Executors Review - Polymorphic Executor

Document #:	P2205R0
Date:	2020-08-19
Project:	Programming Language C++
Audience:	Library Evolution Library

Reply-to:	Inbal Levi <sinbal2l@gmail.com> Ruslan Arutyunyan <ruslan.arutyunyan@intel.com> Zach Laine <whatwasthataddress@gmail.com> Tom Scogland <scogland1@llnl.gov> Dmitri Mokhov <dmitri.n.mokhov@intel.com> Chris Kohlhoff <chris@kohlhoff.com> Daisy Hollman <dshollm@sandia.gov> Jared Hoberock <jhoberock@nvidia.com>

Contents

1 Motivation

This paper was created as a review of "[P0443R13]: A Unified Executors Proposal for C++", and is the result of work by attendees of LEWG listed above.

Our goal was to provide an overview of the any_executor (polymorphic executor) section of P0443, in order to identify and suggest fixes for issues that might occur.

All members of the group have put in great effort to make sure the report is thorough. Special thanks go to Ruslan Arutyunyan who joined forces with Michael J Voss, putting together many usage examples.

Special thanks also to the Authors: Chris Kohlhoff, Daisy Hollman and Jared Hoberock for doing above and beyond to provide guidance, answer our questions and suggest fixes and corrections to the potential issues raised, as well as bringing up issues of their own.

2 Polymorphic Executors

The executors mechanism suggested at [P0443R13] provides facilities for managing and executing parallel work. The polymorphic executor section is suggested as part of the executors proposal, aiming to provide an interface for creation of runtime user defined executors.

An executor is a type which receives an invokable and executes it, and a polymorphic executor is, in a nutshell, a wrapper for an executor. A polymorphic executer can have properties assigned to it, which are the mechanism by which the user can define the execution policies of the executor.

In order to allow the polymorphic executor functionality, the paper includes several facilities:

— bad_executor: Exception type support.
— any_executor: A class template that can be instantiated with different properties (please refer to the properties paper [P1393R0] for further information) containing the executor's infrastructure.
— prefer_only: A property adapter struct, used to disable the is_requirable value of a property.

The any_executor holds the 'target', which is a reference counter wrapper of the executor. When creating or copying an any_executor the reference counter increments, when deleting an any_executor the reference counter decrements.

3 Recommended Changes

3.1 Editorial Notes

3.1.1 [1.3] Executors Execute Work: on std::async equivalent example, add std::required #496

3.1.2 [1.5.1]: The R representing the receiver is not well defined #499

3.1.3 [1.4.2]: Example taken from P1897 is not up to date #502

3.1.4 Review wording of any_executor member functions that require a valid target #492

3.2 Structural Notes

The following section presents suggested changes we've discussed and agreed on in the group.

3.2.1 Properties should be well defined in both copyable and non copyable versions.

We support the change on section [2.2.11.2]: (proposed in properties group paper as well)

```
In several places in this section the operation FIND_CONVERTIBLE_PROPERTY(p, pn) is used.
All such uses mean the first type P in the parameter pack pn for which std::is_same_v<p, P>||
std::is_convertible_v<p, P> is true. If no such type P exists, the operation
FIND_CONVERTIBLE_PROPERTY(p, pn) is ill-formed.
```

3.2.2 any_executor's 'prefer' free function should be a member function.

We suggest the change:

Remove the following from the free functions section of any_executor:

```
- template <class Property, class... SupportableProperties>
- any_executor prefer(const any_executor<SupportableProperties>& e, Property p);
```

Add the following to any_executor's class:

```
+ template <class Property>
+ any_executor prefer(Property&&) const;
```

3.2.3 Change 'require', 'prefer' and 'query' to functions taking forwarding reference.

We suggest 'require', 'prefer' and 'query' to be member functions taking forwarding reference, in order to allow movable properties optimizations. The complete change includes constraints suggested in the following section.

3.2.4 Add constraints to the 'require', 'prefer' and 'query' functions.

In addition to the previous change, we suggest adding the following constraints.

We suggest the change:

```
template <class Property>
any_executor prefer(Property&&) const;

template <class Property>
any_executor require(Property&&) const;

template <class Property>
any_executor query(Property&&) const;
```

with the following description:

```
+ Let P_found be the type FIND_CONVERTIBLE_PROPERTY(Property, SupportableProperties).
+ Let Fn_cref be a function declared void Fn_cref(const P_found&);
+ Let Fn_rvref be a function declared void Fn_rvref(P_found&&);

+ Constraints:

+ * If std::is_move_constructible_v<P_found> and !std::is_copy_constructible_v<P_found>,
+   Fn_rvref(std::forward<Property>(p)) is well-formed.
+ * Otherwise, Fn_cref(std::forward<Property>(p)) is well-formed.
```

3.2.5 Align execution agent's definition with the one in the standard (possibly changing the proposed wording as well).

The paper defines the lifetime of an execution agent in the following way:

```
For the intent of this library and extensions to this library, the lifetime of an execution
agent begins before the function object is invoked and ends after this invocation
completes, either normally or having thrown an exception.
```

The standard defines an execution agent as: "An execution agent is an entity such as a thread..."

From the wording in the paper, after invocation, the execution agent has reached the end of its lifetime. In case of reallocating work to the thread (such as in thread pool) the different functionality to the same thread will be considered as a newly allocated execution agent.

We suggest: Use of execution agent as defined in the standard, redefine execution context.

3.2.6 Properties of any_executor should use prefer_only.

In order to allow usage of properties in the any_executor wrapper, we suggest requiring them to use prefer_only. Additional discussion was made and Chris has suggested that query_only can be proposed to answer the need of understanding whether the property is available. We suggest this discussion to continue, but we would love guidance on whether or not this topic is essential as part of P0443.

We suggest the poll: Is setting a default query for property necessary as part of P0443?

3.2.7 In any_executor's copy constructors from other and from executor, improve constraints and remove deleted overload.

We support the usage of the new constraints syntax in the library.

We suggest the following change in section [2.2.11.2.1]:

```
template<class... OtherSupportableProperties>
any_executor(any_executor<OtherSupportableProperties...> e);

- Remarks: This function shall not participate in overload resolution unless:
- * CONTAINS_PROPERTY(p, OtherSupportableProperties), where p is each property in
- SupportableProperties....

+ Constraints:
+ * any_executor<OtherSupportableProperties...> is not the same type as any_executor.
+ * can_require_v<any_executor<OtherSupportableProperties...>, P>, if P::is_requirable,
+   where P is each property in SupportableProperties....
+ * can_prefer_v<any_executor<OtherSupportableProperties...>, P>, if P::is_preferable,
+   where P is each property in SupportableProperties....
+ * and can_query_v<any_executor<OtherSupportableProperties...>, P>,
+   if P::is_requirable == false and P::is_preferable == false, where P is each property in
+   SupportableProperties....

Effects: *this targets a copy of e initialized with std::move(e).

- template<class... OtherSupportableProperties>
-  any_executor(any_executor<OtherSupportableProperties...> e) = delete;

- Remarks: This function shall not participate in overload resolution unless
```

```
- CONTAINS_PROPERTY(p, OtherSupportableProperties) is false for some property p in
- SupportableProperties....

template<not-same-as<any_executor> Executor>
+   requires executor<Executor>
any_executor(Executor e);

- Remarks: This function shall not participate in overload resolution unless:

+ Constraints:
+ * Executor is not a specialization of any_executor.
+ * Executor satisfies executor.
* can_require_v<Executor, P>, if P::is_requirable, where P is each property in
  SupportableProperties....
* can_prefer_v<Executor, P  >, if P::is_preferable, where P is each property in
  SupportableProperties....
* and can_query_v<Executor, P>, if P::is_requirable == false and P::is_preferable == false,
  where P is each property in SupportableProperties....

Effects: *this targets a copy of e.
```

3.2.8 In any_executor's operations section [2.2.11.2.5], replace 'this function shall not participate in overload resolution' with requires-clauses.

We suggest to change the restrictions in any_executor operations section to the constraints form, similarly to the ones in the section above.

3.2.9 Correct the behavior of `prefer_only` for move-only properties (Non-movable properties #511).

Issue #511 addresses the behavior of `prefer_only` for move-only properties.

As a rule, we suggest prefer_only to be able to constrain the properties as little as possible. We suspect that the current syntax allows passing a reference to the inner property, therefore, it can support this without a change. In order to add reference semantics to the `prefer_only` wrapper, the following change to the member `property` in `prefer_only` struct was suggested.

We propose the following change in `prefer_only` struct, section [2.4.2]:

```
std::conditional_t<
    std::is_move_constructible_v<InnerProperty>
    InnerProperty, const InnerProperty&> property;
```

A similar change could be made to `query_only`, if added.

Chris tested this code, but this is a **work in progress**.

4 Open Discussion Issues

4.1 The model of std::function vs. std::any

In the paper we've observed functionality which uses std::function. std::function was the only available model when this part of the proposal was drafted, we suggest following the std::any model instead.

4.1.1 Incentive for std::any model:

One use case for the polymorphic wrapper is to store an executor (of indeterminate type) but then to destroy it when finished with it. For example:

```
any_executor<> ex = get_an_executor_from_somewhere();
execution::execute(ex, []{ /*...*/ });
ex = nullptr;
```

This may be important when the executor object itself could represent some kind of "resource", e.g. outstanding_work.tracked. One difference between the std::function and std::any wrappers, is that the latter has an explicitly named clear() member. To clear a std::function, one must assign a nullptr or default-constructed object. An explicitly named member may better communicate the intent here.

4.2 Polymorphic wrapper

During the discussion, a question came up of whether a general purpose polymorphic wrapper utility proposal should be discussed separately, since we believe the usage of such a wrapper extends beyond the limits of executors library. While using concepts syntax and examining the facility, we've encountered issues related to SFINAE. We believe creating a facility might save the users the trouble of dealing with such issues.

A detailed description for the topic is in section [4.6] of this paper.

4.3 Add an explicit terms and design section

As described in the discussion on P0433, which was part of the Albuquerque meeting, it seems like the entities defined in the paper might be vague, even to the committee crowd. We suggest adding a separate section that will contain:

— Coherent definitions to terms used throughout the paper.
— Design of the library (described apart from the API)

Having this will go a long way towards communicating the design better to the reader, as well as clear minor conceptual differences in the designed module, and resolve minor inconsistencies such as in section [3.2.5] in this paper.

Recent developments suggest creating a TR for executors. We support adding extended design and examples as part of the TR as well.

4.4 Add a (Short!) rationale for design choices

We suggest adding a rationale for each of the design decisions made in the paper, as part of the design, or in a separate section. As came up in past debates, the executors proposal has been going on for quite some time, and has many authors. A lot of discussion was had, and a description of the rationale for the design of the final paper can be useful, in order to challenge the assumptions made in the paper or agree with them. It will also explicitly state issues that have been considered, making room for topics that haven't to be brought up by members of the committee.

an example of this exists in the paper:

[Note: To meet the noexcept requirements for executor copy constructors and move constructors, implementations may share a target between two or more any_executor objects. –end note]

In conclusion, we suggest adding a section to P0443, similar to, yet expanding the one below:

— Reasoning for existing Scheduler (concept)

Rationale: Scheduler was created separately from executor to provide a lazy execution functionality. It provides the described functionality, as well as allows error handling channels.

— The target may be shared by instances of any_executor.

Rationale: This is the result of the demand: noexcept executor's copy constructor and operator=

— Allowing Non-movable, non-copyable properties.

Rationale: There have been claims that non-copyable property support is not needed, nevertheless, the aim is to avoid binding limitations.

4.5 Explicitly specify the level of abstraction of the library

During the work on this paper, the question of "who will be the audience of this library?" ("which layer of the 'onion' are we aiming for?") came up. The general notion in several discussions varies from users to library implementers. We believe some reference to this is important, since a well defined target audience will affect the level of 'defaultness' in the library.

Our idea for a solution is explicitly targeting library authors by specifing that executors library contains low level facilities, while adding a section that briefly suggests some CPO defaults, which, in turn, create a simplified API.

An additional related suggestion that came up during the discussion was to add a note which will contain examples for defining executor types' aliases for different domains.

4.6 A general remark on usage of concepts and constraints

During the work on this paper, Chris sent us a code example which brought up a SFINAE issue. The issue remained while moving to constraints syntax usage as well.

Daisy and Saar Raz have both come up with a solution by adding a 'short-circuit' constraint:

```
requires (!std::is_same_v<E, any_executor> && executor<E>)
```

This led us to change the specification of the constraints in any_executor's class. Yet, we would like to bring the issue up for discussion, since we believe it's fundamental to the usage of concepts.

Consinder the following code snippet:

```
template <class T>
concept executor =
    requires(const T& t, invocable_archetype f){
    { t.execute(static_cast<invocable_archetype&&>(f)) };
  } && std::is_nothrow_copy_constructible_v<T>;

template <class...>
class any_executor
{
public:
  template <class E>
    requires (!std::is_same_v<E, any_executor> && executor<E>)      // This is essential
  any_executor(E) { }
  any_executor(const any_executor&) noexcept { }

  template <class F>
  void execute(F) const { }
```

```
};

template <executor E = any_executor<>>
class foo
{
public:
  explicit foo(E) { }
  foo(const foo&) noexcept { }
};
```

We have recieved feedback that the issue was also addressed by Richard Smith a few years ago, we believe this topic is worth a discussion as part of EWG.

NOTE: The result of this discussion will affect the constraints described in this document.

5 Examples

During our review, we've reached the conclusion that in order to communicate the proper way to use executors, more usage examples should be provided. Our original intent was to suggest adding a few of them as guidance to P0443, yet, if a TR will be created, they can fit in it instead.

The first example is courtesy of **Chris Kohlhoff**. The second example is part of a batch that was created by **Ruslan Arutyunyan** and **Michael J Voss** (much appreciated!), and was collected in a seperate paper. (they also appear in the properties review paper [P2183R0])

5.1 Polymorphic executor basic usage

The following example demonstrates the usage of any_executor for creating executors on runtime.

The full example can be found here: factory_1.cpp

```
typedef execution::any_executor<> executor_type;
std::static_thread_pool system_pool(16);
executor auto system_ex = pool.executor();

executor_type make_executor(const std::string& type, const std::vector<std::string>& options)
{
    if (type == "system")
    {
          return std::require(system_ex, execution::blocking.never);
       }
       else (type == "debug")
       {
            if (!options.empty())
            {
                 if (executor_type inner_executor = make_executor(options[0],
                     {options.begin() + 1, options.end()}))
                 {
                      return debug_executor<executor_type>(inner_executor);
                 }
            }
        }
        return {};
```

```
}

int main(int argc, char* argv[])
{
    execution::any_executor<> ex = make_executor(argv[1], {argv + 2, argv + argc});
}
```

The example is courtesy of Chris.

5.2 Polymorphic executor with properties

The following example demonstrates creating two any_executor types with the same properties (in changed order), and the usage of them in the algorithm.

```
using any_exec_type =
  asio::execution::any_executor<asio::execution::blocking_t,
                                asio::execution::prefer_only<toy::tracing_t>,
                                asio::execution::blocking_t::always_t>;

using any_almost_same_exec_type =
  asio::execution::any_executor<asio::execution::blocking_t,
                                asio::execution::blocking_t::always_t,
                                asio::execution::prefer_only<toy::tracing_t>>;

void algorithm(any_exec_type ex) {
  auto tt_ex = asio::require(ex, asio::execution::blocking.always);
  std::cout << "Required blocking.always" << std::endl;
  if (asio::query(tt_ex, toy::tracing))
    std::cout << "Using tracing" << std::endl;

  int i = 0;
  asio::execution::execute(tt_ex, [&i]() { i = 1; });
  asio::execution::execute(tt_ex, [&i]() { if (i == 1) i = 2; });
  std::cout << "i == " << i << std::endl;
}

void algorithm_tracing(any_almost_same_exec_type ex)
{
  auto tracing_exec = asio::prefer(ex, toy::tracing_t{true});
  algorithm(tracing_exec);
}

void combined_example() {
  toy::toy_tbb_context ttc;
  algorithm(ttc.executor());
  std::cout << "traced_executions == " << ttc.traced_executions() << std::endl;

  algorithm_tracing(ttc.executor());
  std::cout << "traced_executions == " << ttc.traced_executions() << std::endl;

  asio::static_thread_pool stp(4);
  algorithm(stp.executor());
  std::cout << "traced_executions == " << ttc.traced_executions() << std::endl;
  algorithm_tracing(stp.executor());
```

```
  std::cout << "traced_executions == " << ttc.traced_executions() << std::endl;
}
```

The example is courtesy of Ruslan & Mike.

6 References

[P0443R13] Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, D. S. Hollman, Lee Howes, Kirk Shoop, Lewis Baker, Eric Niebler. 2020. A Unified Executors Proposal for C++. https://wg21.link/p0443r13

[P1393R0] D. S. Hollman, Chris Kohlhoff, Bryce Lelbach, Jared Hoberock, Gordon Brown, Michał Dominiak. 2019. A General Property Customization Mechanism. https://wg21.link/p1393r0

[P2183R0] David Olsen, Ruslan Arutyunyan, Michael J. Voss, Michał Dominiak, Chris Kohlhoff, D.S. Hollman, Kirk Shoop, Inbal Levi. 2020. Executors Review: Properties. https://wg21.link/p2183r0