Document	P2014R0
Date	2020-01-12
Reply To	Lewis Baker <lbaker@fb.com>, Gor Nishanov <gorn@microsoft.com>
Audience	Evolution
Target	C++20

Proposed resolution for US061+US062 - aligned allocation of coroutine frames

Evolution reviewed NB comments US061 and US062 in Belfast 2019 and voted to consider resolving these NB comments for C++20. However, the proposed resolutions in the comments were not sufficiently detailed to be able to put forward as a resolution and so it was requested that a paper be written detailing a proposed resolution.

This paper contains some context on the NB comments, proposed wording for two design alternatives and some design discussion about some of the more subtle aspects of the proposed resolution.

Option 1 for the proposed wording attempts to maintain behaviour consistent with [expr.new] and [expr.delete] with regards to only preferring `std::align_val_t` overloads when the storage to be allocated has new-extended alignment. This results in a more complicated set of rules for lookup, including needing to perform two lookups and then delaying the choice of which one to call until later in the compilation phases when the coroutine-frame layout requirements are known.

Option 2 for the proposed wording instead always prefers to call the std::align_val_t overloads of allocation/deallocation functions regardless of whether the storage required for the coroutine state has new-extended alignment or not. This design is simpler but is inconsistent with the behaviour of [expr.new] and [expr.delete].

Both options are put forward for consideration by the Evolution Working Group.

US061

Coroutine allocation does not consider `std::align_val_t` overloads introduced in C++17

Proposed change:
Add them to the sequence of operator new calls that are attempted using wording similar to 7.6.2.7/18

US062

The construction of the argument list for the call to the allocation function to allocate the 'coroutine state' does not call the overload of `operator new()` that accepts a `std::align_val_t` in the case that the allocation required for the coroutine has 'new-extended alignment'. This means that allocations of coroutine frames may not be correctly aligned in cases where the coroutine state contains overaligned types.

Proposed change:
Apply similar wording from [expr.new]p18:
Insert "If the coroutine state has new-extended alignment then the next argument is std::align_val_t." after "has type size_t."

Insert at end of paragraph:
If no matching function is found and the allocated coroutine state has new-extended alignment, the alignment argument is removed from the argument list, and overload resolution is performed again.

Current wording (N4835):

[dcl.fct.def.coroutine] p9

An implementation may need to allocate additional storage for a coroutine. This storage is known as the coroutine state and is obtained by calling a non-array allocation function (6.7.5.4.1). The allocation function's name is looked up in the scope of the promise type. If this lookup fails, the allocation function's name is looked up in the global scope. If the lookup finds an allocation function in the scope of the promise type, overload resolution is performed on a function call created by assembling an argument list. The first argument is the amount of space requested, and has type std::size_t. The lvalues p1 . . . pn are the succeeding arguments. If no viable function is found (12.4.2), overload resolution is performed again on a function call created by passing just the amount of space required as an argument of type std::size_t.

[dcl.fct.def.coroutine] p12

The deallocation function's name is looked up in the scope of the promise type. If this lookup fails, the deallocation function's name is looked up in the global scope. If deallocation function lookup finds both a usual deallocation function with only a pointer parameter and a usual deallocation function with both a pointer parameter and a size parameter, then the selected deallocation function shall be the one with two parameters. Otherwise, the selected deallocation function shall be the function with one parameter. If no usual deallocation function is found, the program is ill-formed. The selected deallocation function shall be called with the address of the

block of storage to be reclaimed as its first argument. If a deallocation function with a parameter of type std::size_t is used, the size of the block is passed as the corresponding argument.

Reference (other related sections)

[expr.new] p18

Overload resolution is performed on a function call created by assembling an argument list. The first argument is the amount of space requested, and has type std::size_t. If the type of the allocated object has new-extended alignment, the next argument is the type's alignment, and has type std::align_val_t. If the new-placement syntax is used, the initializer-clauses in its expression-list are the succeeding arguments. If no matching function is found and the allocated object type has new-extended alignment, the alignment argument is removed from the argument list, and overload resolution is performed again.

[basic.stc.dynamic.deallocation] p3

Each deallocation function shall return void. If the function is a destroying operator delete declared in class type C, the type of its first parameter shall be C*; otherwise, the type of its first parameter shall be void*. A deallocation function may have more than one parameter. A usual deallocation function is a deallocation function whose parameters after the first are

- optionally, a parameter of type std::destroying_delete_t, then
- optionally, a parameter of type std::size_t, then
- optionally, a parameter of type std::align_val_t.

A destroying operator delete shall be a usual deallocation function. A deallocation function may be an instance of a function template. Neither the first parameter nor the return type shall depend on a template parameter. A deallocation function template shall have two or more function parameters. A template instance is never a usual deallocation function, regardless of its signature.

[expr.delete] p10

If deallocation function lookup finds more than one usual deallocation function, the function to be called is selected as follows:

- If any of the deallocation functions is a destroying operator delete, all deallocation functions that are not destroying operator deletes are eliminated from further consideration.
- If the type has new-extended alignment, a function with a parameter of type std::align_val_t is preferred; otherwise a function without such a parameter is preferred. If any preferred functions are found, all non-preferred functions are eliminated from further consideration.

- If exactly one function remains, that function is selected and the selection process terminates.
- If the deallocation functions have class scope, the one without a parameter of type std::size_t is selected.
- If the type is complete and if, for an array delete expression only, the operand is a pointer to a class type with a non-trivial destructor or a (possibly multi-dimensional) array thereof, the function with a parameter of type std::size_t is selected.
- Otherwise, it is unspecified whether a deallocation function with a parameter of type std::size_t is selected.

Proposed wording (Option 1):

Modify [dcl.fct.def.coroutine] p9 as follows:

> An implementation may need to allocate additional storage for a coroutine. This storage is known as the coroutine state and is obtained by calling a non-array allocation function (6.7.5.4.1). The allocation function's name is looked up in the scope of the promise type. If the lookup finds an allocation function in the scope of the promise type, then;
>
> - A first overload resolution is performed on a function call created by assembling an argument list. The first argument is the amount of space requested and has type `std::size_t`. The second argument is the coroutine state's alignment and has type `std::align_val_t`. The lvalues p1 ... pn are the succeeding arguments. If a viable function is found (12.4.2), and the type of the second parameter is `std::align_val_t` and is not a dependent type then then let the found overload be the *overaligned-allocation-function*, otherwise overload resolution is performed again on a function call created by passing just the first two arguments. If a viable function is found (12.4.2), and the type of the second parameter is `std::align_val_t` and is not a dependent type then let the found overload be the *overaligned-allocation-function*.
> - A second overload resolution is performed on a function call created by assembling an argument list. The first argument is the amount of space requested and has type `std::size_t`. The lvalues p1 ... pn are the succeeding arguments. If ~~no~~a viable function is found (12.4.2), then let the found overload be the *normal-allocation-function*, otherwise overload resolution is performed again on a function call created by passing just the amount of space required as an argument of type `std::size_t`. If a viable function is found (12.4.2), then let the found overload by the *normal-allocation-function*.
>
> Otherwise;
>
> - Let the *overaligned-allocation-function* be `::operator new(std::size_t, std::align_val_t)` and let the *normal-allocation-function* be `::operator new(std::size_t)`

If the coroutine state has *new-extended alignment* and an *overaligned-allocation-function* was found then the coroutine state is allocated by a call to the *overaligned-allocation-function*. Otherwise, if the *normal-allocation-function* was found then the coroutine state is allocated by a call to the *normal-allocation-function*. Otherwise, the program is ill-formed.

Modify [dcl.fct.def.coroutine] p12 as follows:

> The deallocation function's name is looked up in the scope of the promise type. If this lookup fails, the deallocation function's name is looked up in the global scope. If deallocation

function lookup finds more than one usual deallocation function, the function to be called is selected as follows:

- If any deallocation functions found in the scope of the promise type are a destroying operator delete, the program is ill-formed.
- If the coroutine state has new-extended alignment, a function with a parameter of type `std::align_val_t` is preferred; otherwise a function without such a parameter is preferred. If any preferrred functions are found, all non-preferred functions are eliminated from further consideration.
- If exactly one function remains, that function is selected and the selection process terminates.
- Otherwise, a function with a parameter of type `std::size_t` is preferred to a function without a parameter of type `std::size_t`.

~~both a usual deallocation function with only a pointer parameter and a usual deallocation function with both a pointer parameter and a size parameter, then the selected deallocation function shall be the one with two parameters. Otherwise, the selected deallocation function shall be the function with one parameter.~~ If no usual deallocation function is found, the program is ill-formed. The selected deallocation function shall be called with the address of the block of storage to be reclaimed as its first argument. If a deallocation function with a parameter of type `std::size_t` is used, the size of the block is passed as the corresponding argument. If a deallocation function with a parameter of type `std::align_val_t` is used, the requested alignment of the block is passed as the corresponding argument.

Proposed Wording (Option 2)

The design of this option does not attempt to maintain consistent behaviour with [expr.new] and [expr.delete] with respect to only calling std::align_val_t overloads in cases where the allocation has new-extended alignment. Instead it opts to always prefer calling the std::align_val_t overload of the allocation and deallocation functions in over calling an overload without the std::align_val_t parameter.

Modify [dcl.fct.def.coroutine] p9 as follows:

An implementation may need to allocate additional storage for a coroutine. This storage is known as the coroutine state and is obtained by calling a non-array allocation function (6.7.5.4.1). The allocation function's name is looked up in the scope of the promise type. If this lookup fails, the allocation function's name is looked up in the global scope. If the lookup finds an allocation function in the scope of the promise type, overload resolution is performed on a function call created by assembling an argument list. The first argument is the amount of space requested, and has type std::size_t. The second argument is the requested alignment and has type `std::align_val_t`. The lvalues p1 . . . pn are the succeeding arguments. If ~~no~~a viable function is found (12.4.2), and the type of the second

parameter is `std::align_val_t` and is not a dependent type then this overload is selected. Otherwise, overload resolution is performed again on a function call created by passing the amount of space requested as an argument of type `std::size_t` as the first argument, and the requested alignment as an argument of type `std::align_val_t` as the second argument. If a viable function is found (12.4.2) and the type of the second parameter is `std::align_val_t` and is not a dependent type then this overload is selected. Otherwise, overload resolution is performed again on a function call created by passing the amount of space requested as an argument of type `std::size_t` as the first argument, and the lvalues p1 ... pn as the succeeding arguments. If a viable function is found (12.4.2) then this overload is selected. Otherwise, overload resolution is performed again on a function call created by passing just the amount of space required as an argument of type std::size_t.
If the lookup did not find an allocation function in the scope of the promise type then storage is allocated by calling `::operator new(std::size_t, std::align_val_t)` passing the amount of space requested as an argument of type `std::size_t` as the first argument and passing the requested alignment as an argument of type `std::align_val_t` as the second argument.

Modify [dcl.fct.def.coroutine] p12 as follows:

The deallocation function's name is looked up in the scope of the promise type. If this lookup fails, the deallocation function's name is looked up in the global scope. If deallocation function lookup finds a usual deallocation function with a pointer parameter, size parameter and alignment parameter then this will be the selected deallocation function, otherwise if lookup finds ~~both a usual deallocation function with only a pointer parameter and~~ a usual deallocation function with both a pointer parameter and a size parameter, then ~~the~~this will be the selected deallocation function. Otherwise, if lookup finds a usual deallocation function with only a pointer parameter, then this will be ~~shall be the one with two parameters. Otherwise,~~ the selected deallocation function ~~shall be the function with one parameter~~. If no usual deallocation function is found, the program is ill-formed. The selected deallocation function shall be called with the address of the block of storage to be reclaimed as its first argument. If a deallocation function with a parameter of type std::size_t is used, the size of the block is passed as the corresponding argument. If a deallocation function with a parameter of type std::align_val_t is used, then if the block was allocated by a call that included a requested alignment parameter, then the requested alignment is passed as the corresponding argument, otherwise __STDCPP_DEFAULT_NEW_ALIGNMENT__ is passed as the corresponding argument.

Discussion

Issue #1 - Potential ambiguity with interpretation of templated operator new() overload

[basic.stc.dynamic.allocation]/3.1
If the allocation function takes an argument of type std::align_val_t, the storage will have the alignment specified by the value of this argument.

This paragraph has some potentially interesting interactions with coroutine frame allocation which allows an overload of `promise_type::operator new()` to be defined that will receive the arguments to the coroutine function.

This means that if I have:

```
struct task {
  struct promise_type {
    template<typename... Args>
    static void* operator new(std::size_t, Args...);

    ...
  };
};

task foo(std::align_val_t x) {
  co_return;
}

foo(std::align_val_t(1'000'000));
```

Then the compiler will generate a call to `promise_type::operator new(std::size_t, std::align_val_t)` if needed to allocate storage for the coroutine frame.

Does this mean that the implementation of this `operator new()` needs to ensure that the allocation is required to have an alignment of 1'000'000 bytes? Will compilers make any assumptions about this? ie. if the align_val_t argument was not injected by the compiler?

The proposed wording above tries to avoid this case being considered a valid overload for the `std::align_val_t` case by requiring that the found overload not have a parameter in the alignment position that is a dependent-type.

Issue #2 - Destroying operator delete does not make sense

Need to exclude destroying operator delete from being considered for operator delete overloads found within the scope of the promise type.

When destroying the coroutine state we are not actually destroying an object. The coroutine state has no type and so we cannot pass a pointer to the coroutine state type as required by destroying operator delete calls.

Issue #3 - Alignment requirements not known by compiler front-end

The alignment and size of the coroutine state is not known at template instantiation time and can depend on the result of optimisation passes that run in later compilation phases.

Optimisation passes of the compiler may end up eliding storage of some overaligned local-variables if it determines that the construction of those variables is never reachable (ie. dead-code), thus potentially reducing the alignment requirements of the coroutine state compared to what an analysis by a compiler-front end could determine.

Conversely, it's possible that the compiler might inline the allocation of a nested coroutine frame into the caller, which itself may contain overaligned variables, thus potentially increasing the alignment requirement of the coroutine state compared to what an analysis by a compiler-front end could determine.

This means that the choice of whether to call the new-extended alignment allocation function or the normal allocation function will typically be made by the compiler middle/back-end once the coroutine frame layout has been determined, long after template instantiation has completed.

Thus a compiler front-end will typically need to perform lookup, and instantiate if necessary, both flavours of allocation functions, and if both are found, defer the decision about which one to call until after the coroutine frame layout has been calculated.

The proposed wording above attempts to describe this process of performing two allocation function overload resolutions, one for new-extended alignment and one for normal alignment.

While, in some cases, it may be possible for the compiler front-end to determine which one will be called, there is no obvious way to normatively specify when this should occur. Thus even in this situation the compiler should be forced to instantiate both allocation function flavours so that the validity of the program is not determined by some unspecified property of the coroutine being compiled. eg. one of the allocation functions may have been ill-formed were it to be

instantiated.

If a particular compiler skipped instantiating the allocation function in cases where it could determine that it was never going to be called then this might mean the difference between a program being valid on one compiler and one being ill-formed on another compiler that did not have this ability to skip the instantiation.

Issue #4 - Order of resolution (Option 1)

When resolving the overloads of `operator new()` and `operator delete()` to call when allocating/deallocating a coroutine frame, the overload resolution order is different depending on whether the coroutine frame is overaligned or not.

This is how the current wording of option 1 is intended to be interpreted with regards to order of preference of different overloads.

If the coroutine frame has new-extended alignment then the order of preference for overload resolution of the allocation function is:

- `promise_type::operator new(std::size_t, std::align_val_t, Args...)`
- `promise_type::operator new(std::size_t, std::align_val_t)`
- `promise_type::operator new(std::size_t, Args...)`
- `promise_type::operator new(std::size_t)`
- `::operator new(std::size_t, std::align_val_t)`

and the order of resolution of the deallocation function is:

- `promise_type::operator delete(void*, std::size_t, std::align_val_t)`
- `promise_type::operator delete(void*, std::align_val_t)`
- `promise_type::operator delete(void*, std::size_t)`
- `promise_type::operator delete(void*)`
- `::operator delete(void*, std::size_t, std::align_val_t)`

Otherwise, if the coroutine frame has normal (non-extended) alignment then the order of preference for overload resolution is:

- `promise_type::operator new(std::size_t, Args...)`
- `promise_type::operator new(std::size_t)`
- `::operator new(std::size_t)`

and the order of resolution of the deallocation function is:

- `promise_type::operator delete(void*, std::size_t)`
- `promise_type::operator delete(void*)`
- `promise_type::operator delete(void*, std::size_t, std::align_val_t)`
- `promise_type::operator delete(void*, std::align_val_t)`
- `::operator delete(void*, std::size_t)`

Note that there is a slight inconsistency between `operator new()` and `operator delete()` here. If the coroutine frame is not overaligned then the overloads of `operator`

`new()` that contain `std::align_val_t` are not considered. However, for deallocation function, it still considers `std::align_val_t` overloads, it just prefers the overloads without `std::align_val_t`.

Also, there is not currently a requirement to also provide a class-member deallocation function if you provide a class-member allocation function. It will fall-back to global `operator delete()` if you only provide a `promise_type::operator new()`.

Similarly, if you provide only a `promise_type::operator delete()`, it will still fall back to `::operator new()` for the allocation-function.

This seems to be consistent with the behaviour of ordinary class allocation functions, however.