Remove Default Candidate Executor

Document Number: P2161R2
Date: 2020-07-14
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: LEWG

Abstract

This paper proposes that `associated_executor` not provide a default candidate type.

Background

The Networking TS [1] introduces “associators:” Binary class templates whose arguments are the “source type” and the “candidate type” (respectively) (§13.2.7.8 [async.reqmts.associator]). A default (the “default candidate type”) is required to be provided for the “candidate type” (i.e. an associator must be usable as if it were a unary class template).

The purpose of an associator is to obtain an instance of an “associated object” based on a “source object” (an instance of the source type) and optionally a “candidate object” (an instance of the candidate type). The type of the associated object (i.e. the “associated type”) is available through the `type` member type alias and the actual computation of the associated object may be performed via the `get` static member function. This member function must be invocable as if it were a unary or binary function (in the unary case only the source object is accepted whereas the binary case accepts both the source object and candidate object).

There are two associators provided by the Networking TS: `associated_executor` (§13.12 [async.assoc.exec]) and `associated_allocator` (§13.5 [async.assoc.alloc]) which obtain objects whose types satisfy the `Executor` (§13.2.2 [async.reqmts.executor]) and `ProtoAllocator` (§13.2.1 [async.reqmts.proto.allocator]) named type requirements (respectively). They have default candidate types `system_executor` and `allocator<void>` (respectively).

P2149R0 [2] was written in response to discussion in Prague 2020 SG4 which brought the design and usability of `system_executor` into question. P2149R0 proposed a two pronged solution:

- Add `inline_executor` to replace `system_executor` as the default candidate type for `associated_executor` and
- Remove `system_executor`

Subsequent discussion of P2149R0 on the reflector made it obvious that this two pronged approach was coupling two separable questions:

- Should `system_executor` be the default candidate type for `associated_executor`?
- Should `system_executor` exist at all?

This paper provides a vehicle to consider the former question with subsequent revisions of P2149 being a vehicle to consider the latter.

Motivation

In general the presence of a default implies that there is both:

- A choice to be made and
- A certain choice (the default) which is likely to be correct

In choosing either a model of `Executor` or `ProtoAllocator` it is clear that the former of these is satisfied. If this was not the case either:

- Associators for these named type requirements would not exist or
- Those associators would be completely unused

Neither of which is the case.

When applied to the selection of a module of `ProtoAllocator` there is a strong argument to be made that the latter implication is satisfied. Overwhelmingly users do not choose models of `Allocator` other than `allocator<T>`. This strongly indicates that `allocator<void>` is likely to be the correct choice whenever someone would be faced with a choice of `ProtoAllocator`. The standard library already reifies this by defaulting `Allocator` template parameters to `allocator<T>` seemingly at every turn.

The argument for a default `Executor` is much weaker. In the formulation above it is supposed that a default must be “likely to be correct.” There are two meanings of “correct” which we should consider:

- What the user would choose anyway
- Having properties such that it is an acceptable choice notwithstanding

Based on the author’s experience `io_context::executor_type` is much more likely to be used than `system_executor`. However much current experience (including most of the author’s) with the Networking TS (and Asio) is from a time before P1322 [3].

Post-P1322 a persuasive argument could be made that `system_executor` is what the user chooses in most cases. Discussion in SG4 in Prague 2020 and on the reflector indicates that the raison d'être of `system_executor` is to encapsulate the operating system’s global thread pool. This global thread pool may have access to information and functionality the user does not which may make it optimal in most situations.

This is where the second meaning of “correct” becomes relevant. Even if users would overwhelmingly choose `system_executor` there are still users who would not and if `system_executor` has properties which make it a surprising, bug prone default it is ill fitted to that role.

`system_executor` has several such properties.

`system_context` is permitted to execute any number of submitted work items in parallel. Users may have strict parallelism requirements enforced by their choice of executor (e.g. the underlying execution context forms an “implicit strand” [4]). Silently falling back to `system_executor` introduces data races (i.e. undefined behavior) in such situations.

`system_context` makes progress on work items in the background detached from any user controlled thread. The user’s chosen executor on the other hand may have an underlying execution context which permits the user to control precisely when work is and is not executing (e.g. the “run functions” of `io_context` (§14.2 [io_context.io_context])). If work is silently submitted via `system_executor` then work items may be making progress when the user reasonably believes no such thing can occur. This is another source of accidental data races (i.e. undefined behavior).

`system_context` may arbitrarily extend the lifetime of submitted work items and associated services. While the Networking TS provides a way to ensure that the Networking TS no longer makes forward progress on work items (`system_context::stop` and `system_context::join` (§13.19 [async.system.context])) there is no way to ensure that the lifetimes of all submitted work items and associated services have ended. By contrast a user may intend to use an executor whose underlying execution context allows them to precisely control when the lifetime of work items and services shall end (e.g. `io_context` by way of the `ExecutionContext` named type requirement (§13.2.3 [async.reqmts.executioncontext]) (in the case of work items) and deriving from `execution_context` (§13.7.1 [async.exec.ctx.dtor]) (in the case of services)). Inadvertently submitting work items to `system_executor` may therefore lead to all manner of lifetime bugs (i.e. undefined behavior).

Notably each of these properties stems from the fact that the singleton instance of `execution_context` is as such a global variable.

P2149R0 proposes `inline_executor` as a default candidate type for `associated_executor`. However as it lacks a stateful execution context instances of this type are unable to provide a satisfactory implementation of `post` and its implementation thereof simply throws an exception. There’s no reason to move what is logically a programming mistake (not concretely specifying where you want code to execute) to runtime.

In trying to synthesize an executor analogue of `std::allocator<void>` the Networking TS has encountered a problem: Memory and execution agents [5] are both resources which programs must manage but there's a fundamental difference between the two making the former amenable to a default, global implementation but not the latter: Memory is static and does not perform actions independent being acted upon.

Unary `defer`, `dispatch`, & `post`

The majority of the Networking TS does not make use of the default candidate executor. Instead the Networking TS largely specifies that the "I/O executor" (§13.2.7.8 [async.reqmts.async.io.exec]) be provided as a candidate object where an executor is obtained via `associated_executor`. There are three exceptions to this, the three unary overloads of:

- `defer` (§13.24 [async.defer])
- `dispatch` (§13.22 [async.dispatch])
- `post` (§13.23 [async.post])

Each of which is an initiating function (§13.2.7 [async.reqmts.async]) which accepts a completion token (§13.2.7.2 [async.reqmts.async.token]) whose completion handler has a signature of `void()`. Since they accept only a completion token there's no clear candidate object and indeed they are specified not to provide one.

Given what the Networking TS means by "defer" and "dispatch" it is clear that unary `defer` and `dispatch` could be implemented without requiring a handle to an external executor. Since simply invoking the completion handler synthesized from the provided completion token directly within `defer` or `dispatch` does not violate their contract they could be specified to provide an executor which executes all work "inline" as a candidate object. While this approach is possible and does not violate the contracts of those functions it may still be surprising. Users may be expecting a certain associated executor to be chosen, may have associated it incorrectly, and may be surprised that this compile time bug is lifted silently into a runtime bug.

Moreover `post` clearly cannot be implemented in this manner. Due to its contract something beyond an "inline" executor is required.

Removing these overloads completely is the most obvious approach. They are used by the Networking TS itself and therefore the impact would be minimal. However it seems heavy-handed to remove this functionality for all types given that there are many completion handler types which can plausibly benefit from it.

Revision 0 of this paper proposed solving this by specifying `defer`, `dispatch`, and `post` to provide a synthetic candidate object (i.e. an implementation-defined type synthesized solely for this purpose) and then checking at compile time if the type of the associated executor was that type. If this was the case it would indicate that the completion handler did not provide an associated executor and the initiating function would fail to compile.

During review in SG4 (teleconference on May 14, 2020) it was pointed out that while this technique may work in some (or even most) situations it would not work in all situations. There may nominally exist an association for a certain completion handler type however it may simply wrap the candidate object. This does not fulfill the expectation that the association provide a bona fide executor but does obfuscate the implementation-defined type such that the check proposed by revision 0 fails to render this situation ill-formed.

Also discussed during SG4 was providing a truly unary version of `get_associated_executor`. However this would constitute a large change of the associator machinery (turning `get_associated_executor` and `get_associated_allocator` into customization point objects was also discussed) and is beyond the narrow scope of this paper.

The entirety of this issue is caused by the perspective that unary `defer`, `dispatch`, and `post` accept only a completion token and therefore have no “I/O executor” to use as a candidate object. If we shift our perspective we could regard the completion handler (synthesized from the competition token) as an “I/O object” from which must be able to obtain an executor by making a nullary call to the member function `get_executor`. This would be in line with the mechanism employed by the unspecialized version of `associated_executor` (§13.12.1 [async.assoc.exec.members]): If the source type provides a member type alias `executor_type` then an executor is obtained by nullary-invoking the member function `get_executor`.

Proposed Changes

Associator

§13.2.7.8/2-5 [async.reqmts.associator]:

An associator shall be a class template that takes two template type arguments. The first template argument is the source type `S`. The second template argument is the candidate type `C`. ~~*The second template argument shall be defaulted to some default candidate type `D` that satisfies the type requirements `R`.*~~

An associator shall additionally satisfy the requirements in Table 6. In this table, `X` is a class template that meets the associator requirements, `S` is the source type, `s` is a value of type `S` or `const S`, `C` is the candidate type, and `c` is a (possibly const) value of type `C`~~*, `D` is the default candidate type, and `d` is a (possibly const) value of type `D` that is the default candidate object*~~*.*

[...]

Finally, the associator shall provide the following type alias and function template in the enclosing namespace:

```
template<class S, class C = D> using X_t = typename X <S, C>::type;

template<class S, class C = D>
typename X <S, C>::type get_X (const S& s, const C& c = d)
{
    return X <S, C>::get(s, c);
}
```

where X is replaced with the name of the associator class template.

The first and third rows must be stricken from table 6.

associated_executor

§13.1 [async.synop]:

[...]

```
template<class T, class Executor = system_executor>
struct associated_executor;
```

[...]

§13.12/1 [async.assoc.exec]:

Class template `associated_executor` is an associator for the `Executor` type requirements~~, with default candidate type `system_executor` and default candidate object `system_executor()`~~.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

    template<class T, class Executor = system_executor>
    struct associated_executor {
        using type = see below;
        static type get(const T& t, const Executor& e = Executor()) noexcept;
    };

} // inline namespace v1
} // namespace net
```

} // namespace experimental
} // namespace std

The second row must be stricken from table 9.

§13.12/2 [async.assoc.exec.members]:

type get(const T& t, const Executor& e~~ = Executor()~~) noexcept;

[...]

get_associated_executor

§13.1 [async.synop]:

[...]

~~*template<class T>*~~
~~*associated_executor_t<T> get_associated_executor(const T& t) noexcept;*~~

[...]

§13.13/1 [async.assoc.exec.get]:

~~*template<class T>*~~
~~*associated_executor_t<T> get_associated_executor(const T& t) noexcept;*~~

~~*Returns: associated_executor::get(t).*~~

associated_executor_t

§13.1 [async.synop]:

[...]

template<class T, class Executor~~ = system_executor~~>
using associated_executor_t = typename associated_executor::type;

[...]

make_work_guard

§13.1 [async.synop]:

[...]

~~template<class T>~~
~~executor_work_guard<associated_executor_t<T>> make_work_guard(const T& t);~~

[...]

§13.17/5-6 [async.make.work.guard]:

~~template<class T>~~
~~executor_work_guard<associated_executor_t<T>> make_work_guard(const T& t);~~

~~*Returns:* make_work_guard(get_associated_executor(t)).~~

~~*Remarks: This function shall not participate in overload resolution unless* is_executor_v<T> *is* false *and* is_convertible<T&, execution_context&>::value *is* false.~~

dispatch, post, & defer

Insert the following before §13.22/3 [async.dispatch], §13.23/3 [async.post], and §13.24/3 [async.defer]:

Requires: If typename async_completion<CompletionToken, void()>::completion_handler_type::executor_type *is invalid, does not denote a type, or is not the same type as* decltype(declval<const async_completion<CompletionToken, void()>::completion_handler_type&>().get_executor()) *the program is ill-formed.*

§13.22/3.2 [async.dispatch], §13.23/3.2 [async.post], and §13.24/3.2 [async.defer]:

- *Performs* ex.*[...]*(std::move(completion.completion_handler), alloc)*, where* ex *is the result of* ~~get_associated_executor(~~completion.completion_handler~~, e)~~.get_executor()*, and* alloc *is the result of* get_associated_allocator(completion.completion_handler).

§13.22/6 [async.dispatch], §13.23/6 [async.post], and §13.24/6 [async.defer]:

Effects:

- *Constructs an object* completion *of type* async_completion<CompletionToken, void()>*, initialized with* token.
- *If* associated_executor_t<typename async_completion<CompletionToken, void()>::completion_handler_type, Executor> *and* Executor *are the same type, evaluates the expression* get_associated_executor(completion.completion_handler, ex) == ex*, and if* true *then let* f *denote* completion.completion_handler*, otherwise* ~~c~~*Constructs a function object* f *containing as members:*
    - *a copy of the completion handler h, initialized with* std::move(completion.completion_handler)*,*

- *an `executor_work_guard` object `w` for the completion handler's associated executor, initialized with `make_work_guard(h, ex)`*

and where the effect of `f()` is:

- *`w.get_executor().dispatch(std::move(h), alloc)`, where `alloc` is the result of `get_associated_allocator(h)`, followed by*
- *`w.reset()`.*

- *Performs `ex.[...](std::move(f), alloc)`, where `alloc` is the result of `get_associated_allocator(completion.completion_handler)` ~~prior to the construction of `f`~~ immediately after the construction of `completion`.*

Implementations

Chris Kohlhoff has implemented revision 0 [6] and revision 1 [7] of this paper against "standalone" Asio.

Acknowledgements

The author would like to thank Chris Kohlhoff for his assistance in exploring this design space and preparing this paper.

Revision History

Revision 1

- Corrected minor spelling and grammar mistakes
- Added a section addressing concerns relating to unary `defer`, `dispatch`, and `post`
- Updated section on implementations
- Updated proposed changes section relating to unary `defer`, `dispatch`, and `post`

Revision 2

- Changes to unary `dispatch`, `defer`, and `post` now require that `executor_type` yield the type which is returned by `get_executor()`
- Corrected small wording typo

Review History

SG4 Teleconference May 14, 2020

Revision 0 was presented to SG4. The following polls were taken:

Forward P2161R0 to LEWG as-is?
`SF F N A SA`
`0  0 0 6 2`

We should remove the default from `associated_executor` in its current form?
`SF F N A SA`
`6  1 1 0 0`

We should explore adding a unary `associated_executor`?
`SF F N A SA`
`1  1 5 1 0`

Encourage further exploration of adding unary `associated_executor`, calling `.get_executor()`, or another way of getting an executor without a fallback?
Unanimous consent

SG4 Teleconference June 25, 2020

Revision 1 was presented to SG4. The following polls were taken:

Forward P2161R1 as presented (including discussed wording tweaks to the `executor_type` requirement) to LEWG?
`SF F N A SA`
`5  4 0 0 0`

References

[1] J. Wakely. Working Draft, C++ Extensions for Networking N4771
[2] R. Leahy. Remove `system_executor` (Revision 0) P2149
[3] C. Kohlhoff. Networking TS enhancements to enable custom I/O executors (Revision 1) P1322
[4] C. Kohlhoff. Strands: Use Threads Without Explicit Locking

[5] J. Hoberock, M. Garland, C. Kohlhoff, C. Mysen, C. Edwards, G. Brown, D. Hollman, L. Howes, K. Shoop, L. Baker, E. Niebler, et al. A Unified Executors Proposal for C++ (Revision 13) P0443
[6] https://github.com/chriskohlhoff/asio/tree/5b2720d9b52153e342a3eaa5c8723b0eec293903
[7] https://github.com/chriskohlhoff/asio/tree/95dea1ef19f9513c80d7eabe168a67df623d6928