Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 **P0813R1**
Date: 2019-02-20
Reply to: Nicolai Josuttis (nico@josuttis.de)
Audience: LEWG, LWG
Prev. Version: P0813R0

construct() shall Return the Replaced Address Rev. 1

History

Updates in Rev1:

- Adapted to the current working draft for C++20, N4800

Motivation

As described in P0532R0 ("On launder()"), we usually should use the pointer passed to placement new, because not using it after the call can become a problem under some circumstances:

```
struct X {
  const int n;
  double d;
};
X* p = new X{7, 8.8};
new (p) X{42, 9.9};     // request to place a new value into p
int i = p->n;           // undefined behavior (i is probably 7 or 42)
auto d = p->d;          // also undefined behavior (d is probably 8.8 or 9.9)
```

The reason is the current memory model, written in

> **3.8 Object lifetime [basic.life]**
>
> If, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, a new object is created at the storage location which the original object occupied, a pointer that pointed to the original object, a reference that referred to the original object, or the name of the original object will automatically refer to the new object and, once the lifetime of the new object has started, can be used to manipulate the new object, if:
> (8.1) — the storage for the new object exactly overlays the storage location which the original object occupied, and
> (8.2) — the new object is of the same type as the original object (ignoring the top-level cv-qualifiers), and
> (8.3) — the type of the original object is not const-qualified, and, if a class type, does not contain any non-static data member whose type is const-qualified or a reference type, and
> (8.4) — the original object was a most derived object (1.8) of type T and the new object is a most derived object of type T (that is, they are not base class subobjects).

Reinitializing the passed pointer solves the problem:

```
struct X {
  const int n;
  double d;
};
X* p = new X{7};
p = new (p) X{42};      // request to place a new value into p
int i = p->n;           // OK, i is 42, because p was reinitialized
                        //    by the return value of placement new
```

The standard itself also gives an example in **1.8 The C++ object model [intro.object]:**

> [*Example:*
> `struct X { const int n; };`
> `union U { X x; float f; };`
> `void tong() {`
> `  U u = {{ 1 }};`
> `  u.f = 5.f;` *// OK, creates new subobject of* `u`

```
    X *p = new (&u.x) X {2};      // OK, creates new subobject of u
    assert(p->n == 2);            // OK
    …
    assert(u.x.n == 2);           // undefined behavior, u.x does not name new subobject
}
```
—end example]

However, inside the standard library we have several function wrapping calls of placement new.
The most important examples are the `construct()` calls of allocators.
Currently they return void:

According to the Cpp17allocator requirements (**15.5.3.5 Allocator requirements [allocator.requirements]**) `a.construct(c,args)` has a return type `void`, so that no return value of placement new can be used:

Expression	Return type	Assertion/note pre-/post-condition	Default
`a.construct(c, args)`	(not used)	Effect: Constructs an object of type `C` at `c`	`::new ((void*)c) C(forward< Args> (args)...)`

Thus, we can't use the return value of the underlying placement new here, which means that currently the standard allocator interface provides no way to deal with the return value of placement new.

As a result, containers and wrapper types have no way to avoid undefined behavior, if they use allocators and change the value of elements that are const, virtual, references, or whatever is listed in 3.8 Object lifetime [basic.life].

While there is still discussion in the CWG whether to relax 3.8 Object lifetime [basic.life], we should force `construct()` to return the passed (and replaced) pointer to give types using allocators a chance to avoid any unnecessary undefined behavior.

Acknowledgements

Thanks to all who incredibly helped me to prepare this paper, such as all people in the C++ concurrency and library working group.

Proposed Wording

(against N4800)

In Table 34 — Cpp17Allocator requirements

for `a.construct(c,args)` change the return type

from

(not used)

to

c

In **19.10.9 Allocator traits [allocator.traits]**

replace:

```
template <class T, class... Args>
  static void construct(Alloc& a, T* p, Args&&... args);
```

by:

```
template <class T, class... Args>
  static T* construct(Alloc& a, T* p, Args&&... args);
```

In: **19.10.9.2 Allocator traits static member functions [allocator.traits.members]**

replace:

```
template <class T, class... Args>
static void construct(Alloc& a, T* p, Args&&... args);
```

5 *Effects:* Calls `a.construct(p, std::forward<Args>(args)...)` if that call is well-formed; otherwise, invokes `::new (static_cast<void*>(p)) T(std::forward<Args>(args)...)`.

by:

```
template <class T, class... Args>
static T* construct(Alloc& a, T* p, Args&&... args);
```

5 *Effects:* Calls **`p =`** `a.construct(p, std::forward<Args>(args)...)` if that call is well-formed; otherwise, invokes **`p =`** `::new (static_cast<void*>(p)) T(std::forward<Args>(args)...)`.

Returns: **p**

In **19.12.3 Class template polymorphic_allocator [mem.poly.allocator.class]**

replace

```
template <class T, class... Args>
  void construct(T* p, Args&&... args);
```

by:

```
template <class T, class... Args>
  T* construct(T* p, Args&&... args);
```

In **19.12.3.2 polymorphic_allocator member functions [mem.poly.allocator.mem]**

Modify as follows:

```
template<class T, class... Args>
~~void~~ T* construct(T* p, Args&&... args);
```

5 Mandates: Uses-allocator construction of T with allocator *this (see 19.10.8.2) and constructor `arguments std::forward<Args>(args)...` is well-formed.

6 Effects: Construct a T object in the storage whose address is represented by p by uses-allocator construction with allocator *this and constructor arguments `std::forward<Args>(args)...` .

Returns: **p**

7 Throws: Nothing unless the constructor for T throws.

In **19.13.1 Header <scoped_allocator> synopsis [allocator.adaptor.syn]**
replace

```
template <class T, class... Args>
  void construct(T* p, Args&&... args);
```

by:

```
template <class T, class... Args>
  T* construct(T* p, Args&&... args);
```

In **19.13.4 Scoped allocator adaptor members [allocator.adaptor.members]**
Modify as follows:

```
template<class T, class... Args>
~~void~~ T* construct(T* p, Args&&... args);
```

9 Effects: Equivalent to:

```
return apply([p, this](auto&&... newargs) {
               OUTERMOST_ALLOC_TRAITS(*this)::construct(
                 OUTERMOST(*this), p,
                 std::forward<decltype(newargs)>(newargs)...);
             },
             uses_allocator_construction_args<T>(inner_allocator(),
                                                 std::forward<Args>(args)...));
```

In **21.2.1 General container requirements [container.requirements.general]**

Modify as follows:

(15.1) — T is Cpp17DefaultInsertable into X means that the following expression is well-formed:

```
p = allocator_traits<A>::construct(m, p)
```

(15.2) — An element of X is default-inserted if it is initialized by evaluation of the expression

```
p = allocator_traits<A>::construct(m, p)
```

where p is the address of the uninitialized storage for the element allocated within X.

(15.3) — T is Cpp17MoveInsertable into X means that the following expression is well-formed:

```
p = allocator_traits<A>::construct(m, p, rv)
```

and its evaluation causes the following postcondition to hold: The value of *p is equivalent to the value of rv before the evaluation.

[Note: rv remains a valid object. Its state is unspecified —end note]

(15.4) — T is Cpp17CopyInsertable into X means that, in addition to T being Cpp17MoveInsertable into X, the following expression is well-formed:

```
p = allocator_traits<A>::construct(m, p, v)
```

and its evaluation causes the following postcondition to hold: The value of v is unchanged and is equivalent to *p.

(15.5) — T is Cpp17EmplaceConstructible into X from args, for zero or more arguments args, means that the following expression is well-formed:

```
p = allocator_traits<A>::construct(m, p, args)
```

(15.6) — T is Cpp17Erasable from X means that the following expression is well-formed:

```
allocator_traits<A>::destroy(m, p)
```

[Note: A container calls `p = allocator_traits<A>::construct(m, p, args)` to construct an element at p using args, with m == get_allocator().

The default construct in allocator will call `p = ::new((void*)p)T(args)`, but specialized allocators may choose a different definition. —end note]