# Explicit lifetime management

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

| | |
|---|---|
| Document #: | P2590R0 |
| Date: | 2022-05-16 |
| 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 originally proposed in [P0593R6] 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.

[P0593R6] addressed this use case by proposing a standard library facility `std::start_lifetime_as`, in addition to the proposed core language changes, to tell the compiler explicitly that an object should be created at the given memory location:

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

Note that 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, `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.

This direction has already been approved by EWG and LEWG. The core language portion of [P0593R6] 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 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* ]

Modify [allocator.requirements.general] as follows:

`a.allocate(n)`

> *Result:* `X::pointer`
>
> *Effects:* Memory is allocated for an array of `n T` and such an object is created but array elements are not constructed. [ *Example:* When reusing storage denoted by some pointer value `p`, `start_lifetime_as_array<T>(p, n)` ~~`launder(reinterpret_cast<T*>(new (p) byte[n * sizeof(T)]))`~~ can be used to implicitly create a suitable array object and obtain a pointer to it. — *end example* ]
>
> *Throws:* `allocate` may throw an appropriate exception.
>
> [ *Note:* It is intended that `a.allocate` be an efficient means of allocating a single object of type `T`, even when `sizeof(T)` is small. That is, there is no need for a container to maintain its own free list. — *end note* ]
>
> *Remarks:* If `n == 0`, the return value is unspecified.

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 through `p`.

*Effects:* Implicitly creates objects within the denoted region, including an object `A` of type `T` whose address is `p`. The object representation of `A` is the contents of the storage prior to the call to `start_lifetime_as` as if by calling `memcpy` from a copy of the original storage, except that the storage is not accessed.

*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_at` for header `<memory>` with a suitable value to Table 36 in [support.limits.general].

## 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.

[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.