Exception Safety

Background

Embedded development generally disallows exceptions and RTTI. Even non-embedded projects (e.g., Firefox, Chrome, Microsoft Windows) ban RTTI (and exceptions) for similar reasons.

What are these reasons?

1. Violates zero overhead principle
    a. "The overhead arises in various places: In the binary image, we have to store jump tables or other data/logic. At run time, most implementations reserve additional stack space per thread (e.g., a 1K reservation, to save a dynamic allocation) and require and use more-expensive thread-local storage." -P0709R4
    b. Potentially additional stack space reserved per thread
2. Violates the determinism principle
    a. throw requires dynamic memory allocation, catch requires RTTI
        i. RTTI is bad for catch here as even though exception handling usually only requires an upcast from a statically known derived type to a statically known base type, dynamic_cast must still perform other casts (sibling cast, downcasts to statically unknown/known types) (more info in P0709)
        ii. Dynamic memory allocation could throw an exception itself and is thus non deterministic.
    b. Exception flow control is "invisible"
    c. "C++ allows there to be multiple active exception objects of arbitrary types, which must have unique addresses and cannot be folded; and it requires using RTTI to match handlers at run time, which has statically unpredictable cost on all major implementations and can depend on what else is linked into the whole program. Therefore during stack unwinding the exception handling space and time cost is not predictable as it is with error codes. Adequate tools do not exist to statically calculate upper bounds on the actual costs of throwing an exception." -P0709R4

Existing References and Standards

Google C++ Coding Standard

1. Exceptions disallowed
    a. **Quote: “We do not use C++ exceptions.” https://google.github.io/styleguide/cppguide.html#Exceptions**
2. RTTI is discouraged.

a. **Quote: “Avoid using Run Time Type Information (RTTI).” https://google.github.io/styleguide/cppguide.html#Run-Time_Type_Information__RTTI_**

DoD JSF C++ (Stroustrup *et al.*, 2005):

1. Exceptions disallowed (“not 100% predictable from a performance perspective”)
    - **Quote: “Tool support is not adequate at this time”**

Some other worthwhile excerpts:

2. Heap allocation after initialization disallowed (same rationale)
    - **Quote: “Heap fragmentation and hence non-deterministic delays in heap access”**
3. dynamic_cast (RTTI) disallowed (same rationale)
    - **Quote: “Not allowed at this point due to lack of tool support”**
4. Also: all libraries used must be DO-178B level A certifiable or written in house and developed using the same software development processes required for all other safety-critical software. This includes both the run-time library functions as well as the C/C++ standard library functions.

Adaptive Autosar C++14 (2017-2019, *deprecated*):

1. Exceptions

    Exceptions are permitted conditionally, *assuming the following are addressed:*
    - Worst time execution time (WCET)
        - **E.g., override memory allocation in gcc/clang exception impl. (cf. Apex.AI static_exception; ~~*but conflicts with A15-1-1: Only instances of types derived from std::exception should be thrown?*~~)**
        - **What about RTTI in catch clause?**
    - Prefer “zero cost exception handling” when exceptions are not used
    - Follow Java’s checked und unchecked exception paradigm (via Doxygen, tooling)
        - **Expected errors (reasonable to recover from)** are to be documented with @throws tag. Those need to be handled or in turn documented in @throws tag of calling function (checked via tool support). Corresponding exception classes carry a @checkedException tag before their class declaration.
        - **Exception corresponding to unrecoverable errors (default)** do not need to be documented by function or calling functions. Examples: Software errors, i.e. preconditions/postconditions violations, arithmetic errors, failed assertions, sanity checks or invalid variable access (typically logic_error, bad_exception, bad_cast and bad_typeid or their subclasses). Internal errors (e.g., bad_alloc and bad_array_new_length).

- All operations provide either a basic exception guarantee (no resource leak, no object invariant violated upon throw), a strong guarantee (function succeeds or no effect), or a nothrow guarantee.
- Compiler supplier or project to perform analysis for typical failure cases: correct stack unwinding, exception not thrown, other exception thrown, wrong catch activated, memory not available while handling exception.
- Prefer last-ditch *"catch all"* for all exceptions (incl. those from external libraries) to avoid the case of uncaught exceptions (because behavior is unclear: may unwind stack or not before termination)
- If a function's noexcept specification can not be determined, then always declare it to be noexcept(false) (otherwise risk of calling std::terminate()). Conversely, a known non-throwing function shall always have a noexcept specification.

2. Other topics in Autosar C++14

Heap allocation is permitted conditionally, *assuming the following are addressed:*

- Memory leaks (→recommends RAII and smart pointers)
- Memory fragmentation
- Invalid memory access
- Memory allocation failures (→suggests pre-alloc.)
- Deterministic execution time (for a given WCET; includes any kernel calls.)
  - **Quote: need to be executed without context switch and without syscalls**
  - **Quote: need to provide custom implementations of new, delete, malloc, free to hide incorrect dynamic memory alloc./dealloc. in linked libraries**

dynamic_cast (RTTI) is disallowed (*"unsuitable for R-T systems where determined performance are essential"*)

Also: all project's code including used libraries (including standard and user-defined libraries) and any third-party user code shall conform to the AUTOSAR C++14 Coding Guidelines.

- **In practice: Must be developed to ISO26262 or qualified per ISO26262.**

Current Proposals

P0709: Zero-overhead deterministic exceptions (2018-)

Forward looking (C++26 or after); proof of concept implementation is TBD.

Some points:

- There should be **no exceptions** corresponding to unrecoverable errors (*"unchecked"* in Adaptive Autosar C++ lingo above)

    - **Use contracts, asserts for those (e.g., for bugs, pre-/postcondition violations, ...)**
- What remains are recoverable (cf. *"checked"*) exceptions; exception usage should reduce dramatically (*"with contracts, 90% of exceptions become preconditions."*)
- Proposes static exception type akin to Boost expected / Outcome / Rust result type that supports exception semantics and automated propagation.
    - ***"Static by default, dynamic by opt-in"***
- Also mentions static (by default) RTTI (e.g., specific downcasts can be optimized) and support of more complex dynamic_cast by opt-in.

Discussion

Question: Possible to realize "safe exceptions" with today's C++ in today's applications?

Key issues:

- ***Memory allocation*** *(Issues: compiler-level and stdlib-level allocation):*
    a. **Own code:** e.g., Apex.AI static_exception works around compiler-level allocation with memory pool.
    b. **Library code:** e.g., fix std::exception string usage -- cf. Apex.AI CppCon 2019 suggestion

    To group: Have we then solved the memory issue for exceptions?
    Andreas: released code has unit tests to show no malloc/new is called. Incl. thread safety. Itanium ABI -- both clang and gcc → currently gcc only, may work with clang (req. changes)
- ***RTTI*** *(Issues: used in catch clause; general non-determinism of RTTI)*
    a. **Herb:** *"It is an open research question whether C++'s currently specified RTTI is implementable in a way that guarantees deterministic space and time cost."*
    b. **Question:** For particular form of dynamic_cast (that is relevant for exceptions[1]), is it implementable in a way that guarantees deterministic (space and/or) time?
    Follow-up: What if we restrict the particular inheritance hierarchy of exceptions used in the project (no virtual inheritance, no multiple inheritance, etc.)?
    (Any thoughts from compiler vendors here?)
    c. **Question:** If not, can we argue that it's deterministic (and measurable) for the particular program under consideration, permitting a WCET analysis for the program?
    (**But:** *"change in process as shared libraries are loaded and unloaded can lead to different RTTI times."*)
    (**But:** *"Niall Douglas reports real-world cases where a user's linking in other code caused the cost of throw...catch to rise dramatically (e.g., 500ms on a heavily loaded machine)"*)

    To group: Thoughts on deterministic RTTI for exceptions, e.g., by loading all dynamic libraries upfront or only permitting static libs?
- ***Code coverage / invisible control flow***
    - Refers to the lack of visibility what code execution paths are realized in a program that supports exceptions. Especially since multiple exceptions can be in flight. Makes code harder to reason about.

[1] Potential follow-up: Arthur O'Dwyer cppcon '17 talk *"dynamic_cast From Scratch" (sl. 39 and 42ff.)*

- **Proposal:** What if we seek a compromise between Herb's and Adaptive Autosar C++14 recommendations? I.e., only allow *"checked"* exceptions (Autosar lingo, see above; also for required Doxygen annotations) and treat all other ones (*"unchecked"* in Autosar) as unrecoverable that terminate (akin to Herb's suggestion: those are bugs and should be treated as such). Limitation to "checked" ones reduces invisible execution paths.

To group: What are we looking for? Just better tool support to visualize exceptional paths?

- ***Tools*** *(Issues: measuring code coverage, call tree generators, any other issues?)*

  To group: Any thoughts for options for formal or semi-formal validation (of what)?

  Error codes result in much boilerplate code; not better than exceptions

  gcov/lcov: can measure exceptional branches → improve: clearly outline which branches are due to exceptions.

  Error codes are not well-defined and not statically typed/safe ("false" vs "1", "2", …)

- ***WCET*** *(Issue: stack unwinding calls destructors)*
  - This doesn't appear to be a new issue and is akin to error code return, right? Maybe more severe depending on how exceptions are caught and if catch clauses throw further exceptions or rethrow.

    Herb: *"With error codes we have static tools to give upper bound on time." -- which?*

  To group: Is this a tooling question (evaluate all possible throws and their WCET)?

  To group: what is required from OS or compiler to support WCET argumentation?

Stephanie: Statistical argument alone not enough; include uncertainty (variance in timing?)

TODO: *update this document; get in contact with vendors (e.g., through committee) [via corresponding mailing lists] (Bryce help)*

Andreas: several compiler vendors working on ASIL-B (also D?) stdlibs. QNX (ASIL-B libc++), others unknown (libraries).

In summary, there appear to be two paths forward for today's embedded C++:

1. Ban exceptions (and follow gcc/clang instructions to *make sure* they abort), use (Boost) Outcome (as Microsoft IOT does)
2. Use static exceptions, fix string usage inside std::exception, disable RTTI (gcc/clang will still retain RTTI support for catch clause), *"assess"* RTTI (catch clause) execution time via tools (e.g., under assumption that only static libs are used, see above) and draw up some WCET argument based on timing measurements

To group: Is #2 feasible with today's tools and tech.? If not, what is missing?

To group: For the automotive folks, does #2 with the "measured", rather than "guaranteed" WCET support ASIL-B (at least) versus ASIL-D?

Other topics (later session):

- Memory allocator that meets Adaptive Autosar C++14 requirements:
  - **Different pool for different types sufficient to guarantee no fragmentation?**