Array size deduction in new-expressions

Timur Doumler (papers@timur.audio)

Document #: P1009R2
Date: 2019-02-22
Project: Programming Language C++
Audience: Core Working Group

Abstract

In this paper we propose to fix a particular inconsistency in the initialization rules of C++ by allowing array size deduction in *new-expression*s. This aligns their behaviour with that of initialization everywhere else in the language. Our proposed solution also creates consistency for a parenthesized list of values, which P0960 allows to to be used for initializing an array.

1 Motivation

Bjarne Stroustrup pointed out the following inconsistency in the C++ language:

```
double a[]{1,2,3};                  // this declaration is OK, ...
double* p = new double[]{1,2,3};    // ...but this one is ill-formed!
```

Jens Maurer provided the explanation why it doesn't work: For a *new-expression*, the expression inside the square brackets is currently mandatory according to the C++ grammar. When uniform initialization was introduced for C++11, the rule about deducing the size of the array from the number of initializers was never extended to the *new-expression* case. Presumably this was simply overlooked. There is no fundamental reason why we cannot make this work.

Admittedly, deducing the array size in a *new-expression* is code that probably only very few people would actually write. One could therefore argue that this is a problem not worth fixing.

However, when teaching C++ initialization rules, we observe that most people intuitively expect uniform initialization in a *new-expression* to follow the same rules as uniform initialization everywhere else in the language. This exception is very unfortunate and tends to upset and surprise people when pointed out to them.

The existence of such exceptions is exactly the reason why C++ initalization rules are so notorious for being complicated, and why most C++ developers struggle with them. There are just too many non-obvious inconsistencies. We therefore propose to remove this particular one—not because this is a problem that people would frequently run into (they don't), but because fixing it is straightforward, the fix is a pure extension that does not impact any other part of the standard, and it would make initialization rules in C++ simpler, more uniform, and easier to teach.

2 Solution

We propose to allow omitting the array bound in a *new-expression*, as long as the array size can be deduced from the initializer list, in the same way it is already allowed in regular array declarations:

C++17	This proposal

`double a[]{1,2,3};  // OK`
`double* p = new double[]{1,2,3}; // Error`
	`double a[]{1,2,3};  // OK`
	`double* p = new double[]{1,2,3}; // OK`

A special case are arrays with no elements. While a declaration of an object of such type is ill-formed, it is fine to allocate one in a *new-expression*:

```
int a[0]{};            // this declaration is ill-formed, ...
int* p = new int[0]{}; // ...but this one is OK!
```

This keeps consistency with C, where `malloc(0)` returns a (non-dereferenceable) pointer, and is occasionally useful in C++, e.g. in templates where the array size is a non-type template parameter. To be maximally consistent, we propose that an array size of `0` in a *new-expression* should be deduced if the initializer consists of empty braces:

C++17	This proposal

`double* p = new double[0]{}; // OK`
`double* p = new double[]{};  // Error`
	`double* p = new double[0]{}; // OK`
	`double* p = new double[]{};  // OK`

Here, both versions (with or without the `0`) would have the same effect. This way, array size deduction in *new-expression*s behaves the exact same way for any array size that is allowed in a *new-expression*.

Another variation of this inconsistency is an array initialized with a string literal. The proposed wording fixes this case, too:

C++17	This proposal

`char c[]{"Hello"};  // OK`
`char* d = new char[]{"Hello"};  // Error`
	`char c[]{"Hello"};  // OK`
	`char* d = new char[]{"Hello"};  // OK`

3 Interaction with P0960

[P0960] introduces aggregate initialization with a parenthesized list of values, which includes arrays. The proposed wording ensures consistency in this case, too:

P0960	P0960 + this proposal

`double a[](1,2,3);  // OK`
`double* p = new double[](1,2,3); // Error`
	`double a[](1,2,3);  // OK`
	`double* p = new double[](1,2,3); // OK`

4 Previous work

The issue discussed here was already mentioned in an earlier paper by Ville Voutilainen [P0965], although that paper did not propose a technical solution for it. It also mentioned another inconsistency dealing with pointers to pointers. It admitted that this other inconsistency "may be beyond fixing" and would require modifying the grammar in what "is certainly a breaking change". It is a much rarer corner case and we do not consider it here.

5 Proposed wording

The reported issue is intended as a defect report with the proposed resolution as follows. The effect of the wording changes should be applied in implementations of all previous versions of C++ where they apply. The changes are relative to the C++ working paper [Smith2018].

Modify [expr.new] paragraph 1 as follows:

> *noptr-new-declarator* :
> `[` *expression*$_{opt}$ `]` *attribute-specifier-seq*$_{opt}$
> *noptr-new-declarator* `[` *constant-expression* `]` *attribute-specifier-seq*$_{opt}$

Modify [expr.new] paragraph 6 as follows:

> Every *constant-expression* in a *noptr-new-declarator* shall be a converted constant expression of type `std::size_t` and shall evaluate to a strictly positive value. ~~T~~If the *expression* in a *noptr-new-declarator* is present, it is implicitly converted to `std::size_t`. [*Example:* Given the definition `int n = 42`, `new float[n][5]` is well-formed (because `n` is the *expression* of a *noptr-new-declarator*), but `new float[5][n]` is ill-formed (because `n` is not a constant expression). — *end example*]
>
> If the *type-id* or *new-type-id* denotes an array type of unknown bound ([dcl.array]), the *new-initializer* shall not be omitted; the allocated object is an array with `n` elements, where `n` is determined from the number of initial elements supplied in the *new-initializer* ([dcl.init.aggr], [dcl.init.string]).

Document history

— **R0**, 2018-10-08: Initial version

— **R1**, 2018-11-26: Added discussion of no-elements case; revised wording; made proposal a DR; added reference to P0965.

— **R2**, 2019-02-22: Added discussion of string literal case and P0960; revised wording.

Acknowledgements

Many thanks to Richard Smith and Tim Song for their help with the wording, and to JF Bastien and Patrice Roy for their comments.

References

[P0960] Ville Voutilainen and Thomas Köppe. Allow initializing aggregates from a parenthesized list of values. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0960r2.html, 2019-01-21.

[P0965] Ville Voutilainen. Initializers of objects with automatic and dynamic storage duration have funny inconsistencies. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0965r0.html, 2018-02-12.

[Smith2018] Richard Smith. Working Draft, Standard for Programming Language C++. https://github.com/cplusplus/draft, 2019-02-22.