Fixing `std::start_lifetime_as` for arrays

Timur Doumler (papers@timur.audio)
Arthur O'Dwyer (arthur.j.odwyer@gmail.com)
Richard Smith (richardsmith@google.com)

Document #:	P2679R0
Date:	2022-10-15
Project:	Programming Language C++
Audience:	Library Evolution Working Group, Library Working Group

Abstract

`std::start_lifetime_as`, a facility 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. However, it has since emerged that for array types, the current API is broken and inconsistent. This paper proposes the necessary fixes.

1 The problem

[P2590R2] introduced a set of overloads that works for non-array types and array types of known bound, called `std::start_lifetime_as`, and a separate set of overloads for array types of unknown bound, called `std::start_lifetime_as_array`, using a different name with the `_array` suffix.

This API is 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.

Further, the current naming is also inconsistent with itself in multiple ways: the overloads that work for arrays of *unknown* bound have the suffix `_array` in the name, but the overloads that work for arrays of *known* bound do not. This does not make any sense and is highly confusing for users.

In the same way that the naming is inconsistent with existing standard APIs and also with itself, the template parameters are also inconsistent. 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]>`), while for `std::start_lifetime_as_array`, the template argument is not the type `U[]` of the object being created, but 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]`, which makes creating and using arrays with this facility very awkward and unintuitive.

2 Proposed solution

We propose to fix the specification of `std::start_lifetime_as` as follows:

— Add wording to the existing `std::start_lifetime_as<T>` overloads mandating that `T` is not an array type.

— Add new `std::start_lifetime_as<T>` overloads, for the case that `T` is an array type of known bound `U[N]`. These should work exactly like the non-array overloads, except that instead of returning a pointer to the array, they return a pointer to the first element.

— For array types of unknown bound, remove the existing `std::start_lifetime_as_array<U>` overloads and replace them with `std::start_lifetime_as<T>` overloads, where `T` is the type `U[]` and otherwise the functionality is the same.

This leaves us with a single overload set that covers all three cases (non-arrays, arrays of known bound, and arrays of unknown bound). This is clean, user-friendly, and consistent 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`.

In each case, the name of the function is `std::start_lifetime_as`, the template parameter is the type of the object being created, and the return value is either a pointer to the object being created (for non-arrays) or to its first element (for arrays).

We consider this proposal a critical bugfix for `std::start_lifetime_as`. We therefore strongly recommend that it should be adopted in the C++23 timeframe, and that `std::start_lifetime_as` should not be shipped without this bugfix applied.

This proposal fixes a submitted NB comment targeting C++23.

3 Tony table

Before	**After**
`unsigned char* buf = /* ... */;`	`unsigned char* buf = /* ... */;`
`int* p1 = std::start_lifetime_as<int>(buf);`	`int* p1 = std::start_lifetime_as<int>(buf);`
`int(* p2)[10] = std::start_lifetime_as<int[10]>(buf);`	`int* p2 = std::start_lifetime_as<int[10]>(buf);`
`int* p3 = std::start_lifetime_as_array<int>(buf, 10);`	`int* p3 = std::start_lifetime_as<int[]>(buf, 10);`

4 Design considerations

The design intent is that using the wrong interface with the wrong type should lead to a compile error: `std::start_lifetime_as<int[]>(buf)`, `std::start_lifetime_as<int>(buf, 10)`, and `std::start_lifetime_as<int[10]>(buf, 10)` should all be ill-formed and not necessarily SFINAE-friendly, and the same should be true for using either interface with a non-implicit-lifetime type. Simultaneously, the version for non-arrays and the one for arrays with known bound have the same function signatures (apart from the return type), so they need to coexist in such a way that the correct one is selected when used correctly. We therefore need to specify the correct combination of *Mandates:* and *Constraints:* clauses to achieve this.

Each of the three versions (non-arrays, arrays with known bound, and arrays with unknown bound) also still needs to contain four overloads for completeness: for `T*`, `const T*`, `volatile T*`, and `const volatile T*`, respectively. The complete overload set thus now contains 12 functions (up from 8 previously). To reduce the amount of overloads and avoid the combinatorial explosion, we considered an alternative approach: have only one overload per version but the parameter is restricted to `is_void`. However, that approach would forbid conversions. This would render 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.

5 Proposed wording

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

Modify header `<memory>` synopsis [memory.syn] as follows:

```
// [obj.lifetime] Explicit lifetime management

~~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;~~
~~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;~~

template<class T>
  T* start_lifetime_as(void* p) noexcept;                                // T is not array
template<class T>
  const T* start_lifetime_as(const void* p) noexcept;                    // T is not array
template<class T>
  volatile T* start_lifetime_as(volatile void* p) noexcept;              // T is not array
template<class T>
  const volatile T* start_lifetime_as(const volatile void* p) noexcept; // T is not array
template<class T>
  U* start_lifetime_as(void* p) noexcept;                                // T is U[N]
template<class T>
  const U* start_lifetime_as(const void* p) noexcept;                    // T is U[N]
template<class T>
  volatile U* start_lifetime_as(volatile void* p) noexcept;              // T is U[N]
template<class T>
  const volatile U* start_lifetime_as(const volatile void* p) noexcept; // T is U[N]
template<class T>
  U* start_lifetime_as(void* p, size_t n) noexcept;                      // T is U[]
template<class T>
  const U* start_lifetime_as(const void* p, size_t n) noexcept;          // T is U[]
template<class T>
  volatile U* start_lifetime_as(volatile void* p, size_t n) noexcept;    // T is U[]
template<class T>
  const volatile U* start_lifetime_as(const volatile void* p, size_t n) noexcept;
                                                                         // T is U[]
```

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;
```

Constraints: `T` is not an array type.

Mandates: `T` is an implicit-lifetime type.

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

Constraints: `T` is an array type of known bound `U[N]`.

Effects: Equivalent to `start_lifetime_as<T>(p)` for non-array types, except that the return value is a pointer to the first element of *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;
```

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

Mandates: `T` is an array type of unknown bound `U[]`.

Preconditions: `n > 0` is `true`.

Effects: Equivalent to: `return` ~~`*start_lifetime_as<U>(p)`~~`start_lifetime_as<V>(p)`; where ~~`U`~~`V` is the type "array of `n` `U`".

Acknowledgements

Many thanks to Daniel Krügler and Peter Dimov for their valuable comments on the LWG reflector regarding this issue.

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.