Fixing `std::start_lifetime_as` and `std::start_lifetime_as_array`

Timur Doumler (papers@timur.audio)
Arthur O'Dwyer (arthur.j.odwyer@gmail.com)
Richard Smith (richardsmith@google.com)
Alisdair Meredith (ameredith1@bloomberg.net)
Robert Leahy (rleahy@rleahy.ca)

Document #:	P2679R1
Date:	2022-11-11
Project:	Programming Language C++
Audience:	Core Working Group, Library Working Group

Abstract

`std::start_lifetime_as` and `std::start_lifetime_as_array`, facilities to explicitly start the lifetime of an object of implicit-lifetime type inside a block of suitably aligned storage, was introduced in [P2590R2] and voted into C++23. We propose to fix some remaining issues before C++23 ships. We also discuss possible changes to the API that were considered but rejected.

1 Changes proposed

1.1 Allow size `0` for arrays of unknown bound

`std::start_lifetime_as` currently does not work with the value `0` for its second parameter (the size of the dynamic array to be created). Passing `0` is undefined behaviour. This API makes it unnecessarily difficult and error-prone for generic code to interact with it. We propose to allow the value `0`. This proposal resolves NB comment CA-086 targeting C++23. For the size `0` case, the pointer returned can only be used in the manner in which pointers past the end of an array can be used; in particular, the pointer cannot be dereferenced. In generic code, this restriction is typically fine, and allowing size `0` obviates the need to special-case size `0`:

```
void processData(unsigned char* dataFromNetwork, size_t numObjectsToRead) {
    Data* data = std::start_lifetime_as_array<Data>(dataFromNetwork, numObjectsToRead);
    for (size_t i = 0; i < numObjectsToRead; ++i)
       processObject(data[i]);
}
```

For the size `0` case, we further propose to allow `nullptr` as the value of the first parameter, as no storage is actually needed in this case:

```
Data* data = std::start_lifetime_as_array<Data>(nullptr, 0);  // OK
assert(data == nullptr); // holds
```

1.2 Make using wrong overload set ill-formed

`std::start_lifetime_as` creates objects of non-array type and arrays of known bound (static arrays); `std::start_lifetime_as_array` creates arrays of unknown bound (dynamic arrays). Using the wrong interface with the wrong type should be ill-formed:

```
unsigned char* buf = /* ... */
auto* p1 = std::start_lifetime_as<int[]>(buf); // UB, should be ill-formed
```

With the current wording, the code above is undefined behaviour as per [res.on.functions] p2.5. The *Mandates* clause needs to be adjusted to make this code ill-formed instead.

2 Changes considered (not proposed)

In the previous revision [P2679R0] of this paper, and in NB comment GB-087, we had proposed further changes to the API of `std::start_lifetime_as` and `std::start_lifetime_as_array`. We felt that the API was inconsistent with other APIs in the standard library that create objects and accept both array and non-array types, such as `std::make_shared` and `std::make_unique`. These have a version for non-array types, a version for array types of known bound, and a version for array types of unknown bound, respectively, all with the same name, whereas here, `std::start_lifetime_as` handles non-arrays and static arrays, and a version with a *different name*, `std::start_lifetime_as_array`, handles dynamic arrays. In other words, static arrays are handled with a function that does *not* have the `_array` suffix in the name.

Further, the template parameters seem inconsistent between the two overload sets. On the one hand, for `std::start_lifetime_as`, when used with an array type `U[N]` of known bound, the template argument that the user needs to provide is the type `U[N]` of the object being created (for example, `std::start_lifetime_as<int[16]>`). On the other hand, for `std::start_lifetime_as_array`, the template argument is not the type `U[]` of the object being created, but instead the type of its elements `U`.

Finally, the overloads for arrays of *unknown* bound (the ones with the suffix `_array`) return a pointer to the first element of the array, while the overloads without the suffix `_array`, when used with an array type of known bound, return a pointer to the array itself. In other words, a call to `std::start_lifetime_as_array<int>(p, 16)` will return an `int*`, but at the same time a call to `std::start_lifetime_as<int[16]>(p)` will return an `int(*)[16]`.

We therefore proposed in [P2679R0] to redesign this API such that all overloads have the same name `std::start_lifetime_as` (dropping the `_array` suffix for dynamic arrays), the template argument is always the type of the object created, and the return type is either a pointer to the object created (for non-arrays) or a pointer to the first element of the array (for arrays):

```
unsigned char* buf = /* ... */

// Proposed in P2679R0:
int* p1 = std::start_lifetime_as<int>(buf);
int* p2 = std::start_lifetime_as<int[10]>(buf);
int* p3 = std::start_lifetime_as<int[]>(buf, 10);
```

LEWG has pointed out that returning a pointer to the first element would mean that for arrays there will be no way to obtain a pointer to the array itself, since arrays are not pointer-interconvertible with their first element ([basic.compound] p4). In the case of a static array, this design would mean that the compile-time information about the size of the array, which is embedded in the type, would be lost. LEWG asked to revise the paper such that for arrays, `std::start_lifetime_as` returns a pointer to the created array. For static arrays, this design retains the possibility to pass the created array by reference, preserving the size information in the type:

```
void processBlock(Data (&block)[8]);

void doStuff() {
  // ...
  processBlock(*std::start_lifetime_as<Data[8]>(dataFromNetwork));
}
```

This design would also have made the overload set even more consistent, because then, for all possible types, `std::start_lifetime_as` would always return a pointer to the created object.

This design then went to CWG. They pointed out that while returning a pointer to a *static* array is useful for use cases like the above, we should never be returning a pointer to a *dynamic* array from a standard library function. Every existing standard library API that creates a dynamic array (*new-expression*, `operator new`, `std::make_shared`, `std::make_unique`, etc.) only ever returns a pointer to the first element, with no way to obtain a pointer to the whole array. Making returning a pointer to the array work in the dynamic array case would require major heroics in the wording, particularly for the size `0` case, as we do not have a well-defined concept of zero-length arrays (or pointers to them) in the language today. Notably, a dynamic array `T[]` is an incomplete type, so you cannot ever create an object of such a type. But you can also not create an object of type `T[n]`, because for dynamic arrays, `n` has a runtime value unknown to the type system. Therefore, the type of the object actually created falls outside of the C++ type system. We currently get around this problem in the wording by not specifying the type of the object created, but only describing the type in quotes ("array of `n T`") and never giving the user any way to actually refer to such an object (such as through a pointer or a reference).

Unlike for static arrays, a pointer to a dynamic array is also not particularly useful, as it does not contain any information about the size in its type. The only thing you could ever do with such a pointer is to dereference it and let it decay to a pointer to the first element. It is therefore more user-friendly to just give them a pointer to the first element.

This line of reasoning made us realise that `std::start_lifetime_as` (for non-arrays and for static arrays) and `std::start_lifetime_as_array` (for dynamic arrays) are two fundamentally different APIs. In the former, you specify the type of the object to be created, and get a pointer to this object; in the latter, you specify the element type and the desired array size, and get a pointer to the first element of the created array. Both APIs do the correct thing for the types they are meant for. They are also sufficiently different that they should have a different name, which justifies the `_array` suffix for the dynamic array version. The API currently in the C++23 draft is therefore the correct one (with the additions proposed in this paper).

For maximum clarity, the overload set for dynamic arrays should perhaps have been named `std::start_lifetime_as_dynamic_array` instead, but this name is uncomfortably long. LEWG argued further that this API is a low-level facility targeted at expert users who would be able to figure out the difference, particularly with the (still proposed) change to the *Mandates* clause to make usage of either API with the wrong kind of array type ill-formed. In the end, LEWG decided to leave the basic design of the API unchanged and reject NB comment GB-087.

Each version still needs four overloads: for non-`const`, `const`, `volatile`, and `const volatile` pointers, respectively. In an unrelated earlier design consideration, we looked at how we could reduce the amount of overloads. We considered an alternative approach for this API: have only one overload per version (instead of four), but restrict the parameter to `std::is_void`. However, that approach would forbid conversions, rendering the API difficult to use: in practice, the argument is rarely a `void*`, but typically a pointer to storage such as an `unsigned char*` or a `std::byte*`. We therefore rejected such an approach as unviable.

3 Proposed wording

The proposed changes are relative to the C++ working paper [N4917].

Modify [obj.lifetime] as follows:

Explicit lifetime management **[obj.lifetime]**

```
template<class T>
  T* start_lifetime_as(void* p) noexcept;
template<class T>
  const T* start_lifetime_as(const void* p) noexcept;
template<class T>
  volatile T* start_lifetime_as(volatile void* p) noexcept;
template<class T>
  const volatile T* start_lifetime_as(const volatile void* p) noexcept;
```

Mandates: `T` is an implicit-lifetime type and not an incomplete type.

Preconditions: [`(char*)p`, `(char*)p + sizeof(T)`) denotes a region of allocated storage that is a subset of the region of storage reachable through ([basic.compound]) `p` and suitably aligned for the type `T`.

Effects: Implicitly creates objects ([intro.object]) within the denoted region as follows: an object a of type `T`, whose address is `p`, and objects nested within a. The object representation of a is the contents of the storage prior to the call to `start_lifetime_as`. The value of each created object o of trivially-copyable type `U` is determined in the same manner as for a call to `bit_cast<U>(E)` ([bit.cast]), where `E` is an lvalue of type `U` denoting o, except that the storage is not accessed. The value of any other created object is unspecified. [*Note:* The unspecified value can be indeterminate. — *end note*]

Returns: A pointer to a.

```
template<class T>
  T* start_lifetime_as_array(void* p, size_t n) noexcept;
template<class T>
  const T* start_lifetime_as_array(const void* p, size_t n) noexcept;
template<class T>
  volatile T* start_lifetime_as_array(volatile void* p, size_t n) noexcept;
template<class T>
  const volatile T* start_lifetime_as_array(const volatile void* p, size_t n) noexcept;
```

Mandates: `T` is a complete type.

Preconditions: ~~`n > 0` is `true`.~~`p` is suitably aligned for an array of `T` or is null. `n <= size_t(-1) / sizeof(T)` is `true`. If `n > 0` is `true`, [`(char*)p`, `(char*)p + (n * sizeof(T))`) denotes a region of allocated storage that is a subset of the region of storage reachable through ([basic.compound]) `p`.

Effects: If `n > 0` is `true`, e~~E~~quivalent to: `return *start_lifetime_as<U>(p);` where `U` is the type "array of `n` `T`". Otherwise, there are no effects.

Returns: A pointer to the first element of the array, if any; otherwise, a pointer that compares equal to `p` ([expr.eq]).

Document history

— **R0**, 2022-10-15: Initial version.

— **R1**, 2022-11-11: Removed API redesign requested in NB comment GB-087 as it was rejected by LEWG; added support for `n == 0` case resolving NB comment CA-086.

Acknowledgements

Many thanks to Daniel Krügler, Peter Dimov, and Ville Voutilainen for their valuable comments.

References

[N4917] Thomas Köppe. Working Draft, Standard for Programming Language C++. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/n4917.pdf, 2022-09-05.

[P2590R2] Timur Doumler and Richard Smith. Explicit lifetime management. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2590r2.pdf, 2022-07-15.

[P2679R0] Timur Doumler, Arthur O'Dwyer, and Richard Smith. Fixing `std::start_lifetime_as` for arrays. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2679r0.pdf, 2022-10-15.