Relaxing Restrictions on Arrays

Document #:	D1997R1
Date:	2020-01-05
Project:	Programming Language C++ Evolution Working Group

Reply-to:	Krystian Stasiowski <sdkrystian@gmail.com> Theodoric Stier <kerdek7@gmail.com>

Contents

1 Abstract

We propose endowing arrays with initialization and assignment from other arrays, placeholder semantics, pseudo-destructors, and the ability to serve as return types, in order to simplify the language of aggregates.

2 Motivation

2.1 Simplifying the language

Aggregates were created with the purpose of providing semantics and behavior which are reasonable to expect from such types, but, while aggregate classes enjoy most of the provisions, arrays appear to possess some artificial and confusing restrictions. It is currently possible to enjoy the semantics and behaviors of aggregate classes while using an array by wrapping it or using `std::array`, and such is often good practice, but arrays are often self-explanatory, and such wrapping only presents an unnecessary cognitive burden in such cases.

```
int samples_x[5];
int samples_y[5] = samples_x;
// ill-formed, but self-explanatory
```

```
struct samples { int s[5]; };
samples x;
    samples y = x; // OK, but why?
```

Beginners may not understand why arrays are element-wise copy/move constructible/assignable when they are data members but not when they are named by local variables. To an expert, this limitation may appear arbitrary, perhaps even backwards. We want to simplify the language by eliminating some perceived oddities of arrays. Arrays could be readily understood as like aggregate classes (except having elements which are referred to by subscripts instead of names and no members, bases, etc.).

Allowing the initialization and assignment of arrays makes the language more like other high level languages where array assignment is permitted. Addressing these caveats makes the language easier to learn and teach. In addition to this, allowing arrays to be initialized from other arrays, assigned, returned, and destructed makes writing generic code much easier, as it removes the need for special cases when a parameter is of array type.

2.1.1 Initializing array elements of aggregates

An initializer for an aggregate class object whose first element is an array with bound n must contain n initializer elements preceding those for any later element may be initialized. It would be useful and sensible to be able to initialize the object element with a single initializer element.

```
struct lp_3_point
{
    int coords[3];
    float power;
};

auto make_euclidean_3_point(int (&c)[3]) -> lp_3_point
{
    // must write four elements in order to initialize power
    return { c[0], c[1], c[2], 2.0f };
    // would make sense to initialize an array with an array
    return { c, 2.0f };
}
```

2.1.2 Initialization and assignment of members

If the user provides an assignment operator to a class having an array data member, they must explicitly iterate over the elements to be assigned, or wrap the array as above. It would be preferable and simpler to initialize or assign the array all at once.

```
class widget
{
    gadget g[4]; // user-provided assignment is now painful
};
```

2.1.3 Return types

Array return types are legible in trailing return type syntax, and such could often be self-explanatory and preferable. It appears arbitrary and unnecessary that they are forbidden.

```
auto make_coefs() -> int[3]
```

2.2 Advantages over `std::array`

The introduction of `std::array` in C++11 was a bandage over the more glaring problems of using built-in arrays. `std::array` can have some weird or even surprising semantics which can be avoided by using built-in arrays. Making arrays easier to use helps users enjoy their existing advantages.

2.2.1 List initialization issues

One of the more glaring problems faced by `std::array` is how brace elision may at times cause difficult to diagnose syntax errors. Consider the following:

```
struct vec3 { double x, y, z; };
std::array<double, 3> vec = {-1.0, 1.0, 0.0}; // OK
std::array<vec3, 2> vec_2 = {{-1.0, 1.0, 0.0}, {1.0, 0.0, -1.0}}; // ill-formed... but why?
```

This occurs because `std::array` is only considered to have a single element. Brace elision only kicks in when the *initializer-list* does not begin with left brace, which can lead to surprising results as demonstrated by this example. For an array type, this is not an issue:

```
struct vec3 { double x, y, z; };

double vec[3] = {-1.0, 1.0, 0.0}; // OK
vec3[2] vec_2 = {{-1.0, 1.0, 0.0}, {1.0, 0.0, -1.0}}; // also OK
```

As an array of type "array of N `T`" is defined to have N elements, both in the array sense as well as in the aggregate sense.

2.2.2 Declarator syntax

One of largest advantages that arrays hold over `std::array` is the greatly improved syntax used to declare multidimensional arrays. Consider the following declaration of an 3 dimensional array:

```
double matrix_3d[3][3][3];
```

Contrast with a similar declaration using `std::array`:

```
std::array<std::array<std::array<double, 3>, 3>, 3> matrix_3d;
```

The syntax for declaring a reference to an array type could be considered confusing, but this can be remedied through the use of `std::type_identity`, `std::add_lvalue_reference`, or the use of a convenience alias template. The preferred method would be through the use of alias template as to not preclude template argument deduction:

```cpp
template<typename T, std::size_t N>
using array_ref = T(&)[N];
// either of the following work
void f(array_ref<int, 3>);
void g(std::type_identity_t<int[3]>&);
void h(std::add_lvalue_reference_t<int[3]>);
```

2.2.3 Unknown bounds

Array types may be unbounded, and using such offers the benefits of placeholder semantics while providing a useful constraint. A specialization of std::array may at best have templated bound.

2.2.4 Compilation resources

Compilation resource consumption is improved by replacing uses of `std::array` with built-in arrays. Each specialization of `std::array` requires an amount of memory to be used by a compiler, and that memory is not freed right away. Such penalty is not associated with arrays of various sizes and element types because their functionality is built in.

2.2.5 Constant evaluation

Constant evaluation of code which uses `std::array` is not as fast as that which uses built-in arrays; such penalty is associated with every abstraction used in constant evaluation because the related code must be interpreted and checked for illegal behavior [Vandevoorde]. Consequently, many users prefer to process arrays with minimal abstraction. Replacing that code with built in functionality, which does not require such checks, allows further increased constant evaluation performance in addition to simpler code.

2.2.6 Implementation availability

Use of the standard library or templates is restricted in certain scenarios (such as on freestanding implementations).

2.3 "Mandatory" copy elision

In C++20, it is not possible to return array types. This makes the use of `std::array` mandatory if one wants to return an array prvalue to capitalize from the delayed prvalue materialization rules introduced in C++17. There is no reason that this shouldn't be allowed for array types. Consider the following code written in the absence of `std::array`:

```cpp
template<std::size_t N, typename T>
void make_filled(
  T(&arr)[N],
  const T& value)
{
  for (std::size_t i = 0; i < N; ++i)
    arr[i] = value;
}
```

To use this function, one would need to create and default-initialize a separate output array, and then assign each element to `value`. Conversely, once one is able to return an array type:

```cpp
template<
  std::size_t N,
  typename T,
  typename... Ns>
auto make_filled_helper(
```

```
  const T& value
  std::index_sequence<Ns...>)
    -> T[N]
{
  return {(Ns, value)...};
}

template<std::size_t N, typename T>
auto make_filled(
  const T& value)
{
  return make_filled_helper<N>(value,
    std::make_index_sequence<N>());
}
```

Using a helper template, it is possible to return a prvalue of array type, which causes any unnecessary initialization to be completely elided.

This same principle can be applied to actual copy elision (NRVO):

```
template<std::size_t N, typename T>
auto make_filled(
  const T& value) -> T[N]
{
  T out[N]
  for (std::size_t i = 0; i < N; ++i)
    out[i] = value;
  return out;
}
```

Allowing it to be optionally used to eliminate copies for named objects as well. This again makes arrays more similar to other existing types, simplifying the language.

2.4 More use in templates

Many of the existing algorithms and other utilities found in the standard library do not work when using array types. For example, using an array type as a container value type is not permitted as it isn't considered to be *CopyInsertable*. Additionally, most sequence modifying operations provded by the standard library do not have support for array types, as they are neither assignable nor copy constructible. Removing these constraints that are currently inplace would greatly simplify the language, and would allow the use of simpler, less surprising syntax.

For example, consider the following declaration of `std::fill`:

```
template<class ForwardIt, class T = typename iterator_traits<ForwardIt>::value_type>
constexpr void fill(ForwardIt first, ForwardIt last, const T& value);
```

With this addition of a default argument to facilitate the use of an initializer list as an argument for `value`, it makes it possible to use `std::fill` to assign arrays within a multidimensional array. Save for the fact that parameters cannot have array type, and that relational/equality operators using array operators yield unexpected results (and are thus deprecated), in all other aspects arrays will act like other object types, making it possible to write templates that work for all object types. This eliminates boilerplate for those who wish to write templates that work with all object types.

3 Proposal

We propose initialization and assignment of arrays by arrays, array return types, array pseudo-destructors, and reasonable type deduction for array placeholder types.

3.1 Initialization

Expressions of array type may be used as initializers for objects having the same type. The elements of the object are initialized using the corresponding elements of the value as if through a subscript expression.

C++20

```
std::string x[] = { "foo", "bar" };

std::string y0[] = x; // ill-formed


std::string y1[3] = x; // ill-formed


std::string y2[] = std::move(x); // ill-formed
```

Proposed

```
std::string x[] = { "foo", "bar" };

std::string y0[] = x; // ok
// the elements of y0 are "foo", "bar"

std::string y1[3] = x; // ill-formed
// the bounds do not match

std::string y2[] = std::move(x); // ill-formed
// elements of x are moved-from
```

The first bound of an array type in an initializing declaration can be deduced from the type of the initializer expression:

C++20

```
int a[] = { 0, 1 };

int a0[] = a; // ill-formed


int b[2][2] = { {0, 1}, {0, 1} };

int b0[][2] = b; // ill-formed


int b1[2][] = b; // ill-formed
```

Proposed

```
int a[] = { 0, 1 };

int a0[] = a; // ok
// decltype(a0) is int[2]

int b[2][2] = { {0, 1}, {0, 1} };

int b0[][2] = b; // ok
// decltype(b0) is int[2][2]

int b1[2][] = b; // ill-formed
// only the first bound can be deduced
```

3.2 Assignment

Expressions of array type may be used as the right hand side of built-in assignment for array objects having the same element type. The elements of the object are assigned using the corresponding elements of the value as if though a subscript expression.

C++20	Proposed

`int x[] = { 1, 2 };`
`int y[] = { 3, 4 };`
 	
`x = y; // ill-formed`
	`int x[] = { 1, 2 };`
	`int y[] = { 3, 4 };`
	
	`x = y; // ok`
	`// elements of x are 3 and 4`

`std::string x0[] = { "hello", "world" };`
`std::string y0[] = { "foo", "bar" };`
 	
`x0 = y0; // ill-formed`
	`std::string x0[] = { "hello", "world" };`
	`std::string y0[] = { "foo", "bar" };`
	
	`x0 = std::move(y0); // ok`
	`// elements of x0 are "foo" and "bar"`
	`// elements of y0 are moved from`
	`// assigned as if by x0[i] = std::move(y0)[i]`

3.3 Array return types

Array return types are permitted. "Mandatory" copy elision (RVO) (i.e. prvalue semantics) and copy elision (NRVO) is supported for arrays.

C++20	Proposed

`auto f() -> std::string[2]`
`{`
`  return { "foo", "bar" };`
`}`
`std::string x[2] = f(); // ill-formed`
	`auto f() -> std::string[2]`
	`{`
	`  return { "foo", "bar" };`
	`}`
	`std::string x[2] = f(); // ok`
	`// only one constructor call per element`

3.4 Pseudo-destructors

Pseudo-destructors are defined for array types; elements are destroyed in reverse subscript order.

C++20	Proposed

`using T = std::string[2];`
`T x = { "foo", "bar" };`
`x.~T(); // ill-formed`
	`using T = std::string[2];`
	`T x = { "foo", "bar" };`
	`x.~T(); // ok`
	`// two calls to string destructor`

3.5 Placeholder type deduction

The placeholder type `auto` is permitted as the element type of an array; the type of the placeholder, but not the bounds of the resulting array type, is deduced from the initializer. After deduction, the rules in [dcl.array] are used to deduce the first bound from the array, however, this only occurs when the type of the declaration is an array type (not a reference or pointer to array).

C++20

```
int a[] = { 42, 0 };
auto a0[] = a; // ill-formed

auto (&a1)[] = a; // ill-formed




auto b = { { 42, 0 }, { 42, 0 } };
auto b0[][] = b; // ill-formed


auto b1[][2] = b; // ill-formed

auto b2[2][] = b; // ill-formed
```

Proposed

```
int a[] = { 42, 0 };
auto a0[] = a; // ok

auto (&a1)[] = a; // ok
// deduces int, a1 is bound to the result of
// converting a to an array of unknown bound

auto b = { { 42, 0 }, { 42, 0 } };

auto b0[][] = b; // ill-formed
// only first bound can be deduced

auto b1[][2] = b; // ok

auto b2[2][] = b; // ill-formed
// only first bound may be deduced by [dcl.array]
```

3.6 Impact

Such initialization and assignment could simplify the implementation of container types by removing many of the corner cases where array types are arbitrarily inconvenient. This reduces the likelihood of programming errors.

C++20

```
template<size_t N>
class T
{
public:
    T & operator=(T const & other)
    {
        for(int i = 0; i < N; i++) // bug-prone
        {
            samples[i] = other.samples[i];
        }
    }
private:
    int samples[N];
};
```

Proposed

```
template<size_t N>
class T
{
public:
    T & operator=(T const & other)
    {
        samples = other.samples;
        // less bug-prone
    }
private:
    int samples[N];
};
```

We believe that providing this similarity by defining initialization and assignment from an array does not sacrifice any compatibility with the C language. The related semantics are not currently valid in any C program.

```
using T = U[N];
T x;
// well-formed under propoal, ill-formed in previous standards and in C
T y0{ x };
y0 = x;
```

```
[&]() -> T { return x; }();
auto y1[] = std::move(x);
x.~T();
```

Some C++ programs may have their meaning changed. For example, the value defined by `std::is_assignable` will change for some specializations.

4 Design choices

4.1 Assignment and initialization

For both assignment and initialization, the types of the arrays must match exactly, save for cv-qualification. For assignment, this ensures that when the assignment of an array is called for, it will always assign exactly the number of elements contained within the array. In the case of initialization, the same restriction exists for similar reasons: to avoid bugs and surprise resulting from the effect on left-over elements. Additionally, each element of the array is assigned or initialized as if by a subscript expression to carry the value category of the array expression to that of each element during initialization, resulting in the appropriate copy/move constructors being called.

4.2 Placeholder type deduction

The restriction of the placeholder type `auto` being prohibited as the element type of an array is removed. This is to preserve backwards compatibility by not changing the existing behavior of declarations using `auto` with an initializer of array type, and instead introducing new syntax to facilitate the deduction of the arrays element type without the array-to-pointer conversion that usually occurs. The deduction process does not deduce the bounds, and instead they are calculated from the initializer separately as specified in [dcl.array] p7, therefore code such as this:

```
int a[4];
int (&b)[] = a; // ill-formed
auto (&c)[] = a; // ill-formed
```

does not become well-formed, as the accompanying reference binding without the placeholder is not well-formed either. Normal deduction (i.e., for a declaration of the form `auto& t = e`) using a reference to the placeholder type `auto` will be unchanged. Placeholders for a deduced class type and `decltype(auto)` are not permitted as an arrays element type, and remain unchanged.

4.3 Pseudo-destructors

Currently, pseudo-destructors have no effect, but [P0593R5] will change them so that they end the lifetime of the object they are called upon. This should also be permitted for array objects, and have the effect of destroying the object, as specified by [dcl.init] p21. This would destroy the elements of the array in reverse order, and then the array itself. This useful when re-using storage with placement array new, which is now usable due to the resolution of [CWG2382].

4.4 Array return types and parameters

Regarding parameters of array type, it is best to leave them untouched unless there is overwhelming consensus for a change to be made. Currently, parameters of array type are adjusted to pointer types, so introducing such a large breaking change to the language would be unfavorable. Alternate syntax could be used, but it would be inconsistent with the rest of the declarator syntax for arrays and would only serve to further complicate the language.

On the other hand, array return types are a pure extension, allowing for the technique of using out parameters for arrays to be abandoned. Permitting copy elision for array types further solidifies them as first class types on par with classes.

5 Wording

All wording is relative to N4835 except for the changes under 5.3 which are relative to the proposed wording of [P0593R5].

5.1 Allow array assignment

Changes to [expr.ass] p2

2 In simple assignment ~~(=)~~of the form `E1 = E2`, when the left operand is not of array type, the object referred to by the left operand is modified by replacing its value with the result of the right operand. If the left operand is of type "array of *N* `T`", the right operand shall be of the same type (ignoring cv-qualification) and the effect is identical to performing `E1[`*i*`] = E2[`*i*`]` for each `0 =` *i* `<` *N*.

5.2 Allow array initialization

Changes to [dcl.init] p17 sub 5

(17.5) Otherwise, if the destination type is "array of *N* *cv1* `T`" or "array of unknown bound of *cv1* `T`":

(17.5.1) — If the initializer expression is a prvalue of type "array of *N* *cv2* `T`", the initializer expression is used to initialize the destination object.

(17.5.2) — Otherwise, if the source type is "array of *N* *cv2* `T`", each array element x_i is copy-initialized with the expression `e[`*i*`]` for `0 ≤` *i* `<` *N* where `e` is the initializer expression.

(17.5.3) — Otherwise, ~~if the destination type is an array, the object is initialized as follows.~~ let x_1, ..., x_k be the elements of the *expression-list*. ~~If the destination type is an array of unknown bound, it is defined as having *k* elements.~~ [...]

Changes to [dcl.init.list] p3 sub 2

(3.2) If `T` is an aggregate ~~class~~ and the initializer list has a single element of type *cv* `U`, where `U` is `T`, orif `T` is a class type, a class derived from `T`, the object is initialized [...]

Changes to [expr.type.conv] p2

2 [...] ~~If the initializer is a parenthesized optional *expression-list*, the specified type shall not be an array type.~~

5.3 Allow pseudo-destructor calls for arrays

This wording is relative to that of the changes proposed by [P0593R5]

Changes to [expr.prim.id.dtor] p2

2 If the *id-expression* names a pseudo-destructor, `T` shall be a scalar or array type and the *id-expression* shall appear as the right operand of a class member access [...]

Changes to [expr.ref] p3

3 [...] If the object expression is of scalar or array type, `E2` shall name the pseudo-destructor of that same type (ignoring cv-qualifications) and `E1.E2` is an lvalue [...]

Changes to [expr.call] p5

5 [...] If the postfix-expression names a pseudo-destructor, the postfix-expression must be a possibly-parenthesized class member access, and the function call destroys the object ~~of scalar type~~ denoted by the object expression of the class member access.

Changes to [dcl.init] p21

21 [...] Destroying an array destroys each element in reverse subscript order, and then ends the lifetime of the array object.

5.4 Allow returning arrays

Changes to [dcl.fct] p11

11 Functions shall not have a return type of function type ~~array or function~~, although they may have a return type of type pointer or reference to ~~such things~~function. There shall be no arrays of functions, although there can be arrays of pointers to functions.

5.5 Deducing arrays with `auto`

Changes to [dcl.array] p4

4 `U` is called the array *element type*; this type shall not be a placeholder type of the form *type-constraint*`???` `decltype(auto)`, a reference type, a function type, an array of unknown bound, or *cv* `void`.

Changes to [dcl.type.auto.deduct] p4

4 [...] Deduce a value for `U` using the rules of template argument deduction from a function call, where `P` is a function template parameter type and the corresponding argument is `e`, except that if `P` is an array type, `P&` is used in place of `P` in the synthesized function template. If the deduction fails, the declaration is ill-formed. Otherwise, `T'` is obtained by substituting the deduced `U` into `P`.

Changes to [dcl.array] p7

7 [...] In these cases, the array bound *N* is calculated from the ~~number of initial elements (say, *N*) supplied, and the type of the array is "array of *N* `U`".~~initializer as follows:

(7.1) — if the initializer expression is of type "array of *M* `T`" or is an initializer list with one element of type "array of *M* `T`", then *N* is *M*

(7.2) — otherwise, *N* is the number of *initializer-clause*s in the *braced-init-list* or *expression-list*.

The type of the array is "array of *N* `U`".

Changes to [dcl.init.aggr] p9

9 ~~An array of unknown bound initialized with a brace-enclosed initializer-list containing *n initializer-clauses* is defined as having *n* elements.~~

5.6 Copy elision for arrays

Changes to [class.copy.elision] p1

1 When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object or array of class objects, even if the constructor selected for the copy/move operation(s) and/or the destructor for the object or its elements have side effects. In such cases, the implementation treats the source and target of the omitted copy/move operation(s) as simply two different ways of referring to the same object. If the first parameter of the selected constructor is an rvalue reference to the object's type (or, in the case of an array, its element type), the destruction of that object occurs when the target would have been destroyed; otherwise, the destruction occurs at the later of the times when the two objects would have been destroyed without the optimization. [...]

(1.1) — in a `return` statement in a function with a class or array return type [...] the copy/move operation(s) can be omitted by constructing the automatic object or array elements directly into the function call's return object

5.7 Wording cleanup

Changes to [temp.deduct] p11 sub 10

(11.10) Attempting to create a function type in which a parameter has a type of `void`, or in which the return type is a function type ~~or array type~~.

6 Acknowledgements

Thank you to Will Wray for all the time spent brainstorming and his suggestions.

Thanks to Agustín Bergé, Peter Dimov, Vinnie Falco, and all others who discussed this in the CppLang Slack.

7 References

[CWG2382] Paul Sanders. 2018. Array allocation overhead for non-allocating placement new.
https://wg21.link/cwg2382

[P0593R5] Richard Smith. 2019. Implicit creation of objects for low-level object manipulation.
https://wg21.link/p0593r5

[Vandevoorde] Daveed Vandevoorde. C++ Constants.
https://www.youtube.com/watch?v=m9tcmTjGeho