Explicit lifetime management

Timur Doumler (papers@timur.audio)
Richard Smith (richardsmith@google.com)

Document #: P2590R2
Date: 2022-07-15
Project: Programming Language C++
Audience: Library Working Group, Core Working Group

Abstract

This paper proposes a new standard library facility `std::start_lifetime_as`. For objects of sufficiently trivial types, this facility can be used to efficiently create objects and start their lifetime to give programs defined behaviour, without running any constructor code. This proposal completes the functionality proposed in [P0593R6] and adopted for C++20 by providing the standard library portion of that paper (only the core language portion of that paper made it into C++20).

1 Motivation

1.1 Creating objects in storage obtained from non-"blessed" functions

Since C++20, certain functions in the C++ standard library such as `malloc`, `bit_cast`, and `memcpy` are defined to *implicitly create objects* [P0593R6]. As a result, the following code is no longer undefined behaviour:

```
struct X { int a, b; };

X* make_x() {
  X* p = (X*)malloc(sizeof(struct X)); // implicitly creates an object of type X
  p->a = 1;
  p->b = 2;
  return p;
}
```

However, if the memory allocation or memory mapping function is not on this list of "blessed" standard library functions, code like the above still has undefined behaviour in C++20. We are accessing an object through a pointer to `X`, however there is no object of type `X` within its lifetime at that memory location.

For non-standard syscalls such as `mmap` on POSIX systems and `VirtualAlloc` on Windows systems, the implementation can specify that those functions implicitly create objects, and document that. Unfortunately, such a specification is typically absent, and therefore the code is technically undefined behaviour. In practice, this will typically work regardless, because the compiler cannot introspect the implementation of the syscall and prove that it doesn't perform `new (p) std::byte[n]` on its returned pointer.

But what about non-standard memory allocation or memory mapping functions that are provided by the user? Consider, for example, a library providing a memory pool, where the storage reuse is expressed in C++ code rather than in a syscall and is visible to the compiler. The current C++20 wording does not provide a solution for this use case, and code using such storage will be undefined behaviour.

We propose a standard library facility `std::start_lifetime_as` to tell the compiler explicitly that an object should be created at the given memory location without running any initialisation code:

```
struct X { int a, b; };

X* make_x() {
  X* p = std::start_lifetime_as<X>(myMalloc(sizeof(struct X));
  p->a = 1;
  p->b = 2;
  return p;
}
```

1.2 Deserialising objects from a sequence of bytes

Consider a C++ program that receives a sequence of bytes, perhaps over a network or from disk, and it knows that those bytes are a valid object representation of type `X`. How can it efficiently (i.e. without running any constructor code) obtain an `X*` that can be legitimately used to access the object? Any attempt involving `reinterpret_cast` will result in undefined behaviour:

```
void process(Stream* stream) {
  std::unique_ptr<char[]> buffer = stream->read();
  if (buffer[0] == FOO)
    processFoo(reinterpret_cast<Foo*>(buffer.get())); // undefined behaviour
  else
    processBar(reinterpret_cast<Bar*>(buffer.get())); // undefined behaviour
}
```

How can we make this program well-defined without sacrificing efficiency? If the destination type is a trivially-copyable implicit-lifetime type, this can be accomplished by copying the storage elsewhere, using placement new of an array of byte-like type, and copying the storage back to its original location, then using `std::launder` to acquire a pointer to the newly-created object, and finally relying on the compiler to optimise away all the copying. However, this would be very verbose and hard to get right. For expressivity and optimisability, a combined operation to create an object of implicit-lifetime type in-place while preserving the object representation may be useful. This is exactly what `std::start_lifetime_as` is designed to do:

```
void process(Stream* stream) {
  std::unique_ptr<char[]> buffer = stream->read();
  if (buffer[0] == FOO)
    processFoo(std::start_lifetime_as<Foo>(buffer.get())); // OK
  else
    processBar(std::start_lifetime_as<Bar>(buffer.get())); // OK
}
```

Note that in both of these use cases, the lifetime of the object is being started, however no constructor is actually being called and no code runs to achieve this. Just like implicit object creation, `std::start_lifetime_as` only works for *implicit-lifetime types*, i.e. types that are either aggregates or have at least one trivial eligible constructor and a trivial, non-deleted destructor. Note that if an object so created has subobjects that are themselves not of implicit-lifetime type, such subobjects would *not* be implicitly created along with the parent object.

1.3 Difference between `std::start_lifetime_as` and `std::launder`

Note how `std::start_lifetime_as` differs from `std::launder`. As far as the C++ abstract machine is concerned, `std::start_lifetime_as` actually creates a new object and starts its lifetime (even if no code runs). On the other hand, `std::launder` never creates a new object, but can only be used to obtain a pointer to an object that already exists at the given memory location, with its lifetime already started through other means. This is actually a common misconception about `std::launder`. Creating a library facility that actually does the thing that `std::launder` does not do, but is sometimes mistakenly assumed to do, would help remove this pitfall.

2 History

[P0593R5] had wording for both a core language portion and a standard library portion, and this paper in its entirety has already been approved by EWG and LEWG for C++20. The core language portion was then carried over into revision [P0593R6] and made it into C++20. However, the standard library portion did not, because LWG did not have enough time to review the wording before the C++20 cutoff date. In this paper, we have extracted this still-missing library part from [P0593R5] and are hereby proposing it again.

3 Design

We have taken the existing design from [P0593R5] and added `const` and `const volatile` overloads for completeness and consistency. To avoid the combinatorial explosion, we considered a constrained additional template parameter approach, such that the parameter is restricted to `is_void`, but that approach would forbid conversions. This will not work in practice because the argument is rarely a `void*`, but typically a pointer to storage, such as `unsigned char*`. The constrained template approach is therefore not viable.

`std::start_lifetime_as` and `std::start_lifetime_as_array` can never throw an exception because they do not actually run any code, so we added `noexcept`. We further added a precondition missing in [P0593R5] stating that the passed-in storage is suitably aligned for the type of object being created.

With the current design, `std::start_lifetime_as` handles non-array types as well as arrays of known bound, while `std::start_lifetime_as_array` handles arrays of unknown bound. It has been pointed out that this is inconsistent with `std::make_shared` and `std::make_unique`, where arrays of unknown bound are handled by a constrained function template with the same name, rather than a function with the `_array` suffix. We believe that `std::start_lifetime_as` could be redesigned to be consistent with this. This also raises the question of whether the case of arrays of known bound should also be handled by a separate constrained function template, again like `std::make_shared` and `std::make_unique`. With that, `start_lifetime_as<int[16]>(storage)` could be made to return an `int*`. With the current design, it returns an `int(*)[16]`. However, with the current deadline for the C++23 CD, we cannot consider further design changes at this stage, because that would mean deferring the whole feature to C++26. We therefore prefer this to be raised as an NB comment.

Finally, we discussed how `std::start_lifetime_as_array` should handle the case of `n == 0`. On the one hand, it seems useful to support it: imagine that we receive an optional sequence of elements over a network, preceded by a header that tells you how many elements there are, and we `std::start_lifetime_as_array` that sequence before accessing the elements. If there are no elements, the call to `std::start_lifetime_as_array` has no effect. On the other hand, it is not clear what the return value should be in this case. CWG and LWG therefore decided to not add support for the `n == 0` case at this stage. If necessary, this can also be handled via an NB comment.

4 Proposed wording

The proposed changes are relative to the C++ working draft [N4910].

Add a new paragraph below [basic.compound] paragraph 4 ("Two objects `a` and `b` are *pointer-interconvertible* if"...) as follows:

> A byte of storage b is *reachable through* a pointer value that points to an object x if there is an object y, pointer-interconvertible with x, such that b is within the storage occupied by y, or the immediately-enclosing array object if y is an array element.

Modify [intro.object] paragraph 13 as follows:

> Any implicit or explicit invocation of a function named `operator new` or `operator new[]` implicitly creates objects in the returned region of storage and returns a pointer to a suitable created object. [*Note:* Some functions in the C++ standard library implicitly create objects ([obj.lifetime], [allocator.traits.members] ,[c.malloc], [cstring.syn], [bit.cast]). — *end note*]

In header `<memory>` synopsis [memory.syn], add the following after the declarations of `std::align` and `std::assume_aligned`:

```
// [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;
```

Add the following subclause immediately after [ptr.align]:

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.
>
> *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>
  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;
```

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

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

Modify [ptr.launder] as follows:

`template<class T> [[nodiscard]] constexpr T* launder(T* p) noexcept;`

Mandates: `!is_function_v<T> && !is_void_v<T>` is `true`.

Preconditions: `p` represents the address *A* of a byte in memory. An object *X* that is within its lifetime and whose type is similar to `T` is located at the address *A*. All bytes of storage that would be reachable through ([basic.compound]) the result are reachable through `p` ~~(see below)~~.

Returns: A value of type `T*` that points to X.

Remarks: An invocation of this function may be used in a core constant expression if and only if the (converted) value of its argument may be used in place of the function invocation. ~~A byte of storage *b* is reachable through a pointer value that points to an object *Y* if there is an object *Z*, pointer-interconvertible with *Y*, such that *b* is within the storage occupied by *Z*, or the immediately-enclosing array object if *Z* is an array element.~~

Add feature test macro `__cpp_lib_start_lifetime_as` for header `<memory>` with a suitable value to Table 36 in [support.limits.general].

Document history

— **R0**, 2022-05-15: Initial version.

— **R1**, 2022-06-15: Expanded motivation; various wording fixes following CWG review.

— **R2**, 2022-07-15: Added `const` and `const volatile` overloads, `noexcept`, and alignment precondition; moved wording for *reachable through* from [ptr.launder] to [basic.compound]; minor wording fixes following CWG and LWG review.

Acknowledgements

Many thanks to Jens Maurer and Hubert Tong for their help with the wording.

References

[N4910] Thomas Köppe. Working Draft, Standard for Programming Language C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/n4910.pdf, 2022-03-17.

[P0593R5] Richard Smith. Implicit creation of objects for low-level object manipulation. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0593r5.html, 2019-10-06.

[P0593R6] Richard Smith. Implicit creation of objects for low-level object manipulation. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0593r6.html, 2020-02-14.