Project:	ISO JTC1/SC22/WG21: Programming Language C++
Doc No:	WG21 **P0532R0**
Date:	2017-01-14
Reply to:	Nicolai Josuttis (nico@josuttis.de)
Audience:	CWG, EWG, LEWG, LWG
Prev. Version:	----

On launder()

For C++17 we currently introduce std::launder() as a result of a NB comment for C++14:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3903.html#FI15

It was discussed as Core Issue 1776:

http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#1776

The first concrete proposal (with motivation for launder) was:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4303.html

The wording finally accepted was P0137R1:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0137r1.html

This issue also affects the following NB comments for C++17: JP20, CA12.

This paper first tries to explain exactly the problem std::launder() tries to solve (a problem that has existed for decades). It then also discusses the problems that in my opinion arise from this solution to motivate why in my opinion a different solution makes more sense.

To summarize, instead of introducing std::launder() I strongly recommend to make existing code that currently exists using placement new for elements with constant/reference member valid without any modification using std::launder().

Which Problem shall launder() solve?

According to the current standard, the following code results into undefined behavior:

```
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).

A corresponding example is part of the standard:

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]

Note that this behavior is not new. It exists since C++03.

As a consequence from the corresponding wording in the standard we'd have to ensure for all cases, where placement new is used for objects that might have constant or reference members, that the return value of placement new always is used with each access to the memory:

```
struct X {
  const int n;
};
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
```

How common is this undefined behavior currently? One way is to check whether the return value of placement new is used in generic code.
I only have one number about the Google code base thanks to Titus Winters:

> *Of the ~800 instances of placement new I can find, about 40% do not use the resulting pointer (and are thus presumably prone to the launder issue you discuss).*

Why using the return value of placement new is a problem

Unfortunately, in practice you cannot always easily use the return value of placement new. You might need additional objects and the current allocator interface does not support this.

Placement new into members

One example where using the return value causes overhead is when the storage is an existing member. That, for example would be the case for `std::optional` and `std::variant`.

Here is a simplified example:

```
template <typename T>
class coreoptional
{
  private:
    T payload;
  public:
    coreoptional(const T& t)
     : payload(t) {
    }
    template<typename... Args>
    void emplace(Args&&... args) {
      payload.~T();
```

```
          ::new (&payload) T(std::forward<Args>(args)...); // *
        }
        const T& operator*() const & {
          return payload;   // **
        }
    };
```

If here T is a structure with constant or reference members:

```
    struct X {
      const int _i;
      X(int i) : _i(i) {}
      friend std::ostream& operator<< (std::ostream& os, const X& x) {
        return os << x._i;
      }
    };
```

then the following code results into undefined behavior:

```
    coreoptional<X> optStr{42};
    optStr.emplace(77);
    std::cout << *optStr;     // undefined behavior (probably outputs 42 or 77)
```

The reason is that the output operation calls `operator*` (see ****** above), which uses the memory, where placement new placed a new value in, without using the return value (see ***** above).

In a class like this, you'd have to add an additional pointer member that keeps the return value of placement new and is used whenever the value is needed:

```
    template <typename T>
    class coreoptional
    {
      private:
        T payload;
        T* p;             // to be able to use the return value of placement new
      public:
        coreoptional(const T& t)
         : payload(t) {
            p = &payload;
        }
        template<typename... Args>
        void emplace(Args&&... args) {
          payload.~T();
          p = ::new (&payload) T(std::forward<Args>(args)...);
        }
        const T& operator*() const & {
          return *p;    // don't use payload here!
        }
    };
```

std::launder() was introduced to avoid this overhead (see below).

Placement new into allocated memory

Now you might say that this overhead of adding a pointer member is not a problem for (container) classes that allocate memory on the heap, because they anyway internally hold a pointer to the memory, which is used whenever access to the data is needed.
But, we run into a couple of different problems.

First, we can't use the return value of placement new here, when allocators are used, because currently the allocator interface provides no way to deal with the return value of placement new.

According to the allocator requirements (**17.5.3.5 Allocator requirements [allocator.requirements]):**

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)...)`

`a.construct(c,args)`has a return type `void`, so that no return value of placement new can be used.

Thus, currently (for C++ code up to version C++14) any container such as `std::vector` using elements with constant/reference members almost immediately runs into undefined behavior.
For example:

```
struct X {
  const int i;
  X(int _i) : i(_i) {}
  friend std::ostream& operator<< (std::ostream& os, const X& x) {
    return os << x.i;
  }
};

std::vector<X> v;
v.push_back(X(42));
v.clear();
v.push_back(X(77));
std::cout << v[0];   // undefined behavior
```

The reason is that `vector::push_back()` (as any other inserting function) calls placement new via allocator's `construct()`, which ignores the return value of placement new. And when we place new values into memory of objects with constant subobjects, according to '3.8 Object lifetime [basic.life] §8' we don't have the guarantee that the internally used objects holding the value refers to the new value.

In detail, the vector code might look as follows (with some simplifications, such as not using move semantics):

```
template <typename T, typename A = std::allocator<T>>
class vector
{
  public:
    typedef typename std::allocator_traits<A> ATR;
    typedef typename ATR::pointer pointer;
  private:
    A _alloc;         // current allocator
    pointer _elems;   // array of elements
    size_t _size;     // number of elements
    size_t _capa;     // capacity
  public:
    void push_back(const T& t) {
      if (_capa == _size) {
        reserve((_capa+1)*2);
      }
      ATR::construct(_alloc, _elems+_size, t);   // calls placement new
      ++_size;
    }
    T& operator[] (size_t i){
      return _elems[i];   // UB for replaced elements with constant members
    }
    …
};
```

Again, note that `ATR::construct()` does not return the return value of the called placement new. Thus, we can't use this return value instead of `_elems`.

Again `std::launder()` was proposed to solve this problem, but as I will describe below, we can't solve this problem with `std::launder()` at all.

However, note that before C++11, using elements with constant members either was not possible (vector) or formally not supported, because elements had to be CopyConstructible and Assignable (although node based containers such as lists worked perfectly fine with elements with const members). But with C++11, introducing move semantics, elements with constant members as class X above are supported and cause this undefined behavior.

How launder() tries to solve the problem

CWG decided to solve the problem by introducing std::launder() as follows (see core issue 1776 and the other links on top of this paper):

Whenever programs affect data where this problem might occur, you have to access the data via std::launder():

```
struct X {
  const int n;
};
X* p = new X{7};
new (p) X{42};                   // request to place a new value into p
int b = p->n;                    // undefined behavior
int c = std::launder(p)->n;      // OK, c is 42 (launder(p) forces to double-check)
int d = p->n;                    // still undefined behavior
```

Note that std::launder() does not "white wash" the pointer for any further usage. As the last line demonstrates, you have to use std::launder() at any time you access data where placement new was called for.

That means the currently (technically) broken code should be fixed now but modifying this code using std::launder(), whenever replacement memory with constant/reference elements is used.

There might be slightly different solutions, though. For example as Richard Smith pointed out:

> *Another option would be to do the laundering only when elements are inserted or removed (_M_begin = std::launder(_M_begin)) -- under the assumption that your compiler is smart enough to remove these stores when generating code.*

But in any case something must be fixed in the existing code of all these wrapper/container classes.

Why launder() doesn't work for allocator-based containers

However, for all allocator-based containers, such as std::vector, there is still a problem: std::launder() does not help to avoid undefined behavior.

Consider for example, what it would mean to use std::launder() in the simplified vector example above to access the data of possible replaced objects with constant elements:

```
template <typename T, typename A = std::allocator<T>>
class vector
{
  public:
    typedef typename std::allocator_traits<A> ATR;
    typedef typename ATR::pointer pointer;
  private:
    A _alloc;          // current allocator
    pointer _elems;    // array of elements
    size_t _size;      // number of elements
```

```
        size_t _capa;     // capacity
        …
    };
```

To fix the fact that `operator[]` results into undefined behavior, we'd have to apply std::launder() to the pointer referring to the replaced memory:

- The obvious solution might be the following code:
  ```
  T& operator[] (size_t i){
      return std::launder(_elems)[i];     // might still be UB
  }
  ```

  Unfortunately this might again result into undefined behavior, because the type of `_elems` might not be a raw pointer type. `std::allocator_traits<A>::pointer` might be a class type as Jonathan Wakely pointed out:
  std::launder takes a raw pointer as its argument. If the allocator's "pointer" type is not a raw pointer you can't pass it to launder.

- Well, we have another raw pointer, `this`.
  But:
  ```
  T& operator[] (size_t i){
      return std::launder(this)->_elems[i];     // launder() has no effect
  }
  ```
  has no effect because the type of `std::launder(this)` is equivalent to just `this` as Richard Smith pointed out:
  Remember that launder(p) is a no-op unless p points to an object whose lifetime has ended and where a new object has been created in the same storage.

So, as in general we don't have any raw pointer we can use for std::launder() here to get the desired effect.

Thus, std::launder() doesn't solve the problem to avoid undefined behavior in allocator-based containers when having elements with constant/reference members.

Why not fixing the Basic Lifetime Guarantees Instead?

The obvious question is, why don't we simply fix the current memory model so that using data where placement new was called for implicitly always does launder?

First note again: This problem is not new. The relevant wording was not part of C++98, but part of C++03 (that time 3.8 §7). So you can argue that for a long time, this problem at least formally exists.

Another question is about existing practice.

That is, it all depends on whether compilers use this kind of optimization:

Regarding the constant/reference optimizations, usually current compilers do not:

- AFAIK, gcc currently does not optimize the constant/reference case, but does optimizations for the vptr part.
- AFAIK, clang currently is working on some const-related optimizations so that these problems might now come into effect.
- Regarding the XL compilers, Hubert Tong wrote:

  The XL compilers also do optimize for the constant/reference case;
  however, it is meant to be limited to cases of static storage duration, e.g.,
  ```
  struct A {
    const int x;
    int y;
  };

  A a = { 42, 0 };
  ```

```
void foo(); // cannot see definition

int main(void) {
  foo();
  return a.x;
}
```

The read of a.x is replaced with 42. This has been the case since before 2004.

So, now we will be able to see the outcome of this core rules (in fact, I expect a lot of code to break then, so that in real life, things will probably be rolled back sooner or later; but we should avoid this unnecessary effort and chaos).

Is there more than the constant/reference member issue?

Another question is if we have similar problem beside the constant/reference issue.

In fact, Richard Smith stated in a private email:

> However, it's worth noting that the "dynamic type" provision is known to be *extremely* valuable. In particular, given:
>
> ```
> X *p;              // p is known to point to an object whose dynamic type is Y
> p->virtual_f();    // can devirtualize
> p->virtual_g();    // can devirtualize this, too
> ```
>
> ... we get *big* performance improvements from being able to devirtualize the second call, and that is only possible because we're permitted to assume that the first function did not change the dynamic type of the object via placement new.

Richard Smith also provided another example:

> ```
> struct B { virtual void f(); };
> struct D1 : B { virtual void f(); };
> struct D2 : B { virtual void f(); };
> variant<D1, D2> v;
> ```
>
> Here, variant needs to use something like launder internally if it wants to allow the active element to change between D1 and D2 and the virtual call to reliably call the right function (this is essentially the same as the p->virtual_f() / p->virtual_g() case above, once you inline away the variant implementation).

AFAIK, these cases can't (or can rarely) happen when using containers or wrapper types. So, it is probably valuable still to provide std::launder() for these cases, while fixing the const/reference case where std::launder() can't help.

Hiding the problem could make things worse

Note also that this problem might be caused indirectly by calling corresponding member functions.

Consider:

```
Obj x;
x.use();                    // ok
mutate(x);                  // does this call placement new?
x.use();                    // may have undefined behavior
std::launder(&x)->use();    // may be necessary
x.use();                    // may still have undefined behavior
```

As far you don't know that mutate() does not call placement new, you have to use std::launder() to be sure each time you use object `x`. But again, we don't have a raw pointer to apply std::launder() to (here also `std::launder(&x)` is the same as `&x` because `x` is still alive.

Nathan Myers recommend here:

> In that vein, maybe laundry is the wrong operation at the wrong
> time. What we really want is to declare that a particular use of
> a name (e.g. an argument) taints it. This would be attached
> to certain arguments of the functions, including member functions,
> that actually do the nasty work. E.g.
>
> ```
>   template <T> void default_construct(T taint* p) { new(p) T(); }
> ```
>
> (Probably the compiler should warn, then, if an argument not so
> declared is passed down to one that is.) For the case of vector
> push_back, only one vector member would be tainted. That member
> would best be identified as subject to tainting by member functions
> so declared. I.e., calling a tainting member function doesn't
> taint all of *this, but only taintable member vec<T>::ptr.
>
> In any case, the rvalue itself then would be laundered invisibly
> by the compiler at each place where it is needed. Compilers are
> better at this sort of thing than programmers are (and much
> better than people are). I believe "taint" would have some of
> the flavor of "restrict", albeit with important differences.)
> The case of vector<>::data() and vector<>::push_back() seems
> to need a similar relationship between the former's result and
> a subsequent call to the latter.
>
> Short of adding such a feature, limiting the latitude extended
> to implementations by the AFNOR change in 98->03 seems necessary
> -- possibly in an interim corrigendum to C++14 (and even C++11).
> Another intermediate measure might be to make launderation sticky
> (a.k.a. "whitewashing"), so that the third "o.use()" in the
> example above would be OK:
>
> ```
>   Obj o;
>   o.mod(); // o tainted
>   launder(&o);
>   o.use(); // ok
> ```
>
> But it would be harder to put this in a corrigendum, which seems
> to be needed regardless of what we do for C++17.

Summary and Recommendation

The problem described here uses a couple of programming techniques available for more than 20 years:

a) Placement new without always using the return value
   - because it's convenient
   - because it avoids introducing additional objects (as for variant or optional)
   - because there is no other option in all classes using allocators

b) Classes with constant members

c) Allocator-based containers

Although the not every combination was possible all the time (e.g. vectors could not use elements with constant members until C++11), I'd assume that there is a lot of code out there using placement new in combination with objects with constant members (before C++11 it e.g. could easily happen with node-based containers; since C++11 this is possible with any standard container).

If compilers turn a corresponding optimization on now, I'd assume that a lot of existing code will (silently) break. So, current implementation might cancel such a modification in practice. But I strongly suggest clarify the wording now to make this existing code valid. Standards should always standardize existing practice.

I don’t know, though, how much code is broken, if one of these optimization is no longer available. Regarding the constant/reference member part I don’t think that it is a lot of code because such an optimization seems not be widely available yet.

In practice, we can’t expect that all existing generic wrapper and container classes will be adopted accordingly (the problem not only applies to standard library classes).

In addition, as pointed out, the current solution with std::launder() does not solve the UB problem with allocator-based containers having elements with const/reference members at all.

So, the only valid option I see and recommend is to come back to the C++98 version of [basic.life] and allow only optimizations that we already allow for classes without const/reference members.

Acknowledgements

Thanks a lot to many people who gave me their free time so that I could understand and document the issue. If I understood something wrong, it’s my fault.

Thanks especially to Richard Smith, Hubert Tong, Jonathan Wakely, Ville Voutilainen, Nathan Myers, Titus Winters for their feedback and contribution.

Other References:

https://groups.google.com/a/isocpp.org/forum/#!msg/std-proposals/93ebFsxCjvQ/myxPG6o_9pkJ

https://accu.org/content/conf2012/JonathanWakely-CXX11_allocators.pdf