator type
  Example: it might use a `std::reverse_iterator` internally
- `operator++()` may forward the increment request to the adapted iterator in a different way
  Example: increment twice instead of once
- `operator*()` may change the value it retrieved from dereferencing the adapted iterator
  Example: conditionally replace the value with another one

Because the adaptor syntax cannot be used when dealing with several ranges Boost.Range also provides helper functions for these cases.

Boost.Range

How to use Boost's Range Adaptors

Each range adaptor has its own header, but there is also `<boost/range/adaptors.hpp>` to include all of them.

For each adaptor there exist two forms of usage:

- The constructor form: `boost::adaptors::adapt(range, arguments…)`
- The pipe form: `range | boost::adaptors::adapted`
  or `range | boost::adaptors::adapted(arguments…)`

The pipe form is the preferred one as it allows for easier reading if several adaptors are concatenated.

Not let's see some of them in action!

Boost.Range

Iterate in reverse order

The adaptor `reversed` (or `reverse()`) allows to traverse a range in reverse order.

Example:

```
#include <boost/range/adaptor/reversed.hpp>
int const range[5] = { 0, 1, 2, 3, 4 };
for (auto const entry : range | boost::adaptors::reversed)
{
    std::cout << entry << ", ";
}
```

Output:

```
4, 3, 2, 1, 0,
```

Link to Boost Documentation

Boost.Range

Iterate over a slice of a range

The adaptor `sliced` (or `slice()`) allows to traverse a part of the range. This subrange is denoted by the start and end index.

But beware, iterators form an half-open range and so do indices in the context of Boost.Range.
The end index actually is the “past-the-end index”.

Example:

```
#include <boost/range/adaptor/sliced.hpp>

int range[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
for (auto const entry : range | boost::adaptors::sliced(3u, 6u))
{
    std::cout << entry << ", ";
}
```

Output:

```
3, 4, 5,
```

Link to Boost Documentation

Boost.Range

Iterate several ranges at the same time

The helper function `combine()` allows to combine several ranges into one range so that they can be traversed in parallel.

Note that all ranges need to have equal size due to the implementation of the resulting iterator’s inequality operator.

Example:

```cpp
#include <boost/range/combine.hpp>

int const ints[4] = { 0, 1, 2, 3 };
auto const doubles = { 0.0, 0.1, 0.2, 0.3 };
auto const strings = std::vector<std::string>{ "zero", "one", "two", "three" };

std::cout << std::fixed << std::setprecision(1);

for (auto const & entry : boost::range::combine(ints, doubles, strings))
{
    std::cout << (boost::get<0>(entry)) << " | "
              << (boost::get<1>(entry)) << " | "
              << (boost::get<2>(entry)) << ", ";
}
```

Output:

```
0 | 0.0 | zero, 1 | 0.1 | one, 2 | 0.2 | two, 3 | 0.3 | three
```

Link to Boost Documentation

Boost.Range

Iterate two ranges consecutively

The helper function `join()` allows to concatenate two ranges of same item types into one range so that they can be traversed one after the other.

Obviously the values in both range must have the same type. The only exception are classes with the same base class, which can be mixed but should be accessed by reference to avoid slicing. If the types are convertible (e.g. `int` and `double`) your code will compile but an internal function will return a reference to a temporary.

Note that the joined range incurs a performance cost due to the need to check if the end of a range has been reached internally during traversal.

Example:

```
#include <boost/range/join.hpp>

int const range1[3] = { 0, 1, 2 };
auto const range2 = { 3, 4, 5, 6 };

for (auto const entry : boost::range::join(range1, range2))
{
    std::cout << entry << ", ";
}
```

Output:

```
0, 1, 2, 3, 4, 5, 6,
```

Link to Boost Documentation

Boost.Range

Iterate a range and get the current index

The adaptor `indexed` (or `index()`) allows to traverse a range and at the same time dispose of the current index. The start index is expected as argument, by default the start index is 0.

Example:

```
#include <boost/range/adaptor/indexed.hpp>
int const range[4] = { 0, 1, 2, 3 };
for (auto const & entry : range | boost::adaptors::indexed{ 100 })
{
    std::cout << entry.index() << ": " << entry.value() << "\n";
}
```

Output:

```
100: 0
101: 1
102: 2
103: 3
```

Link to Boost Documentation

Boost.Range

Iterate every nth item of a range

The adaptor `strided` (or `stride()`) allows to traverse a range and only visit every nth item. The difference between two items is expected as argument.

Example:

```
#include <boost/range/adaptor/strided.hpp>
int const range[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
for (auto const entry : range | boost::adaptors::strided( 3u ))
{
    std::cout << entry << ", ";
}
```

Output:

```
0, 3, 6, 9,
```

Link to Boost Documentation

Boost.Range

Iterate only over unique items

The adaptor `uniqued` (or `unique()`) allows to traverse a range and only visit consecutively unique items.
With this adaptor it is possible to work with const ranges which cannot be done with std::unique.

Example:

```
#include <boost/range/adaptor/uniqued.hpp>
int const range[10] = { 0, 1, 1, 2, 1, 3, 3, 3, 3, 4 };
for (auto const entry : range | boost::adaptors::uniqued)
{
   std::cout << entry << ", ";
}
```

Output:

```
0, 1, 2, 1, 3, 4,
```

Link to Boost Documentation

Boost.Range

Iterate over a large range of integers

If one needs a large range of consecutive integers the first idea might by to use a vector and store these integers in it. However this will allocate memory, something which is not always desirable or feasible. With Boost's `irange` one can generate an integer range without allocating memory.

The arguments are the start index and past the end index as usual.

Note however that in performance critical situations a less-safe regular loop is preferable.

Example:

```
#include <boost/range/irange.hpp>

auto const range = boost::irange(0, 100);
auto const result = std::accumulate(range.begin(), range.end(), 0);
std::cout << "Sum: " << result << std::endl;
```

Output:

```
Sum: 4950
```

Link to Boost Documentation

Your own range adapter

When Boost is no help

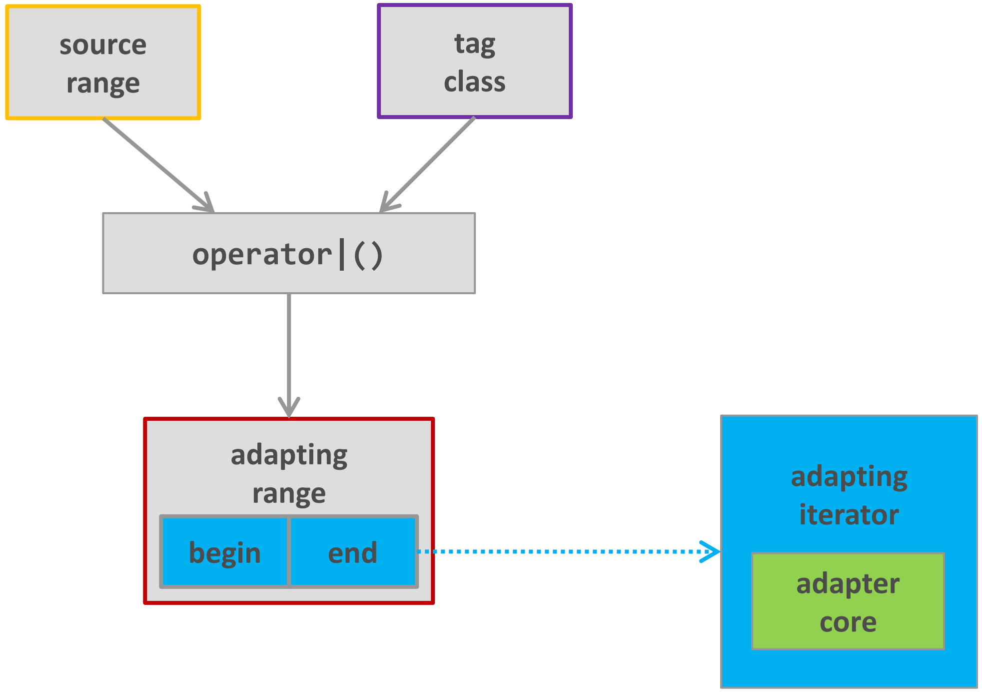
In some domains you often need to traverse a range in adjacent pairs:

- For example to calculate the length of a polygonal line you need to sum the lengths of the single legs.
- To get the length of a leg you need the distance from start to end point.
- The current end point will be the start point of the next leg.

Hence we want a range adapter which traverses a range as adjacent pairs.
The range { 0, 1, 2, 3, 4 } shall become { (0, 1), (1, 2), (2, 3), (3, 4) }.

Your own range adapter

Range adapter structure



Let us implement these parts…

Your own range adapter

The adapter core

The adapter core is a (mostly anonymous) type which makes for easy usage inside the for loop body.
It provides member functions to access the adapted value in an easy way.

adapter core

Remember the `indexed`-adaptor, there the adapter core provided the two member functions `value()` and `index()`.

But there is more to the adapter core. Your adapting iterator will need to return a reference or a pointer to the core. To be able to do so it must “live” inside of the iterator. This can be achieved via inheritance or via composition. Although we want to favor composition this is not always the best thing to do.

Note that the adapter core is not necessary in all cases. When your adapter only filters the source range you can pass on the reference or the pointer which you received from the adapted iterator.

Your own range adapter

The adapter core

adapter core

```
template <typename Iter>
class adjacent_pair
{
    using dereferenced_type = typename std::iterator_traits<Iter>::reference;

public:
    dereferenced_type forward() const
    {
        return *m_forward;
    }

    dereferenced_type backward() const
    {
        return *m_backward;
    }

protected:

    adjacent_pair() = default;

    explicit adjacent_pair(Iter const frontward, Iter const backward)
        : m_forward(frontward)
        , m_backward(backward)
    {}

    Iter m_forward;     // allusion to front()
    Iter m_backward;    // allusion to back()

};
```

Your own range adapter

The adapting iterator

The adapting iterator implements the minimal STL iterator interface that is needed for the range-based for loop:

- Comparison for inequality
- Prefix increment
- Dereference

Furthermore we should add

- Default constructor
- Typedefs for `std::iterator_traits<>`

Copy constructor and copy assignment operator get added by default

Note that this still does not satisfy any of the STL's iterator concepts, so STL algorithms might not compile when used with this iterator.

Your own range adapter

The adapting iterator

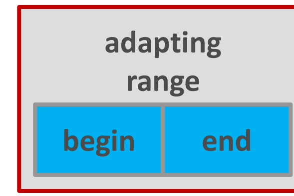
```cpp
template <typename Iter>
class adjacent_iterator : private adjacent_pair<Iter>
{
public:
    using iterator_category = std::forward_iterator_tag;
    using value_type = std::add_const_t<adjacent_pair<Iter>>;
    using difference_type = typename std::iterator_traits<Iter>::difference_type;
    using pointer = std::add_pointer_t<value_type>;
    using reference = std::add_lvalue_reference_t<value_type>;

    adjacent_iterator() = default;
    adjacent_iterator(Iter const forward, Iter const backward)
        : adjacent_pair<Iter>(forward, backward)
    {}

    adjacent_iterator & operator++()
    {
        this->m_forward = this->m_backward;
        ++(this->m_backward);
        return *this;
    }

    reference operator*() const
    {
        return *this;
    }

    friend bool operator!=(adjacent_iterator const lhs, adjacent_iterator const rhs)
    {
        return lhs.m_backward != rhs.m_backward;
    }
};
```

adapting iterator

Your own range adapter

The adapting range

To achieve a valid range we need a class with `begin()` and `end()` member functions along with several typedefs. Luckily Boost already includes the class <u>`iterator_range`</u> which is exactly what we need:




```
#include <boost/range/iterator_range.hpp>
```

```
template <typename Iter>
using adjacent_range = boost::iterator_range<adjacent_iterator<Iter>>;
```

We also need a tag class to select the correct overload of `operator|()`:


```
struct adjacent_tag {};
```

```
static adjacent_tag adjacent;
```

Note that the tag class must be defined in the same namespace as `operator|()`, otherwise ADL will not work.

Your own range adapter

The range factory

operator|()

Finally we need to implement operator|(), but one small piece is still missing before we are able to do that:

```
template <typename Range, typename Iter>
adjacent_range<Iter> operator|(Range & range, adjacent_tag)
{ /* ... */ }
```

Written like this Iter will not be deduced but must be given. We want Iter to be deduced from Range to improve usability. So let us write a small helper trait class:

```
namespace detail
{
    using std::begin;

    template <typename Range>
    struct iterator_type
    {
        using type = decltype(begin(std::declval<Range>()));
    };

    template <typename Range>
    using const_iterator_type_t =
        typename iterator_type<typename std::add_const<Range>::type>::type;
}
```

Now we got everything in place to finish operator|().

Your own range adapter

The range factory

operator|()

```
template <typename Iter>
adjacent_range<Iter> make_adjacent_range(
    Iter const begin, Iter const next, Iter const end)
{
    return adjacent_range<Iter>(adjacent_iterator<Iter>(begin, next),
        adjacent_iterator<Iter>(end, end));
}

template <typename Range>
adjacent_range<detail::const_iterator_type_t<Range>> operator|(Range const & range, adjacent_tag)
{
    using std::begin; using std::end;

    auto const range_begin = begin(range);
    auto const range_end = end(range);

    if (range_begin == range_end)
    {    // empty range
        return detail::make_adjacent_range(range_end, range_end, range_end);
    }

    auto const range_next = std::next(range_begin);

    if (range_next == range_end)
    {    // single element range
        return detail::make_adjacent_range(range_end, range_end, range_end);
    }

    return detail::make_adjacent_range(range_begin, range_next, range_end);
}
```

Your own range adapter

Demonstrating the result

We move everything into the namespace `diy` and write a small test program:

```
int const range[5] = { 0, 1, 2, 3, 4 };

for (auto const & item : range | diy::adjacent)
{
    std::cout << "(" << item.forward() << ", " << item.backward() << "), ";
}
```

Result:

```
(0, 1), (1, 2), (2, 3), (3, 4),
```

I tested this with MSVC v120, v141, gcc 8.1 in C++11 mode and clang 5.0 in c++11 mode.
See demos online on Coliru for gcc and clang.

Further links

- Documentation of Boost.Range 2.0
- Eric Niebler‘s experimental range-v3 library which takes these ideas to the next level
  - Eric Niebler‘s talk at CppCon 2015 showing an application of his library
- Stephan T. Lavavej‘s answer on the problem of traversing adjacent pairs (STL is the std lib maintainer of Microsoft)

```
for (auto & person : audience)
{
    person.thankYou();
}
```