Making Iterators, Views and Containers Easier to Write with Boost.STLInterfaces

Zach Laine

Iterators are essential for getting the STL containers to interoperate with each other and the standard algorithms.

It often comes up that you want to write an iterator for a type you've written, so your type can enjoy the same degree of interoperability.

But that's hard.

Boost.STLInterfaces automates writing much of the API for an iterator, making it easy instead of hard.

```
struct repeated_chars_iterator
{
    using value_type = char;
    using difference_type = std::ptrdiff_t;
    using pointer = char const *;
    using reference = char const;
    using iterator_category = std::random_access_iterator_tag;

    constexpr repeated_chars_iterator() noexcept :
        first_(nullptr),
        size_(0),
        n_(0)
    {}
    constexpr repeated_chars_iterator(
        char const * first,
        difference_type size,
        difference_type n) noexcept :
        first_(first),
        size_(size),
        n_(n)
    {}

    constexpr reference operator*() const noexcept
    {
```

```
struct repeated_chars_iterator : boost::stl_interfaces::iterator_interface<
                                     repeated_chars_iterator,
                                     std::random_access_iterator_tag,
                                     char,
                                     char>
{
    constexpr repeated_chars_iterator() :
        first_(nullptr), size_(0), n_(0)
    {}
    constexpr repeated_chars_iterator(
        char const * first, difference_type size, difference_type n) :
        first_(first), size_(size), n_(n)
    {}

    constexpr char operator*() const { return first_[n_ % size_]; }
    constexpr repeated_chars_iterator & operator+=(std::ptrdiff_t i)
    {
        n_ += i;
        return *this;
    }
    constexpr auto operator-(repeated_chars_iterator other) const
    {
        return n_ - other.n_;
    }
```

When I started using `iterator_interface` to write iterators, I accidentally fixed latent bugs in the hand-written iterators I replaced.

Boost.STLInterfaces handles creation of views and sequence containers, in addition to just iterators. Let's talk first about views.

`view_interface`

C++20 introduced a new template, `std::ranges::view_interface`. When you write a type that you want to use as a view, you give it `begin()` and `end()`, and then inherit from `view_interface`.

Quick reminder -- the C++20 ranges algorithms allow you to use a *sentinel* for the end. So instead of having iterator pairs, you may have an iterator-sentinel pair.

```
struct null_sentinel_t {};
bool operator==(char const * ptr, null_sentinel_t) { return !*ptr; }

// NOT std::distance()!
assert(std::ranges::distance("null-terminated", null_sentinel_t{}) == 15);
```

```
template<class D>
  requires is_class_v<D> && same_as<D, remove_cv_t<D>>
class view_interface : public view_base {
private:
  constexpr D& derived() noexcept {                  // exposition only
    return static_cast<D&>(*this);
  }
  constexpr const D& derived() const noexcept {      // exposition only
    return static_cast<const D&>(*this);
  }
public:
  constexpr bool empty() requires forward_range<D> {
    return ranges::begin(derived()) == ranges::end(derived());
  }
  constexpr bool empty() const requires forward_range<const D> {
    return ranges::begin(derived()) == ranges::end(derived());
  }

  constexpr explicit operator bool()
    requires requires { ranges::empty(derived()); } {
      return !ranges::empty(derived());
    }
  constexpr explicit operator bool() const
    requires requires { ranges::empty(derived()); } {
```

```
struct null_sentinel_t {};
bool operator==(char const * ptr, null_sentinel_t) { return !*ptr; }

struct my_view : std::ranges::view_interface<my_view>
{
    my_view() = default;
    my_view(char const * ptr) : ptr_(ptr) {}

    char const * begin() const { return ptr_; }
    null_sentinel_t end() const { return null_sentinel_t{}; }

private:
    char const * ptr_;
};

assert(std::ranges::distance(my_view("bar")) == 3);
```

This also works, even though we never wrote members `front()` and `operator[]()`.

```
assert(my_view("bar").front() == 'b');
assert(my_view("bar")[1] == 'a');
```

Boost.STLInterfaces provides a version of `view_interface` that works via SFINAE in C++14 and C++17.

Iterators

Custom iterators you write yourself have many uses. You may want to write checking iterators that only do checks in debug builds; you may want to add iterators to legacy containers; and you may want to have an iterator just for one specific use, *ad hoc*.

Let's take a look at an *ad hoc* case.

If you have two sequences, and you want to do one or more operations over all the elements of both sequences, you have to write each operation by hand:

```
using vec_iter = std::vector<int>::const_iterator;
using list_iter = std::list<int>::const_iterator;

boost::variant<vec_iter, list_iter>
find(vec_iter first1, vec_iter last1, list_iter first2, list_iter last2, int i)
{
    // Yuck.
}
```

An alternative is to make an iterator that can handle the iteration, and then use a standard algorithm. Eric Niebler's range-v3 library has a view called a `concat_view` that does this nicely:

```
std::vector<int> vec = /* ... */;
std::list<int> list = /* ... */;
auto const it = std::ranges::find(::ranges::concat(vec, list), 42);
```

Sometimes you cannot use `ranges::concat()`, because the types in the two sequences have no common reference type, even though you know how to project the two types into a common value.

```
struct my_int
{
    my_int() = default;
    explicit my_int(int v) : value_(v) {}
    int value_;
};
static_assert(!std::common_reference_with<my_int, int>);

int as_int(int i) { return i; }
int as_int(my_int mi) { return mi.value_; }
```

For this example of just finding a value, we could just look in the first sequence, then in the second. For more complicated algorithms, that may not be possible. What if a step in the algorithm may use multiple values, and what if one such set of values uses part of each sequence?

```
template<typename T, typename R1, typename R2>
struct concat_iter : boost::stl_interfaces::proxy_iterator_interface<
                         concat_iter<T, R1, R2>,
                         std::bidirectional_iterator_tag,
                         T>
{
    using first_iterator = decltype(std::declval<R1 &>().begin());
    using second_iterator = decltype(std::declval<R2 &>().begin());
    struct end_tag{};

    concat_iter() : in_r1_(false) {}
    concat_iter(
        first_iterator it,
        first_iterator r1_last,
        second_iterator r2_first) :
        r1_last_(r1_last),
        it1_(it),
        r2_first_(r2_first),
        it2_(r2_first),
        in_r1_(true)
    {
        if (it1_ == r1_last_)
            in_r1_ = false;
    }
```

```
template<typename T, typename R1, typename R2>
struct concat_view
    : boost::stl_interfaces::view_interface<concat_view<T, R1, R2>>
{
    using iterator = concat_iter<T, R1, R2>;

    concat_view(iterator first, iterator last) :
        first_(first), last_(last)
    {}

    iterator begin() const noexcept { return first_; }
    iterator end() const noexcept { return last_; }

private:
    iterator first_;
    iterator last_;
};

template<typename T, typename R1, typename R2>
auto concat(R1 & r1, R2 & r2)
{
    using iterator = concat_iter<T, R1, R2>;
    return concat_view<T, R1, R2>(
        iterator(r1.begin(), r1.end(), r2.begin()),
```

```
std::vector<int> vec = { 0, 58, 48 };
std::list<int> list = { 80, 39 };
auto const v = concat<int>(vec, list);
assert(std::ranges::find(v, 42) == v.end());
assert(std::ranges::find(v, 58) == std::next(v.begin()));
```

For bonus points, I could have given `concat()` a projection, so that it would work for the `my_int` case.

Proxy Iterators

Sometimes, the `reference` type of an iterator cannot be a language reference.

For instance, `std::ranges::iota_view` produces values for which there is no underlying storage, so it cannot produce references.

For another example, an iterator that changes from UTF-8 encoding to UTF-32 encoding produces values for which there is no underlying storage.

Boost.STLInterfaces supports proxy iterator types directly.

```
struct zip_iter : boost::stl_interfaces::proxy_iterator_interface<
                      zip_iter,
                      std::random_access_iterator_tag,
                      std::tuple<int, int>,
                      std::tuple<int &, int &>>
{
    zip_iter() : it1_(nullptr), it2_(nullptr) {}
    zip_iter(int * it1, int * it2) : it1_(it1), it2_(it2) {}

    std::tuple<int &, int &> operator*() const
    {
        return std::tuple<int &, int &>{*it1_, *it2_};
    }
    zip_iter & operator+=(std::ptrdiff_t i)
    {
        it1_ += i;
        it2_ += i;
        return *this;
    }
    friend std::ptrdiff_t operator-(zip_iter lhs, zip_iter rhs) noexcept
    {
        return lhs.it1_ - rhs.it1_;
    }
```

`zip_iter`'s reference type is `std::tuple<int &, int &>`, which is not itself a reference (though it contains references).

This is fine by the C++20 iterator concepts; `zip_iter` models `std::random_access_iterator`. In pre-C++20, `zip_iter` is technically an input iterator, since it's `reference` is not a reference type.

It is important to note that even though `zip_iter` is technically an input iterator, it can be used as a random access iterator with the standard algorithms, because it has all the right operations, and they do no concept checks.

You can even make it work with `std::sort()` if you specialize `std::iter_swap()`.

Sequence containers

There is some interest in WG21 in standardizing `static_vector` and `small_vector`.

`static_vector` and `small_vector` will have the same API as `vector`, except that `static_vector` is not allocator-aware.

```cpp
template<class T, class Allocator = allocator<T>>
class vector {
public:
  // types
  using value_type             = T;
  using allocator_type         = Allocator;
  using pointer                = typename allocator_traits<Allocator>::pointer;
  using const_pointer          = typename allocator_traits<Allocator>::const_poir
  using reference              = value_type&;
  using const_reference        = const value_type&;
  using size_type              = implementation-defined; // see [container.requir
  using difference_type        = implementation-defined; // see [container.requir
  using iterator               = implementation-defined; // see [container.requir
  using const_iterator         = implementation-defined; // see [container.requir
  using reverse_iterator       = std::reverse_iterator<iterator>;
  using const_reverse_iterator = std::reverse_iterator<const_iterator>;

  // [vector.cons], construct/copy/destroy
  constexpr vector() noexcept(noexcept(Allocator())) : vector(Allocator()) { }
  constexpr explicit vector(const Allocator&) noexcept;
  constexpr explicit vector(size_type n, const Allocator& = Allocator());
  constexpr vector(size_type n, const T& value, const Allocator& = Allocator());
  template<class InputIterator>
    constexpr vector(InputIterator first, InputIterator last, const Allocator& =
```

There's a lot of redundancy there. For instance, consider the range `vector::assign()` overload, and its typical implementation:

```
template<typename InputIterator>
constexpr void assign(InputIterator first, InputIterator last)
{
    auto out = derived().begin();
    auto const out_last = derived().end();
    for (; out != out_last && first != last; ++first, ++out) {
        *out = *first;
    }
    if (out != out_last)
        derived().erase(out, out_last);
    if (first != last)
        derived().insert(derived().end(), first, last);
}
```

It turns out that the sequence container requirements in C++ require a `erase()` and `insert()` overloads with the right semantics. So the code on the previous slide works for any sequence container.

`assign()` could have been a container-based algorithm like `std::erase()`, instead of a member function.

Boost.STLInterfaces has a `sequence_container_interface` template, analogous to `std::ranges::view_interface`, that helps reduce the code you have to write to implement a `std::vector`-like type.

```cpp
template<typename T, std::size_t N>
struct static_vector :
    boost::stl_interfaces::sequence_container_interface<
        static_vector<T, N>,
        boost::stl_interfaces::element_layout::contiguous>
{
    using value_type = T;
    using pointer = T *;
    using const_pointer = T const *;
    using reference = value_type &;
    using const_reference = value_type const &;
    using size_type = std::size_t;
    using difference_type = std::ptrdiff_t;
    using iterator = T *;
    using const_iterator = T const *;
    using reverse_iterator =
        boost::stl_interfaces::reverse_iterator<iterator>;
    using const_reverse_iterator =
        boost::stl_interfaces::reverse_iterator<const_iterator>;

    static_vector() noexcept;
    explicit static_vector(size_type n);
    explicit static_vector(size_type n, T const & x);
    template<typename InputIterator, typename Enable = /* ... */>
```

A lot of that is just `using` declarations. In total, you need to implement 22 functions (many of them constructors or special member functions), and `sequence_container_interface` provides the other 39.

Reducing the implementation burden from 61 functions to 22 makes writing a sequence container much more palatable. Not having to test the other 39 members is a vital part of that.

The presence of `emplace_back()` is noteworthy. If it is present, it is a hint to the `sequence_container_interface` template that `push_back()` and `pop_back()` have efficient implementation, and so should be part of the generated interface. `emplace_front()` serves the analogous purpose for mutation at the beginning.

If any of the generated operations is more efficiently implementatble directly, you can just add it to your derived type. Its existence will hide the member functions with the same name inherited from `sequence_container_interface`.

Use cases for container-like types are pretty common. Now that you can write full sequence containers with about the same level of effort as you use to spend writing more *ad hoc* container-like types, you should do so.

Doing so has many of the same interoperability benefits that writing conforming iterators does.

Writing custom sequence containers is so much easier now that I recommend not writing allocator support into them. If you want a different allocation strategy, just write a different container.

Allocators are notoriously easy to make subtle mistakes with, and it is impossible to write a universally efficient `insert()` that works for very different allocation strategies.

Consider what `insert()` would look like for `static_vector`, `small_vector`, and `std::vector`, respectively. If you implement these three types with a common template that uses different allocators, you cannot vary the implementation of `insert()`.

Future Work

Concepts

C++20 concept constraints are implemented already, though they are not available in the current Boost release (1.74). The next release or two should find this work integrated.

The library contains two namespaces, `boost::stl_interfaces::v1` and `boost::stl_interfaces::v2`; the former is for pre-C++20 code (using SFINAE), and the latter is for C++20 and later (using concepts).

When you build in pre-C++20 mode, `v1` is inlined; when you build in C++20-and-later mode, `v2` is inlined.

The result is that you can just use `boost::stl_interfaces::foo` -- without naming `v1` or `v2` -- and you get the SFINAE-constrained or the concept-constrained version, as appropriate, without having to change your code.

Other Container Types

I'd really like to add support for associative containers (e.g. `std::map`, `std::set`).

The problem is that most interesting containers these days are not node-based, and the node-related API is part of the associative container requirements. Maybe one day those will get teased apart, or maybe I'll just make a template that ignores that API. It depends on user interest, really.

Questions?

https://github.com/tzlaine/stl_interfaces