Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 **P0813R0**
Date: 2017-10-13
Reply to: Nicolai Josuttis (nico@josuttis.de)
Audience: LEWG, LWG
Prev. Version:

construct() shall Return the Replaced Address

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 allocator requirements (**17.5.3.5 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.

Proposed Wording

(against N4687)

In Table 31 — Allocator requirements

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

(not used)

to

c

In **23.10.8 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: **23.10.8.2 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 **23.12.3 Class template polymorphic_allocator [mem.poly.allocator.class]**
replace

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

template <class T1, class T2, class... Args1, class... Args2>
  void construct(pair<T1,T2>* p, piecewise_construct_t,
                 tuple<Args1...> x, tuple<Args2...> y);
template <class T1, class T2>
  void construct(pair<T1,T2>* p);
template <class T1, class T2, class U, class V>
  void construct(pair<T1,T2>* p, U&& x, V&& y);
template <class T1, class T2, class U, class V>
  void construct(pair<T1,T2>* p, const pair<U, V>& pr);
template <class T1, class T2, class U, class V>
  void construct(pair<T1,T2>* p, pair<U, V>&& pr);
```

by:

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

template <class T1, class T2, class... Args1, class... Args2>
  pair<T1,T2>* construct(pair<T1,T2>* p, piecewise_construct_t,
                         tuple<Args1...> x, tuple<Args2...> y);
template <class T1, class T2>
  pair<T1,T2>* construct(pair<T1,T2>* p);
template <class T1, class T2, class U, class V>
  pair<T1,T2>* construct(pair<T1,T2>* p, U&& x, V&& y);
template <class T1, class T2, class U, class V>
  pair<T1,T2>* construct(pair<T1,T2>* p, const pair<U, V>& pr);
template <class T1, class T2, class U, class V>
  pair<T1,T2>* construct(pair<T1,T2>* p, pair<U, V>&& pr);
```

In **23.12.3.2 polymorphic_allocator member functions [mem.poly.allocator.mem]**
in all definitions of construct() member functions

replace the return type

```
void
```

by the type of the passed argument p

```
T*  or  pair<T1,T2>*
```

and add a clause:

Returns: **p**

and in any ``*Effects:* Equivalent to:'' clause such as

```
construct(p, piecewise_construct,
          forward_as_tuple(pr.first),
          forward_as_tuple(pr.second));
```

add

```
return
```

to convert it to something like:

```
return construct(p, piecewise_construct,
                 forward_as_tuple(pr.first),
                 forward_as_tuple(pr.second));
```

In **23.13.1 Header <scoped_allocator> synopsis [allocator.adaptor.syn]**
replace

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

template <class T1, class T2, class... Args1, class... Args2>
  void construct(pair<T1, T2>* p, piecewise_construct_t,
                 tuple<Args1...> x, tuple<Args2...> y);
template <class T1, class T2>
  void construct(pair<T1, T2>* p);
template <class T1, class T2, class U, class V>
  void construct(pair<T1, T2>* p, U&& x, V&& y);
```

```
template <class T1, class T2, class U, class V>
  void construct(pair<T1, T2>* p, const pair<U, V>& x);
template <class T1, class T2, class U, class V>
  void construct(pair<T1, T2>* p, pair<U, V>&& x);
```

by:

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

template <class T1, class T2, class... Args1, class... Args2>
  pair<T1, T2>* construct(pair<T1, T2>* p, piecewise_construct_t,
                  tuple<Args1...> x, tuple<Args2...> y);
template <class T1, class T2>
  pair<T1, T2>* construct(pair<T1, T2>* p);
template <class T1, class T2, class U, class V>
  pair<T1, T2>* construct(pair<T1, T2>* p, U&& x, V&& y);
template <class T1, class T2, class U, class V>
  pair<T1, T2>* construct(pair<T1, T2>* p, const pair<U, V>& x);
template <class T1, class T2, class U, class V>
  pair<T1, T2>* construct(pair<T1, T2>* p, pair<U, V>&& x);
```

In **23.13.4 Scoped allocator adaptor members [allocator.adaptor.members]**
in all definitions of construct() member functions

replace the return type

void

by the type of the passed argument p

T* or pair<T1,T2>*

and add a clause:

Returns: **p**

and in any Effects clause calling

```
OUTERMOST_ALLOC_TRAITS(*this)::construct(OUTERMOST(*this), p, …)
```

add "p =" so that we get:

```
p = OUTERMOST_ALLOC_TRAITS(*this)::construct(OUTERMOST(*this), p, …)
```

and in any "*Effects:* Equivalent to:" clause such as

```
construct(p, piecewise_construct,
          forward_as_tuple(pr.first),
          forward_as_tuple(pr.second));
```

add in front:

```
return
```

to convert it to something like:

```
return construct(p, piecewise_construct,
                 forward_as_tuple(pr.first),
                 forward_as_tuple(pr.second));
```

In **26.2.1 General container requirements [container.requirements.general]**

extend any call of construct() such as:

- T is *DefaultInsertable into X* means that the following expression is well-formed:

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

by a leading p= such as:

- T is *DefaultInsertable into X* means that the following expression is well-formed:

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

Replace:

[*Note:* A container calls `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 ::new((void*)p) T(args), but specialized allocators may choose a different definition. —*end note*]

by

[*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 ::new((void*)p) T(args), but specialized allocators may choose a different definition. —*end note*]

In D.9 The default allocator [depr.default.allocator]
replace:

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

by:

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

and replace:

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

by:

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

6 *Effects*: As if by: `p = ::new((void *)p) U(std::forward<Args>(args)...);`

Returns: `p`

Acknowledgements

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