Relaxing Restrictions on Arrays

Document #:	D1997R0
Date:	2019-11-25
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 bring consistency to the semantics of aggregates.

2 Motivation

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`. Wrapping data in a type with a descriptive name is often good practice, but some arrays are 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 artificial, perhaps even backwards. Arrays could be readily understood as like aggregate classes except having elements which are referred to by subscripts instead of names and no member functions, base classes, or operator overloads.

Array return types are legible in trailing return type syntax.

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

An aggregate class whose first element is an array with bound n must have n initializers before any later element may be initialized.

```
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 };
}
```

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.

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

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. This opens up several use cases, such as the following use for `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 with ease.

3 Proposal

We propose to define initialization and assignment of an array type by a similar array type, to define array return types, to define array pseudodestructors, and to define reasonable type deduction for array placeholder types.

3.1 Initialization

Expressions of array type may be used as initializers for array objects having the same element type. The elements of the object are initialized using the corresponding elements of the value. Array bounds are deduced when the declaration type is an unbounded array.

C++20	Proposed

`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`
	`std::string x[] = { "foo", "bar" };`
	`std::string y0[] = x; // the elements of y0 are "foo", "bar"`
	`std::string y1[3] = x; // ill-formed; bounds do not match`
	`std::string y2[] = std::move(x); // elements of x are moved-from`

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.

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; // the elements of x are 3, 4`

3.3 Array return types

Array return types are permitted. Copy elision is defined for arrays.

C++20

```
auto f() -> std::string[2]
{
    return { "foo", "bar" };
}
std::string x[2] = f(); // ill-formed
```

Proposed

```
auto f() -> std::string[2]
{
    return { "foo", "bar" };
}
std::string x[2] = f(); // 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

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

Proposed

```
using T = std::string[2];
T x = { "foo", "bar" };
x.~T(); // 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 is deduced from the initializer.

C++20

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

Proposed

```
int x[] = { 42, 0 };
auto y[] = x; // placeholder is int
```

3.6 Impact

Such initialization and assignment could simplify the implementation of container types such as `std::vector`, which 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 backward compatibility with the C language. The related semantics are not currently valid in any C or C++ program.

C++20

```
using T = U[N];
T x;
// all ill-formed
T y0{ x };
y0 = x;
[&]() -> T { return x; }();
auto y1[] = std::move(x);
x.~T();
```

Proposed

```
using T = U[N];
T x;
// all well-formed
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.

C++20

```
static_assert(!std::is_assignable_v<T(&)[N], T[N]>
static_assert(!std::is_assignable_v<T(&)[N], T(&)[
static_assert(!std::is_assignable_v<T(&)[N], T(&&)
```

Proposed

```
static_assert(std::is_assignable_v<T(&)[N], T[N]>);
static_assert(std::is_assignable_v<T(&)[N], T(&)[N]>);
static_assert(std::is_assignable_v<T(&)[N], T(&&)[N]>);
```

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 is done to ensure that when the assignment of an array is called for, it will always assign exactly the number of elements contained within the array, to avoid surprising the user through not assigning every element. In the case of initialization, the same restriction exists for similar reasons, as to not leave elements that are value or default-initialized. 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 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.

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` $\leq i <$ N.

5.2 Allow array initialization

Changes to [dcl.init] p17 sub 5

(17.5) Otherwise, if the destination type is an array:

(17.5.1) — If the source type is "array of M *cv1* `T`", and the destination type is "array of N *cv2* `U`" or "array of unknown bound of *cv2* `T`", M shall be N and `T` shall be `U`. Then, if the initializer expression is a prvalue, the initializer expression is used to initialize the destination object. Otherwise, each array element x_i is copy-initialized with the expression `e[`i`]` for `0 ≤ `i` < `N where `e` is the initializer expression.

(17.5.2) — 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 [...]

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*$_{opt}$ `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

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

7 References

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