N4393 | Noop Constructors and Destructors

Pablo Halpern phalpern@halpernwightsoftware.com

2015-04-10

1 Abstract

This paper proposes a language feature for no-op constructor and destructor invocations, i.e., the ability to invoke a constructor or destructor without actually causing the state of memory to change. This feature allows a collection of correctly-configured bits to become an object, and an object to become a collection of bits without actually executing any code. Use cases and potential syntax for this feature are presented. This paper does not present precise wording, for which I would seek collaboration from someone in the Core Working Group if and when the Evolution Working Group approves the idea.

2 Motivation

The use case that inspired this feature was "destructive move", as proposed in N4158. That paper proposed a function template, `unitialized_destructive_move`, that destroyed one object and created another object, sometimes as a single, indivisible, operation, without invoking the objects' destructor or constructor. The proposal was worded in such a way as to give this template a special status in the core language, but it was considered inelegant that an object's lifetime would start at the completion of its constructor *and one other way.* This led me to search for a generalized way of getting the desired effect without making `destructive_move` special in the core language. In the processes, I considered other situations in my career where I had wanted the ability to skip a constructor or destructor invocation. For example, had this feature been available in C++0x, the piecewise constructor for `pair` might have been unnecessary.

3 Description of Proposed feature

The proposal is to use a special token sequence as a sort of "magic cookie" to invoke a constructor or destructor such that the invocation has no effect on the state of the program, but retains the quality of beginning or ending the lifetime of an object. To avoid beginning a premature bike-shed discussion on the syntax of this magic cookie, let's just call it `__COOKIE__`. Later in this paper, I list a number of combinations of tokens that could unambiguously work for `__COOKIE__`. The important thing is that, whatever `__COOKIE__` is, it is *not* an expression.

If `__COOKIE__` is passed as the sole argument to a constructor, then that constructor invocation has no effect on the state of the program. However, after the invocation, the lifetime of the object, and its base class and member subobjects, is deemed to have begun, just as if a real constructor had been invoked. Similarly, passing `__COOKIE__` to a destructor (yes, a destructor call with arguments!) would have no effect on the state of program, but the lifetime of the object, and its base class and member subobjects, is deemed to have ended.

A class cannot declare `__COOKIE__` constructors or destructors – they are automatically available in every type. An invocation of a `__COOKIE__` constructor or destructor does not require overload resolution, since no function is actually being called.

An invocation of a `__COOKIE__` constructor on an object of class type with virtual functions, virtual base classes, or subobjects with virtual functions or virtual base classes is ill formed. The reason for this restriction is described below in Future directions, Establishing compiler-managed invariants, along with a possible way to lift the restriction.

An invocation of a `__COOKIE__` constructor or destructor is valid before or after the invocation of a real constructor or destructor, respectively, and is idempotent with other, `__COOKIE__` invocations on the same object. ([basic.life] paragraph 4 already allows us to call constructors on live objects.) It is the responsibility of the caller to ensure that the bytes that make up an object constructed using the `__COOKIE__` constructor are valid; setting the bytes to a valid pattern can be done either before or after the noop constructor is invoked.

4 Use cases

4.1 Destructive Move

Destructive move is the motivating use case for the noop constructor and destructor feature. Assume an implementation of `std::list` that has a heap-allocated sentinel node. The move constructor is not `noexcept` because it must allocate a sentinel node for the moved-from list in order to avoid the "emptier than empty" condition. However the *destructive move* operation *can* be `noexcept` because there is no moved-from object left behind. The trick is to move the list without calling the move constructor. The implementation of `uninitialized_destructive_move` for such a list might look like this:

```
template <class T, class A>
void uninitialized_destructive_move(std::list<T,A> *from,
                                    std::list<T,A> *to) noexcept
{
    // Preconditions: `from` points to a valid list object;
    // `to` points to uninitialized memory.

    typedef std::list<T,A> list_t;

    // Bless the new list
    new (to) list_t(__COOKIE__);

    // Move data members over. Note that no new sentinel node is allocated.
    to->m_begin = from->m_begin;
    to->m_end  = from->m_end;
    new (&to->m_allocator) A(std::move(from->m_allocator));

    // Unbless the old list
    from->m_begin = from->m_end = nullptr_t;  // unnecessary, but safe
    from->~list_t(__COOKIE__);

    // Postconditions: `from` points to uninitialized memory;
    // `to` points to a valid list object
}
```

4.2 Trivial Destructive Move

If the allocator is trivially movable and trivially destructible, the destructive move operation above can be simplified to an invocation of `memcpy`:

```
template <class T, class A>
std::enable_if<is_trivially_destructive_movable_v<A>, void>
uninitialized_destructive_move(std::list<T,A> *from,
                               std::list<T,A> *to) noexcept
{
    typedef std::list<T,A> list_t;
    std::memcpy(to, from, sizeof(list_t));
    new (to) list_t(__COOKIE__);
    from->~list_t(__COOKIE__);
}
```

This idiom would work for the vast majority of value classes, even those that (like list) are not trivially movable and destructible. The idiom is generalized in N4158 for all "trivially destructive movable" types.

4.3 Swizzle to disk

A noop constructor is useful any time the bits that compose an object are arranged outside of the object's constructor. A carefully-designed data structure (containing no absolute pointers) can be written straight to disk and read back again:

```
// Type that uses relative pointers and is designed for storage on disk
class record { ... };

record *my_record = ...;
...
file.write(my_record, sizeof(record));
...

record *my_record2 = static_cast<record*>(operator new(sizeof(record)));
file.read(my_record2, sizeof(record));
new (my_record2) record(__COOKIE__);
```

Note that `record` cannot have virtual functions or virtual base classes. However, see Future directions, below, for a possible enhancement that would allow swizzling and `memcpy` of a broader range of types.

4.4 Choosing a constructor at run time

Sometimes it is necessary to choose a constructor at runtime, passing arguments of different types or different number of arguments depending on some condition:

```
struct Y {
    Y(float) noexcept;
    Y(int, float) noexcept;
    ...
};
```

```
struct X {
    Y m_y;
    ...
    X(float a, int b);
};

X::X(float a, int b) : m_y(__COOKIE__) {
    // Choose one of two constructors for m_y
    if (b > 0)
        new (&m_y) Y(b, a); // Invoke Y(int, float)
    else
        new (&m_y) Y(a);    // Invoke Y(float)
    ...
}
```

5 Dangers

It should be obvious to anybody reading this paper that both the no-op constructor and no-op destructor are extremely dangerous operations. One could easily write code that uses an object that has not been initialized or which has already been destroyed. If the no-op destructor is used on a variable with static or automatic lifetime, the real destructor will still be called.

The dangers of these constructs are not, however, any worse than the current dangers of calling a destructor manually (`x.~T()`), allocating an object using malloc without calling the constructor, or (deliberately) overwriting an object using `memcpy`. Just like these other programming techniques, proper application should typically be left to expert programmers writing reusable libraries (including standard library components such as `uninitialized_destructive_move`).

A related concern is that more care must be taken to ensure exception safety. Beginning an object's lifetime artificially by means of a no-op constructor does not change whether or not its destructor is invoked during exception unwinding. If the destructor runs, we risk attempting to destroy an incompletely-constructed object. If the destructor does not run, we risk leaking resources from the incompletely-constructed object. This problem not unlike that of a constructor, which must clean up a partially-constructed object if an exception is thrown. Indeed, the function that calls a no-op constructor is a pseudo-constructor and needs to ensure either that class invariants hold or that the object is unwound (possibly calling the no-op destructor) in the event of an exception or early return.

6 Alternatives for `__COOKIE__`

There are an infinite number of existing tokens or token sequences that could be used for `__COOKIE__` in the language definition. I will list a few here, and leave it up to your imagination to come up with better ones. In looking at the aesthetics of each example, think of it in the context of declaring a variable `x`. For example, if `__COOKIE__` were `=0`, then a variable declaration with a no-op constructor would look like `X x(=0)`.

Some possibilities are:

```
    =0
    =delete
    =void
    %
    /
    ^
    ;
    .
    ?
    <0>
    <void>
    ~ (requires look-ahead)
    ! (requires look-ahead)
    delete (requires look-ahead)
    =noop (context-sensitive keyword)
```

Humorous suggestion: A number of emoticons would also work. `:-)`

7 Alternatives considered

7.1 Special Case for `uninitialized_destructive_move`

N4158 had weasel words that allowed `uninitialized_destructive_move` to start and end the lifetimes of its arguments without articulating a language mechanism.

Pros:

- Library-only change for all known compilers. However, tools that track object lifetime would need to hook all specializations of `uninitialized_destructive_move`.

Cons:

- Special-case weasel words are not elegant.
- Does not support other use cases besides destructive move.

7.2 Function templates `bless`/`unbless`

Instead of adding a magic cookie, we could add two magic function templates:

```
namespace std {
    template <class T> T* bless(void* obj) noexcept;
    template <class T> void unbless(T* obj) noexcept;
}
```

The `bless` function would begin the lifetime of an object without invoking its constructor. The `unbless` function would end the lifetime of an object without invoking its destructor.

Pros:

- Trivial no-op functions with no compiler changes on most implementations. However, tools that track object lifetime would need to hook all specializations of `bless` and `unbless`.
- General-purpose functions can be used for use cases other than destructive move, such as swizzling from disk.

Cons:

- Changes the rule that an object's life begins at the end of its constructor invocation to a less elegant rule that an object's life begins at the end of its constructor invocation *or on return from std::bless*. A similar change would be needed for the destructor/`std::unbless`.
- Does not allow suppressing normal constructor invocation for automatic, static, and member variables and base class subobjects. Hence does not support use cases like the "Choosing a constructor at run time" use case, above.

7.3 Special version of `operator new`

It was suggested that the expression, `new(addr, __COOKIE__) T`, should have the effect of beginning the lifetime of the `T` object at `addr`. An alternative syntax would be `new(addr) __COOKIE__ T`.

Pros:

- Probably requires slightly fewer changes to the standard than the proposal presented in this paper.
- Supports the destructive move and swizzling use cases.
- The second version of the syntax could be extended to work with all overloads of `operator new`. For example `T* p = new __COOKIE__ T;` would allocate space appropriate for a `T` object but not construct it, exactly like `T* p = new T(__COOKIE__);` does in the current proposal.

Cons

- It is not clear now no-op destruction would be described. The `operator delete(void*, __COOKIE__)` corresponding to `operator new(size_t, __COOKIE__)` would normally be used only for exception handling (for an exception that could never occur).
- It is a bit strange to put this behavior on `operator new`, which is concerned primarily with memory allocation, not construction. Of course, the constructor is normally called the invocation of `operator new`, but construction details are normally in the constructor, not in `operator new`.
- Does not allow suppressing normal constructor invocation for automatic, static, and member variables and base class subobjects. Hence does not support use cases like the "Choosing a constructor at run time" use case, above.

8 Impact on the standard

This document does not contain formal wording, pending approval for the concept by the EWG. Formal wording will require collaboration with a member of the CWG. Nevertheless, we can expect wording changes will be necessary in the section on object lifetime [basic.life] and may be necessary in the sections for postfix expressions [expr.post], and unary expressions [expr.unary]. Although noop constructors and destructors are not user-defined, there is possibly some impact on the section for special member functions [special]. It is unlikely that there would be any impact on the wording around access to volatile objects [intro.execution], but only a core review would establish this for sure.

9 Future directions

9.1 Establishing compiler-managed invariants

During object construction, a program establishes two types of invariants: those that are managed by the author of the class, and those that are managed by the compiler. The latter category comprises the setting of vtbl pointers and virtual-base pointers. For a class that has no virtual functions and no virtual base classes, the class author can use a no-op constructor to construct a valid object by setting the member variables of the class (including base-class member variables) to valid values. This is not (in general) possible for classes that have virtual functions or virtual base classes because the job of establishing some of the invariants is given to the compiler. A possible enhancement, therefore, would be to have a the `__COOKIE__` constructor establish the compiler-managed invariants of the class (and those of its data member and base-class subobjects) while leaving the user-managed aspects of the class alone. If this enhancement were adopted, it would be desirable to use a different `__COOKIE__` so that the user who is expecting a true no-op doesn't get surprised.

9.2 Suppressing automatic destruction

Although the `__COOKIE__` constructor suppresses automatic constructor invocation for objects of any storage duration, this proposal does not provide a mechanism for suppressing the implicit destructor invocations for automatic, static, and member variables and base class subobjects. Such a feature might be desirable, especially for member and base class subobjects, which might need to be destroyed within the body of the class destructor, e.g., to obtain an unorthodox order of destruction. Local variables, too, might occasionally need to be destroyed in other than reverse order of construction, or it might be desirable to suppress destruction during exception unwinding in certain cases.

The ability to suppress automatic destruction could be seen as a natural extension of this proposal, but it would add significantly more core language changes and it is not central to the goal of providing a way to begin or end the lifetime of an object without invoking its constructor or destructor.

10 Implementation Concerns

We have no experience implementing this proposal. We do have experience faking it for some use cases (including `uninitialized_destructive_move`).

Nevertheless, it should be easy to implement in a compiler – just recognize the magic cookie and replace constructor or destructor invocation by a no-op. Tools that track object lifetime would treat the special calls like normal constructor or destructor calls with implicit construction or destruction of subobjects.

11 Acknowledgments

Thanks to Alisdair Merideth, Mike Henry Verschell, John Lakos, Mike Giroux, and Hyman Rosen for reviewing drafts of this paper and helping me clarify some concerns.

12 References

N4158 *Destructive Move*, Pablo Halpern, 2014-10-12