Document number: D0635R0
Date: 2017-04-23
Author: Dan Raviv <dan.raviv@gmail.com>
Audience: LEWG => LWG

Add `c_array()` member function to `std::array`

I. Introduction

Given an object `ar` of type `std::array<T,N>` and a function `foo` which accepts a reference or pointer to `T[N]`, e.g.,

```
void foo(T(&)[N]);
```
or
```
void foo(T(*)[N]);
```

there is currently no good way to pass the C-array wrapped by `ar` to `foo`, simply because there is no standard way within C++'s safe subset to get a reference to the C-array wrapped by `ar`.

This paper proposes adding to `std::array` a member function which returns a reference to the wrapped C-array in order to make the class compatible with such functions.

II. Motivation and Scope

To pass the C-array wrapped by `ar` to `foo`, a programmer must resort to one of the following:

1) Rewrite `foo` to accept a `std::array<T,N>` parameter instead of a reference (or pointer) to `T[N]`. This isn't always possible (`foo` might be provided by a 3rd-party), isn't always desirable (`foo` might be used in projects which build under a pre-C++11 compiler or, in the case of a pointer to `T[N]` parameter, might be implemented in C), and always takes some programmer time.

2) Add an overload to `foo` which accepts a `std::array<T,N>` instead of a reference (or pointer) to `T[N]`. This isn't always possible (`foo` might be provided by a 3rd-party, or implemented in C), usually isn't desirable (the extra overload is boilerplate for the same implementation), and always takes some programmer time. Adding the overload also needs to be done repeatedly for each function similar to `foo`.

3) Rewrite `ar` so it is of type `T[N]` instead of type `std::array<T,N>`. This isn't always possible and usually isn't desirable.

4) Use a safe but non-standard way to get the C-array wrapped by `ar`, e.g., directly access the raw C-array in case it is a data member of `ar`. This limits the code's portability and future-proofness.

5) Use standard C++ code which isn't in the safe subset, e.g., `reinterpret_cast<T(*)[N]>(ar.data())`. This limits the contexts in which the code can be used, e.g., `reinterpret_cast` cannot be used in `constexpr` expressions. It is also error-prone and makes for less safe code.

Adding a member function to `std::array` to get a reference to its wrapped C-array would allow naturally passing the C-array to `foo` without having to use one of the above workarounds. This proposal would allow just calling `foo(ar.c_array())` in the `T(&)[N]` parameter type case, or `foo(&ar.c_array())` in the `T(*)[N]` parameter type case.

The scope of this proposal isn't large because functions accepting a reference (or pointer) to `T[N]` aren't common. However, especially prior to the addition of `std::array` to the C++ standard (which can now be used as the parameter type instead), such functions were a valid and safe choice for cases where a pre-determined number `N` of objects of type `T` had to be passed to a function, e.g.,

```
class Function {
   /// ...
};

// See https://en.wikipedia.org/wiki/Hilbert_transform#Conjugate_functions
Function InverseHilbertTransform(const Function (&)[2]);

std::array<Function, 2> hilbertParts = /* ... */;
auto func = InverseHilbertTransform(hilbertParts.c_array()); // Use proposed
member function.
```

Similarly, functions accepting a pointer to `T[N]` are a valid and safe choice in C for cases where a pre-determined number `N` of objects of type `T` should be passed to a function, e.g.,

```
// C header point3d.h
void f3dPoint(int (*coordinates)[3]);

// C++ code
extern "C" {
#include "point3d.h"
}
std::array<int, 3> coordinates = {1, 2, 3};
```

```
f3dPoint(&coordinates.c_array()); // Use proposed member function.
```

Making such existing functions compatible with `std::array` would encourage programmers to keep using `std::array` to represent their data, even if they need to make use of such existing functions. If the compatibility isn't added, then in some cases programmers would prefer using raw C-arrays to represent their data to avoid being forced to use one of the workarounds listed above, making their code less safe and modern. Since making writing safe and modern C++ code easier is a goal of the standard, the compatibility should be added.

IV. Impact On the Standard

There is no impact on the standard other than adding the proposed member function to `std::array`, the implementation of which is trivial: returning a reference to the wrapped C-array.

V. Design Decisions

1) This paper proposes adding a new `c_array()` member function to `std::array` which returns a reference to the wrapped C-array.

Note that `boost::array` already has a `c_array()` member function, returning a pointer, but according to previous discussion[1] it's only there for historical reasons. In any case, this is a minor concern because `boost::array` is now deprecated in favor of `std::array` anyway, and it is only mentioned here for completeness.

2) Alternatives considered and rejected:

> a) Adding a `std::array::elems` public data member of type `T[N]`, which would *not* be for exposition only. This would directly accessing `elems` to get a reference to the wrapped C-array. Note that `std::array::elems` does not exist at all in the current working draft of the next standard[1], whereas previous versions of the standard included such a data member for exposition only.
>
> The clear drawback of this solution is that it limits the flexibility of the `std::array` implementation. Specifically, Lawrence Crowl wrote: *"it might be better to leave `std::array::elems` "for exposition only" to allow alternate representations to allocate the array data dynamically. This might be of interest to the embedded community, having to deal with very limited stack sizes"*[1].
>
> Another drawback of this solution is that it would force `std::array` to remain an aggregate type in future versions of the standard. The original reasoning for

making `std::array` an aggregate type was for the class to be *"designed to function as closely as possible as a drop-in replacement for a traditional array… it must be implemented as an aggregate type… in order to support initializer syntax"*[3]. Since C++ now supports braced initialization for non-aggregate classes via constructors from `std::initializer_list`, it's possible for `std::array` to support initializer syntax without being an aggregate type, which might be desirable in the future to remove some limitations of aggregate types from `std::array`.

Finally, the paper introducing `std::array` relies on the fact that `std::array::elems` is for exposition only as a mitigating factor to the fact that *"Traditionally public data members are discouraged"*[3] and that `std::array::elems` is such a public data member, since *"the name of the data member is implementation defined so cannot be portably relied on"*[3] anyway. Making `std::array::elems` not for exposition only would remove this mitigating factor.

b) Changing the existing `std::array::data()` member function so it returns a reference to the C-array instead of a raw pointer.

A major drawback of this solution is that it breaks backwards compatibility with the existing implementations[1].

Another drawback is that changing the return type of this member function directly conflicts with the intent of the original paper which introduced `std::array`, which reads *"The return type of `data()` is chosen to be (const) T *… This maintains the similarity with `basic_string::data()`, avoids surprises if template type deduction is performed on the result, and reduces temptation to try clever manipulations…"*[3].

c) Adding an explicit conversion operator. This has been suggested in the past and rejected because *"it would be inconvenient to use"*[1].

d) Doing nothing. As described under 'Motivation and Scope' above, the drawback of this is backwards-incompatibility of `std::array` with existing functions which accept a reference (or pointer) to a C-array.

Specifically, the proposal in this paper was previously open as LWG issue 930 but closed as NAD (Not a Defect) because *"There are known other ways to do this, such as small inline conversion functions"*[1]. This paper makes the argument that such conversion functions have significant drawbacks, as described above under 'Motivation and Scope', workarounds 4 and 5.

In more detail, workaround No. 4, using a safe but non-standard way to get `ar`'s underlying C-array, could be implemented as: [4]

```
template <typename T, size_t N>
constexpr auto& c_array(std::array<T, N>& ar)
{
#if defined(_MSC_VER)
    return ar._Elems;
#elif defined(_LIBCPP_VERSION)
    return ar.__elems_;
#elif defined(__GLIBCXX__)
    return ar._M_elems;
#else
#error "unknown standard library"
#endif
}
```

which obviously has limited portability and isn't even future-proof to changes in the supported standard libraries' implementations.

Workaround No. 5, using a solution which is standard C++ but isn't in the safe subset, could be implemented as: [5]

```
template<typename T, size_t N>
auto inline c_array(std::array<T, N>& ar) {
    return reinterpret_cast<T(&)[N]>(*ar.data());
}
```

which has the drawback of being unusable in `constexpr` expressions.

VI. Proposed Wording

- Under [array.overview] add the following to the definition of std::array:

```
   using c_array_type = T[N];
   constexpr c_array_type& c_array() &;
   constexpr const c_array_type& c_array() const &;
   constexpr c_array_type&& c_array() &&;
```

- Add a subsection [array.c_array] after the subsection [array.data]:

```
   array::c_array [array.c_array]
   constexpr c_array_type& c_array() &;
   constexpr const c_array_type& c_array() const &;
   constexpr c_array_type&& c_array() &&;
```

Returns: An array of type `c_array_type` spanning the range `[data(), data() + size())`

- Under [array.zero] add:

  The type `c_array_type` is unspecified for a zero-sized array.

and make the change:

  The effect of calling `c_array()`, `front()`, or `back()` for a zero-sized array is undefined.

VII. Acknowledgements

Much of this proposal is based on work previously done by Niels Dekker and others on LWG issue 930.
Thanks to Walter E. Brown for reviewing this paper and for his helpful comments, as well as for his willingness to present the paper at WG21 meetings.
Thanks to Daniel Krügler, Erich Keane, Joe Gottman and Zhihao Yuan on the std-proposals newsgroup for their comments and help writing this proposal.

VIII. References

[1] LWG Issue 930, http://cplusplus.github.io/LWG/lwg-closed.html#930
[2] http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/n4618.pdf
[3] Alisdair Meredith, N1548, "A Proposal to Add a Fixed Size Array Wrapper to the Standard Library Technical Report", http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1548.htm
[4] Zhihao Yuan, https://groups.google.com/a/isocpp.org/d/msg/std-proposals/C6uVlKTOI_w/Hm-U72f-CwAJ
[5] Erich Keane, https://groups.google.com/a/isocpp.org/d/msg/std-proposals/C6uVlKTOI_w/A8PkF1n1CwAJ