Document number: P1888R0
Date: 2019-10-07
Project: Programming Language C++, WG21, SG1, LEWG, LWG
Authors: Matti Rintala, Michael Wong, David S. Hollman, Pekka Jääskeläinen
Email: matti.rintala@tuni.fi, michael@codeplay.com
Reply to: Matti Rintala

Executors without exception handling support

1 Introduction

The topic of this paper is to discuss how to support executors which cannot without unacceptable time or space overhead support exception handling (throwing and/or catching exceptions) in the code sent to be executed. Examples of such are executors for some heterogeneous execution platforms like GPUs, FPGAs, etc. The topic of this paper was originally a small part of *P0797R2 Handling Exceptions in Executors* [1], but in Cologne meeting (June 2019) it was decided that the topic should be separated into a separate paper, since not supporting exception handling was somewhat different topic than the rest of the paper (how to handle potentially multiple errors from a parallel execution).

2 Revision history

2.1 2019-10-07 P1888R0 BFS Meeting

- Initial version

3 The problem

Work with C++ executors [2, 3] will eventually move towards supporting heterogeneous execution platforms (GPUs, FPGAs, etc.). Those execution platforms may not be able to fully support all C++ features, at least without unacceptable time or space overhead. One such feature is exception handling (there are probably more possible C++ features that some execution environment cannot support, but those are not the focus of this paper). While theoretically it would probably be possible to support exception handling as a mechanism on every execution platform, requiring that could be technically challenging and cause a heavy burden on executor implementors, especially if demand for exception handling support is low in the domain in which the executor is expected to be used (if errors are typically signalled using some other disappointment mechanism like error codes, std::expected, for example). Supporting exceptions in an executor might also incur a significant performance overhead (or force execution to fall back to CPU).

The whole topic of error handling and propagation in executors is currently under heavy discussion and development, but this paper concentrates on situations where supporting exception handling (throwing/catching exceptions) may not be reasonable for *any* code executed under the executor, regardless of whether those exceptions are eventually propagated to the executor to be handled. Thus, what this paper discusses is different than noexcept. Even if a function is noexcept, it may internally throw/catch exceptions or call code that does so, so executing it may be problematic for certain executors due to the need to add exception handling facilities to the code executed on the execution platform. The transitive closure of everything that gets executed must have no exception handling (throws and potentially no try-catches). Another way to state that is that while noexcept is an interface property, the no-exception-handling requirement is a requirement on the transitive closure of the executed code.

It should be noted that finding out what code is executed under which executor may be a run-time issue (if code is passed to be executed using function pointers with run-time determined values, etc.), so diagnosing violations of the no-exception-handling requirement may sometimes be impossible at compile time. For example, *"Offloading C++17 Parallel STL on System Shared Virtual Memory Platforms"* [4] is an example of a proof of concept implementation of C++17 parallel STL on heterogeneous shared virtual memory platforms, which off-loads code to be executed on GPU. In that implementation, functions containing exception handling are simply omitted from the intermediate language, so that the off-loading engine fails to find those functions if they are scheduled to be executed on GPU [4, p. 7] and falls back to host (CPU) execution.

It would definitely be better to get diagnostics about violations of the no-exception-handling restriction during compilation time, and preferably as close to the user's code as possible (instead of just an obscure diagnostics about code in some third-party library, for example). This would become possible if users can explicitly express that their code shouldn't transitively contain any exception handling.

It is understood that limiting support for exception handling may be somewhat controversial, since it allows implementation of executors that do not fully implement the C++ language. However, the need for being able to use executors without exception handling capabilities has been brought up many times in heterogeneous C++ telecon discussions. The authors would like to emphasize that the intent is just to explicitly allow not supporting exception handling in the limited context of code given to be executed under certain executors. The idea is not to ignore exceptions, but to allow stating that exceptions are not allowed to occur in certain contexts. If that limitation is violated, possible outcomes could be compiler diagnostics, run-time errors, or undefined behaviour.

4 Requirements for the solution

This section outlines the properties a solution should ideally have to solve the problem. Some of the general requirements are the following:

- Since supporting exception handling is difficult, costly, or even impossible in some execution environments (GPUs for example), it should be possible to provide executors that do not support exception handling and detect use of those executors in generic code.
- As a variation of the above, it should be possible to implement an executor that provides fast execution for code that does not require exception handling support, and slower "fallback" execution for code requiring exception handling. (For instance, a CUDA compiler could migrate the entire stack back to the CPU, hard block the device, and handle exceptions there.)
- Since exception handling is part of the core language, it should always be supported by default, and not supporting it should be explicitly stated in the code. Also, thrown exceptions should never be silently ignored.
- Since not supporting exception handling concerns just some executors, the mechanism should be such that others can ignore it completely (i.e., if you are happy with exceptions, you shouldn't have to write any extra code).
- If exception handling support is possible but slow (which is often the case with data parallelism, like vectorization), users should be able to choose between the faster execution without exceptions and slower execution with exception handling support. The "faster execution" means speed relative to the same executor with exceptions enabled. In some cases the compiler could deduce from the code whether exception handling support needed, but this is generally not possible in all cases.

These requirements will affect at least three parties, described below.

4.1 Executor implementors

Parties implementing executors would have at least the following needs:

- Mechanisms for not supporting exception handling should not affect implementors of executors which always support exception handling. No additional code should be required to support exception handling.
- For execution environments where supporting exception handling is infeasible (or at least so expensive that exceptions are not ever used), it should be possible to implement executors that do not support exception handling at all. Trying to use such an executor to execute code that uses exception handling should fail, at least at run time, but preferably at compile time (at least as often as possible).
- In environments where supporting exception handling is possible but costly (supporting exception handling incurs considerable space/time cost even if exceptions are not thrown), it should be possible to write executors which provide both "default" slower execution with exception handling supported, and also faster execution for cases where exception handling support is not required. Users and generic algorithm users should be able to choose which they prefer, or the compiler could automatically use the faster approach if it deduces that exception handling support is not needed.

4.2 Executor and generic algorithm users

Executors can either be used directly to execute code, or they are passed to generic algorithms. In both cases the users of executors would need at least the following:

- The default should be that any code provided to the executor may contain exception handling, unless otherwise indicated or deduced. Again, if the possibility of getting faster execution while not having exception handling support is of no interest to the user, no additional code should be required.
- Execution of code should be as fast as possible (but no faster). That is, users do not want to pay for something they do not need (exception handling support for code which has no exception handling). But if the user's code requires exception handling support, it should be used even if it slows down execution, or the user should be informed that the executor doesn't support executing code containing exception handling.
- The code submitted to be executed by the executor either requires exception handling support or not, depending on the situation. In an optimal world the compiler would always be able to deduce whether code requires exception handling support and make execution faster by omitting exception handling if that's not the case. In any case it would be useful if the user explicitly expresses whether the code is expected to require no exception handling (possibly allowing faster execution), or whether the code definitely needs exceptions. Explicitly expressing the intent would allow the compiler to issue diagnostics in some cases when the code does not match the intent (exceptions are used in code which is marked as not requiring exception handling support). It should be noted that the existence of try-catch blocks in the code does not necessarily mean that exception handling support is required, if it's not possible that exceptions ever get thrown in the code (in which case the catch clauses could simply be ignored).
- If the executor does not meet the user's requirements (it cannot support exception handling even though such support is needed), diagnostics should be provided. Preferably during compilation,

but at least during run time. The mismatch should never result in different behaviour of the program.

4.3 Generic algorithm implementors

With "generic algorithm" this paper refers to (generic) code that receivers executors and code to be executed as parameters. Generic algorithms use the provided executor to execute the provided user code, the algoritm's own code, or a combination of those (user's code wrapped in algorithm's code, for example). That results in at least the following needs:

- Once again, if the generic algorithm writer is not interested in the possibility of gaining speed from not requiring exception handling support, the writer should be able to ignore the topic completely with no ill effects. (Of course the compiler could still be able to deduce that in some cases no exception handling is actually ever needed and choose faster execution without exception handling support as an optimization.)
- On the other hand, generic algorithms that explicitly do not require exception handling support should still be able to use "normal" exception-handling-enabled executors without any modifications.
- If the provided executor cannot support exception handling while that is required by the executed code (user's code, algorithm's code or a combination), diagnostics should be provided, preferably during compilation.
- Since an algorithm may use the executor to run several kinds of codes, some of which require exception handling and some of which don't, the algorithm should be able to explicitly choose between exception handling support and faster execution without it.
- Since the code to be executed may be a combination of user-provided code and algorithm's own code, there should be an easy way to request faster execution only if the combined code doesn't require exception handling. Again, requiring both the user and the algorithm writer to explicitly express whether they think exception handling is required makes it possible for the compiler to issue diagnostics when the expressed intent is in conflict with the executor, even if the compiler cannot check all of the actual code.

5 Approaches considered

The aim of this paper is to get feedback on which direction to take to address the need for non-exception-handling-supporting executors. The approaches described below are general directions that differ in how implicit/explicit they are, how much compiler magic is needed, etc. Further study is needed for any of these approaches once the general direction is chosen.

In the approaches described below, there are three ways to interpret what "no exception handling required" means. Clearly throwing an exception is exception handling, and code directly or indirectly containing throw statements needs exception handling (unless the compiler can prove the throw statement can never be executed).

If the code contains throw statements that are never executed, that could count either as exception-handling-required or not. However, if the compiler cannot prove that the throw can never be executed, diagnosing violations of the no-exception-handling limitation would require replacing the throw statement with some diagnostics code (like calling std::terminate), which could be almost as expensive as supporting exceptions.

Finally, if the code contains just try-catch blocks, they could either be regarded as "exception handling" (and be prohibited on executors not supporting exception handling), or the catch blocks could simply be silently ignored, since they do not participate in the semantics of the code unless exceptions are thrown (which is not allowed).

The transitive no-exception-handling-required property discussed above should be contrasted with the -fno-exceptions compiler flag, which causes the compiler not to emit any exception handling code to the output and diagnose both throw statements and try-catch blocks. But least on gcc, functions compiled with -fno-exceptions can be called from code compiled with -fexceptions and vice versa without diagnostics, causing incorrect run-time semantics if exceptions occur.

Finally, it may be questionable whether the Executor concept allows executors which cannot support exception handling. Exception handling is part of the C++ language, and therefore *every* function/callable in C++ is allowed to do it, and it's implicitly part of the Callable concept, which Executor requires from the code passed to it. If an executor cannot support exception handling at all (and not just execute slower with it enabled), does it model the Executor concept? If not, then even executors, for which exception handling is unfeasible, should still somehow implement exception handling support (by falling back to CPU execution, for example), and provide execution without exception handling as an optimization. From the user's point of view this is probably not as good as getting a compile-time or run-time error.

If Executor's execute is interpreted to require supporting exception handling, it could be possible to add a customization point like execute_no_eh(), whose contract does not require exception handling support. It would by default call execute(), but executors could customize it to provide faster execution without exception handling support.

5.1 Require all executors to support exception handling

One obvious option is to do nothing, i.e. require that all executors support exception handling (at least for now), even if terribly slowly, or that they fallback to host execution if the compiler cannot prove that executed code requires no exception handling. In a few years time it's likely that execution platforms like GPUs get more helpful features for integration with the host execution (this seems to be the direction with unified memory etc.), so it's possible that supporting exceptions becomes easier on heterogeneous platforms as well. This is also tied to proposals like "Zero-overhead deterministic exceptions: Throwing values" [5] which may make low-overhead exception handling easier to achieve in the future.

5.2 Implicit support, compiler diagnostics/std::terminate on violation

The easiest way to allow executors without exception handling support is to specify that such executors either cause compiler diagnostics or call std::terminate, if an attempt is made to execute code containing exception handling on the executor. Similarly if exception handling support is possible but slower, the compiler and/or run-time environment could choose between faster/slower execution based on whether it detects the code not to require exception handling support.

One problem with this approach is that it's completely implicit. If an executor does not support exception handling but code using exceptions is set to be executed, there are situations where it's impossible to recognize this during compile time (function pointers with run-time determined values is one such example). The problem would most likely still be detected at run time when the actual code to be executed is known and an attempt is made to off-load it to a GPU, for example, so calling std::terminate is possible.

Similarly the speed gain from not having to support exceptions would be essentially an optimization in this approach. With an executor that can provide both fast exceptionless execution and slower execution with exception handling, the benefit is that all code not requiring exception handling would automatically get faster execution. But on the other hand there would be no way to *require* faster execution (and get diagnostics if exception handling is needed by the executed code), and even finding out whether faster or slower execution was chosen (and why) could be challenging.

The implicitness is problematic since not requiring exception handling is essentially a "deep noexcept" that also applies to all library functions called by the executed code, etc. Therefore it's possible that even if some code doesn't need exception handling now, a new version of a library called by the code might change that, suddenly causing either std::terminate or unbearably slow execution.

5.3 Implicit support, violation is undefined behaviour

Another possible approach is similar to the one in the section above, but making it undefined behaviour to attempt executing code requiring exception handling on an executor not supporting it. This would make it even more difficult to diagnose violations of the executor requirements, and possibly even to notice them. On the other hand, it would allow the compiler to ignore code paths ending in a throw statement, as well as catch blocks, when compiling code to the platform not supporting exceptions.

This approach would probably be much more problematic in practise, since undefined behavior could make really difficult to verify that the code to be executed really has the "deep noexcept" property.

5.4 (Mis)use noexcept to indicate no-exception-handling-required

One way to diagnose some violations of the no-exception-handling requirement would be to require that code passed to an executor not supporting exception handling should be noexcept. That requirement is not strict enough, because noexcept functions can still internally do as much exception handling as they want. However, it would still require the user to explicitly mark their code as noexcept, and noexcept code containing exception handling could be detected later during compilation or at run-time.

Even with this approach it may be unclear whether an executor requiring the callable to be noexcept is really modelling the Executor concept, which has no such requirement.

5.5 Use attributes to indicate no-exception-handling-required

One way to make the intent of the user more explicit would be to allow users to mark their code as not requiring exception handling. For example, a new attribute `[[no-exception-handling]]` (bikeshedding required) could be used to mark code which is supposed to require no exception handling support. An executor not supporting exception handling could then require that code passed to be executed has to have this attribute, or it could choose faster/slower execution strategy based on the existence of the attribute. As an optimization, in many cases the compiler could still be able to deduce this attribute from the code without the user having to explicitly provide the attribute.

At least in some cases the attribute could also be used to check that code marked as `[[no-exception-handling]]` indeed doesn't contain exception handling. However, since all function calls etc. not marked with the attribute potentially require exception handling, it would be infeasible for the compiler to check everything (at least until link time when the implementation of every function is available). However, the attribute would still make it possible to spot cases where code marked with the attribute directly contains exception handling.

One problem with this approach is that generic algorithms often wrap their own code around provided user code. If the algorithm's own code didn't require exception handling, the combined code would require exception handling only if the provided user code required it (i.e. was decorated with the attribute). This would require generic algorithms to be able to query the existence of the attribute in provided user code, and conditionally mark the algorithm's code as `[[no-exception-handling]]`. Currently there's no way in C++ to query attributes in generic code, and that capability doesn't exist in the Reflection TS [6] either, even though such support has been suggested elsewhere [7].

Using an attribute would make exception handling requirements more explicit, but that alone would probably not allow the compiler to diagnose all violations of the attribute. Calling std::terminate (or undefined behaviour) would still be needed for the cases which the compiler didn't notice.

5.6 Use executor properties

One possibility to handle restricting exception handling would be to use the properties mechanism suggested for executors [8]. The property mechanism allows attaching a set of properties to an executor, and these properties may both affect the behaviour of the executor, and they can be queried by user or generic algorithm code.

Since the properties can affect the behaviour of the executor, it would definitely be possible to implement the choice between faster execution without exceptions and slower execution with exceptions by property *no-exception-handling-support-required* (bikeshedding definitely required), that users can use to express that executed code doesn't need exception handling. "Normal" executors which do not benefit from disabling exception handling could simply ignore this property (i.e., users could only prefer this property, not require it). Using a property would allow generic algorithms to query the executor about this property, and adapt if necessary. This property would be supported only by executors which either gain performance by not supporting exception handling, and by executors where not supporting exception handling is the only supported option.

As exception handling is part of the C++ language, it would be best if *no-exception-handling-support-required* property is never set by default for any executor. For executors which cannot provide exception handling support, this could be handled by having a "default" executor that cannot execute anything (trying to do so would be a compilation error), and the actual working executor is obtained by preferring the *no-exception-handling-support-required* property. That would increase safety, since the user/algorithm would have to explicitly request an executor without exception handling support. The question in this kind of approach is whether Executor concept allows providing execute() but causing compilation error if it's actually called.

Similarly there could also be property *exception-handling-supported,* that could be required by code that explicitly needs exception handling support on executors where that's not always the case. That would allow executors which do not support exception handling at all to not support this property, causing requiring the property to fail. The property mechanism allows defaults for the properties, so executors which do not care about not supporting exception handling could have the property set by default without even having to be aware of it.

These kind of properties would allow the executor to choose whether to support exception handling or not. Similarly the user of an executor could signal whether exception handling support is required or not. The situation for a generic algorithm writer would still be problematic, though. If a generic algorithm receives an executor and user code to be executed (possible wrapped inside algorithm's own code), the algorithm can query whether the executor can support exception handling and whether the property *no-exception-handling-support-required* has been set. However, it wouldn't have any knowledge on whether the provided user code contains exception handling or not. There would have to be a documented contract saying that if an executor has *no-exception-handling-support-required* set, then any code provided to be run on it would have to be without exception handling (or course the violation would eventually be noticed, at run-time at least).

Similarly, if a generic algorithm not aware of the *no-exception-handling-support-required* property is passed an executor where the property is set (and the executor doesn't support exception handling), then the algorithm's own code sent to the executor could still contain exception handling, and the violation would be noticed later, possibly at run-time.

5.7 Summary

Below is a table containing the discussed approaches and their main features.

	Always require EH	**Implicit no-EH**	**Implicit no-EH + UB**	**noexcept**	**Attributes**	**Properties**
Explicit executor requiremen ts	—	—	—	(✔)	(✔)	✔
Explicit user intent	—	—	—	(✔)	✔	(✔)
Violations caught	—	✔	—	✔	✔	✔
Gen. algo can query executor	—	—	—	—	(✔)	✔
Gen. algo can query user code	—	—	—	✔	(✔)	—

6 Design questions

- The main design question is, which direction should be taken, if any?
- Are executors not supporting exception handling common enough to take into account in the standard?
- How early should violations of the no-exception-handling requirement be detected (is leaving that to run-time ok)?
- Do executors without exception-handling-support model the Executor concept?

7 Acknowledgement

Thanks for Michael Wong, David Hollman, Gordon Brown, Patrice Roy, Domagoj Šarić, Mark Hommen, and everyone on the Heterogeneous C++ telecon for their input and comments.

8 References

[1] Matti Rintala, Michael Wong, Carter Edwards, Patrice Roy, and Gordon Brown. *Handling Exceptions in Executors.* P0797R2

[2] Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, David Hollman, et al. *A Unified Executors Proposal for C++.* P0443R10

[3] Jared Hoberock, Michael Garland, Bryce Adelstein Lelbach, Michał Dominiak, Eric Niebler, Kirk Shoop, Lewis Baker, Lee Howes, David S. Hollman, Gordon Brown. *A Compromise Executor Design Sketch.* P1660R0

[4] Pekka Jääskeläinen, John Glossner, Martin Jambor, Aleksi Tervo, Matti Rintala: *Offloading C++17 Parallel STL on System Shared Virtual Memory Platforms*, 3rd Workshop on Open Source Supercomputing (OpenSuco3 within ISC 2018, June, Frankfurt, Germany)

[5] Herb Sutter. *Zero-overhead deterministic exceptions: Throwing values.* P0709R4

[6] *Working Draft, C++ Extensions for Reflection.* N4818 (June 17, 2019)

[7] Manu Sanchez. *Reflections On User Defined Attributes.* *https://manu343726.github.io/2019-07-14-reflections-on-user-defined-attributes/* (July 14, 2019)

[8] David Hollman, Chris Kohlhoff, Bryce Lelbach, Jared Hoberock, Gordon Brown, and Michał Dominiak. *A General Property Customization Mechanism.* P1393R0