C++20 Executors are Resilient to ABI Breakage | P1405R0

Authors: Jared Hoberock, jhoberock@nvidia.com
Chris Kohlhoff, chris@kohlhoff.com
Document Number: P1405R0
Date: 2019-01-21
Audience: SG1 - Concurrency and Parallelism, LEWG
Reply-to: sg1-exec@googlegroups.com
Abstract: This paper argues that P0443's model of executors does not present a signficant ABI risk and provides guidance to implementors for avoiding ABI breaks as C++ executors evolve.

Introduction

While discussing the feasibility of adopting P0443's executor model for C++20 in San Diego, some committee members voiced concerns that certain P0443 features may be susceptible to ABI breakage. Because some vendors cannot productize C++ Standard Library features which break ABI, it is important to demonstrate that ABI breaks can be avoided as P0443 model of executors evolves.

Most of the functionality of P0443 are property types intended to be used as empty or nearly-empty tag types. We do not envision adding functions or state to instances of these types, and thus do not expect them to be susceptible to ABI breakage. However, there are two areas of functionality which we do expect to enhance over time: polymorphic executors and `static_thread_pool`. This paper investigates the risk of ABI breakage posed by this functionality and concludes that P0443 will be resilient to ABI changes if implementors structure their implementations accordingly.

Polymorphic Executors

Some have voiced concerns that future innovations to executors that introduce new functionality could yield ABI breaks when this functionality is reflected in polymorphic executor interfaces. We believe this concern results from a misunderstanding of how P0443's polymorphic executor model works.

P0443's `execution::executor` is a template alias. Its first template parameter is an interface-changing property which is used to select the actual polymorphic executor type:

```
template<class InterfaceProperty, class... SupportableProperties>
using executor = typename InterfaceProperty::template
  polymorphic_executor_type<InterfaceProperty, SupportableProperties...>;
```

P0443's polymorphic executor model limits the applicability of each polymorphic executor type to a single kind of executor distinguished by its first `InterfaceProperty` template parameter. In this way, rather than nominating a single monolithic polymorphic executor type that must be compatible with any kind of executor, P0443 allows each interface-changing property to name its own distinct polymorphic executor type. The indicated polymorphic executor type only needs to support executors with that interface-changing property. This means, for example, that `OneWayExecutor`s and `BulkOneWayExecutor`s use distinct underlying

polymorphic wrapper types. As additional executor concepts are standardized, we envision that these too will use distinct polymorphic types.

Because the introduction of new executor concepts implies a corresponding introduction of distinct, unrelated polymorphic executor types, we do not expect the ABIs of existing polymorphic executor types to be fragile to new functionality.

`static_thread_pool`

The other source of P0443's ABI breakage concerns is `static_thread_pool`. Unlike polymorphic executors, `static_thread_pool` is more susceptible to ABI breaks as its standard and implementations evolve. We would like to standardize `static_thread_pool` as specified by P0443 while retaining the ability to innovate within its interface as well as implementors' ability to optimize implementations.

This leads to two questions:

1. Does P0443 contain enough customization machinery to enable future innovation?
2. How should implementors approach the portion of `static_thread_pool` which forms its ABI boundary?

In order to explore these questions, we experimented with a simplified form of `static_thread_pool`. Our motivation was to demonstrate to implementors how to apply P0443's properties system to incorporate novel models of allocation into the thread pool while maintaining an implementation hardened against ABI changes. Specifically, P0443 specifies the `execution::allocator_t` property allowing executors to expose a customizable allocator used when allocating work items. In addition to `allocator_t`, consider another hypothetical property which would allow the user of the thread pool to intrusively allocate work items himself. We seek to demonstrate how both allocation approaches can coexist within the same thread pool implementation and recommend a forward-looking implementation strategy. By adopting such a strategy, we believe implementors will avoid ABI breakage.

A sketch of the basic approach follows.

```
// consider a hypothetical interface property
// which allows the user to provide preallocated memory
// to use for execution agents created through a given executor
struct preallocated_oneway_t { ... };

class thread_pool
{
  private:
    // an abstract base class for work items
    // this type forms the ABI boundary
    struct function
    {
      virtual ~function() {}
      virtual void call(void*) = 0;
      virtual void destroy() = 0;

      // a linked list of functions
      std::unique_ptr<function, deleter<function>> next_;
    };

    // work items represented by this type use an allocator
    template<class Function, class ProtoAllocator>
    struct function_with_allocator : function
    {
```

```
      ...
    };

    // work items represented by this type use preallocated memory
    template<class Function>
    struct function_with_preallocated_memory : function
    {
      ...
    };

    ...

    // the head of a linked list of work items
    std::unique_ptr<function, deleter<function>> head_;

  public:
    // work items created by this type of executor use preallocated memory
    class preallocated_oneway_executor;

    // work items created by this type of executor use an allocator
    template<class ProtoAllocator = std::allocator<void>>
    class oneway_executor;

    // asking thread_pool for an executor creates a oneway_executor
    // a preallocated_oneway_executor may be created via
    // .require_concept(preallocated_oneway)
    oneway_executor<> executor() const;

    ...
};
```

The basic idea is to implement the thread pool's queue of work items in an allocation-agnostic way via an abstract base class. In this way, the concrete allocation approach can be controlled via user requirements and further enhanced in future revisions of the C++ Standard Library. A fully-worked implementation with additional detail is available.

Introducing New Properties

It's natural to wonder how to maintain the ABI stability of types related to executors as new properties and behaviors are standardized. After all, doesn't a new property imply the possibility of new data members which may alter class layout? Not necessarily. Let's imagine the task of introducing support for a new property type to concrete executors, polymorphic executors, and execution contexts like `static_thread_pool`.

Concrete executors. Existing concrete executor types can support new a property (say, `new_property_t`) without breaking ABI if that property can be supported without changing class layout. In such a case, a new member function overload for `.require(new_property_t)` can be safely added to the type's interface, and the result of this overload can be the original executor type. On the other hand, suppose introducing support for the property would change the executor type's layout. For example, by introducing a new piece of state. In this case, an implementor can still add a new member function overload for `.require(new_property_t)`. However, instead of resulting in the original type of executor, this function should return a new executor type. In either case, the class layout of the pre-existing, original executor type remains unchanged.

Polymorphic executors. Recall that each property has a corresponding type. Furthermore, recall that a polymorphic executor type can only support a property by naming that property's type in its list of template

parameters. Together, this means that the introduction of a new property implies the instantiation of a corresponding new polymorphic executor type. By design, existing polymorphic executors cannot support new property types they don't already know about. The result is that as new properties are introduced, they are irrelevant to pre-existing polymorphic executor instantiations, and therefore can not break their ABI.

Furthermore, P0443's specification of polymorphic executors constrains implementations towards a model of polymorphism that supports arbitrary property queries. P0443 specifies a narrowing converting constructor that converts a polymorphic executor into another polymorphic executor with a narrower set of supportable properties. In order to conform to this specification, we believe implementations will adopt an approach whereby polymorphic executors inherit from a common virtual base class. In order to support arbitrary property queries, this base class should traffic in `void` pointers.

Execution contexts. Finally, consider how execution contexts such as `static_thread_pool` might support new properties. Because we envision properties as applying to executor behavior, they are only indirectly relevant to execution contexts. As we have noted, new properties may be introduced without breaking ABI by modifying existing executor types in a way that preserves their layout, or by introducing new executor types when layout preservation is not possible.

We envision for execution contexts to produce associated executors via factory functions such as `static_thread_pool::executor`. When the executor type returned by this function may be enhanced without changing its layout, `static_thread_pool` need not be changed. On the other hand, when support for a new property requires the introduction of a new executor type, there must be a way to obtain one of these new executors. One way, as we have previously discussed, is via an additional `.require(new_property_t)` overload. Another way is by introducing a new non-virtual factory function member to the execution context. In either case, we expect that such functions may be introduced without affecting the execution context's class layout and therefore need not break ABI.

Summary

P0443 exposes two potential sources of ABI breakage. The first source is P0443's polymorphic executor facilities. Upon examination, we have disqualified this as a source of ABI problems. The second source of ABI concerns is `static_thread_pool`. We believe adopting forward-looking implementation strategies like the ones described here will lend resilience against ABI breaks amidst future enhancements to `static_thread_pool`. We encourage implementors concerned about ABI to refer to our reference implementation when implementing P0443.

Acknowledgements

Thanks to Lewis Baker, Michael Garland, Bryce Lelbach, Eric Niebler, Thomas Rodgers, Kirk Shoop, and Matthias Stern for participating in discussions on P0443's ABI stability and to Billy O'Neal for feedback on this paper as well as his implementor perspective on ABI stability.