Better support for `constexpr` in `std::array`

Document #:	P0107R0
Date:	2015-09-23
Project:	Programming Language C++ Library Evolution Group
Reply-to:	Louis Dionne <ldionne.2@gmail.com>

1 Introduction

[N3598] removed the implicit marking of `constexpr` member functions as `const`. However, the member functions of `std::array` were not revisited after this change, leading to a surprising lack of support for `constexpr` in `std::array`'s interface. This paper fixes this omission by adding `constexpr` to the member functions of `std::array` that can support it with a minimal amount of work.

2 Motivation and Scope

With the advent of generalized constant expressions, it can be useful to create a `std::array` inside a `constexpr` function, and then modify it. Without making some member functions `constexpr`, this is impossible or overly difficult. For example, the following does not compile:

```
template <std::size_t N, std::size_t Size>
constexpr std::array<std::size_t, N * Size>
    cycle_indices(std::array<std::size_t, Size> a)
{
    std::array<std::size_t, N * Size> result{};
    for (std::size_t i = 0; i < N * Size; ++i) {
        result[i] = i % Size;
    }
    return result;
}
```

Adding `constexpr` support for most of `std::array`'s member functions would be trivial and would make such code compile. However, this paper does not propose systematically adding the `constexpr` keyword to standard library types that could support it, and it does not even add `constexpr` to all of `std::array`'s member functions. Even though the author thinks that it should eventually be done, the scope of this proposal is purposefully kept minimal.

3 Impact on the Standard

This proposal is a pure library extension. It does not require any new language features, and it merely adds consistency to `std::array`'s interface.

4 Proposed Wording

Add to `<array>` synopsis of [N4296]:

```
template <class T, size_t N>
constexpr bool operator==(const array<T,N>& x, const array<T,N>& y);

template <class T, size_t N>
constexpr bool operator!=(const array<T,N>& x, const array<T,N>& y);

template <class T, size_t N>
constexpr bool operator<(const array<T,N>& x, const array<T,N>& y);

template <class T, size_t N>
constexpr bool operator>(const array<T,N>& x, const array<T,N>& y);

template <class T, size_t N>
constexpr bool operator<=(const array<T,N>& x, const array<T,N>& y);

template <class T, size_t N>
constexpr bool operator>=(const array<T,N>& x, const array<T,N>& y);
```

Add to *23.3.2.1 class template array overview* of [N4296]:

```
// iterators:
constexpr iterator                begin() noexcept;
constexpr const_iterator          begin() const noexcept;
constexpr iterator                end() noexcept;
constexpr const_iterator          end() const noexcept;
constexpr const_iterator          cbegin() const noexcept;
constexpr const_iterator          cend() const noexcept;

// element access:
constexpr reference        operator[](size_type n);
constexpr reference        at(size_type n);
constexpr reference        front();
constexpr reference        back();
constexpr T*               data() noexcept;
constexpr const T *        data() const noexcept;
```

Add to *23.3.2.5 array::data* of [N4296]:

```
constexpr T* data() noexcept;
constexpr const T* data() const noexcept;
```

5 Discussion

One might observe that some member and non-member functions were not made `constexpr` by this paper.

- The `rbegin`, `rend`, `crbegin`, and `crend` member functions are not made `constexpr`. The reason is that these functions return `reverse_iterator`s, which are not literal types. While we could have decided to go for it and make `reverse_iterator` a literal type, this is left to another proposal in order to leave this proposal small and uncontroversial. While leaving these functions non-`constexpr` leaves some inconsistency in `std::array`'s interface, this inconsistency is precedented by the overloads of `rbegin`, `rend`, `crbegin`, and `crend` for builtin array types, which are not `constexpr` for the same reason.
- The `fill` member function is not made `constexpr` by this paper. The reason is that `fill` can be implemented in terms of `memset` for some types. Since `memset` is not `constexpr`, requiring `fill` to be `constexpr` would force it to be implemented using an explicit loop all the time. Such a pessimization is deemed unacceptable. Overcoming this limitation would most likely require the ability to overload on `constexpr`, which is out of scope of this paper.
- The `swap` member function and the overload of the `swap` free function for `std::array` is not made `constexpr` by this paper. The reason is that the `swap` function is not required to be `constexpr` for other types, which means that `std::array`'s `swap` can't be `constexpr` in the general case. To keep this proposal self-contained and minimal, this inconsistency could be tackled by a different paper adding general support for `constexpr` in `std::swap`. Another possibility would be to amend this paper and make `swap` `constexpr` for `std::array` whenever it can be, i.e. whenever the elements of the array are `constexpr` swappable.

6 Implementation Experience

This proposal was implemented and tested in libc++, and it seems to work just fine.

7 Acknowledgements

Thanks to Marshall Clow for providing comments, and to David Sankel for providing comments and accepting to champion the paper in Kona.

8 References

[N3598] Richard Smith, *constexpr member functions and implicit const*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3598.html

[N4296] Richard Smith, *Working Draft, Standard for Programming Language C++*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf