Array assignment and initialization

Document #: D0000R0
Date: 2019-11-16
Project: Programming Language C++
Evolution Working Group
Reply-to: Krystian Stasiowski
<sdkrystian@gmail.com>
Theodoric Stier
<kerdek7@gmail.com>

1 Abstract

We propose allowing initialization, assignment, and placeholder semantics for array types with other arrays 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 the provisions, arrays appear to possess some artificial and confusing limitations. It is 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?
```

It should not make sense to a beginner 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_lp_3_point(int (&c)[3]) -> lp_3_point
{
    // Must write four elements in order to initialize power
    return { c[0], c[1], c[2], 1.0f };
    // Would make sense to initialize an array with an array
    return { c, 1.0f };
}
```

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
};
```

An added benefit of allowing the initialization and assignment of arrays is it that it makes the language more like other high level languages where array assignment is permitted. Giving arrays initialization and assignment semantics similar to that of scalar and class types makes the language easier to learn and teach when this caveat is addressed.

3 Proposal

We propose to define initialization of an array type by a like array type as element-wise and sequenced, to define array return types, and to define reasonable type deduction for array placeholder types. Such initialization and assignment could simplify the implementation of container types such as `std::vector`, which reduces the likelihood of programming errors.

We believe that providing this similarity by defining initialization and assignment from an array does not sacrifice any backward compatibility with the C language. None of the related syntax is valid in any C or C++ program. Some C++ programs may have their meaning changed. For example, the value defined by `std::is_assignable` will change for some specializations.

4 Wording

All wording is relative to N4835.

4.1 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` $\leq i < N$.

4.2 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 initializer expression is of array type and the cv-unqualified versions of the element types of the source and destination type differ, the program is ill-formed. Then, if the initializer expression is a prvalue, the initializer expression is used to initialize the destination object.

(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* or, if the source type is "array of *k* `T`", the *postfix-expressions* `e[0]`, …, `e[`*k* `- 1]` where `e` is the initializer expression. If the destination type is an array of unknown bound, it is defined as having *k* elements. […]

Changes to [dcl.init.list] p3

3 List-initialization of an object or reference of type `cv` `T` is defined as follows:

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`, or, if `T` is a class type, a class derived from `T`, the object is initialized […]

4.3 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.

4.4 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,~~ 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.~~

4.5 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

4.6 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~~.