Explicit lifetime management

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

Document #:	P2590R1
Date:	2022-06-15
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 directly create objects and start their lifetime on-demand to give programs defined behaviour. 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

Since C++20, certain functions in the C++ standard library such as `malloc`, `bit_cast`, and `memcpy` implicitly create objects and start their lifetime [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));
  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 functions such as `mmap` on POSIX systems and `VirtualAlloc` on Windows systems, the implementation can ensure that those functions implicitly create objects, and document that. In the absence of such documentation, we probably still won't hit undefined behaviour in practice, because the compiler typically 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;
}
```

An even more interesting use case arises when a C++ program is given a sequence of bytes (perhaps from a disk or network), and it knows those bytes are a valid representation of type `X`. How can it efficiently obtain a `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. 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 also 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.

[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 actually 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.

2 Proposed wording

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

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<typename T> T* start_lifetime_as(void *p);
template<typename T> volatile T* start_lifetime_as(volatile void *p);
template<typename T> T* start_lifetime_as_array(void *p, size_t n);
template<typename T> volatile T* start_lifetime_as_array(volatile void *p, size_t n);
```

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

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

```
template<typename T> T* start_lifetime_as(void *p);
template<typename T> volatile T* start_lifetime_as(volatile void *p);
```

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

Requires: [`p`, `(char*)p + sizeof(T)`) denotes a region of allocated storage that is a subset of the region of storage reachable ([ptr.launder]) through `p`.

Effects: Implicitly creates objects 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 as if 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 may be indeterminate. — *end note*]

Returns: A pointer to `A`.

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

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

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.

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.