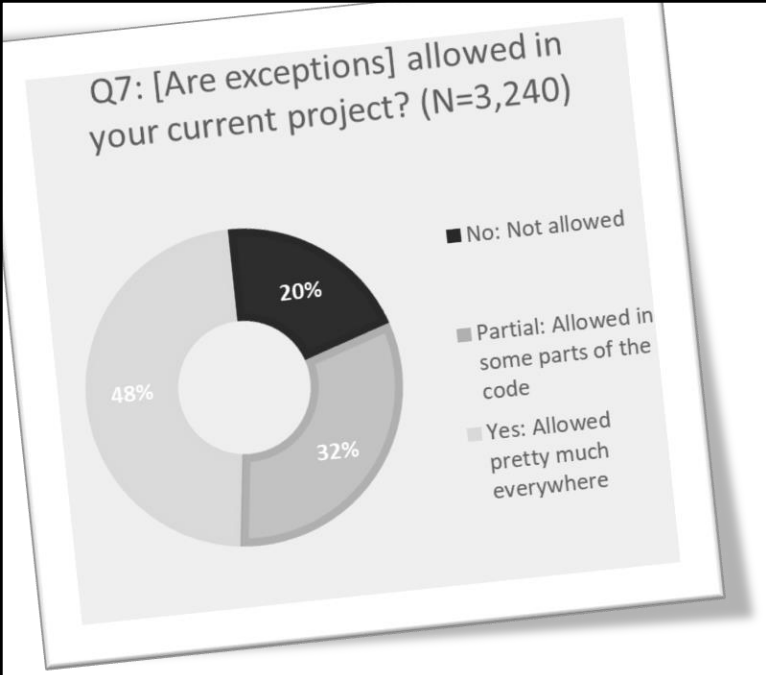


De-fragmenting C++

Making exceptions more affordable and usable

Herb Sutter

Acknowledgments: Thanks for feedback!

- JF Bastien
- Charley Bay
- Vicente Botet
- Paul Bendixen
- Jonathan Caves
- Alex Christensen
- Ben Craig
- Pavel Curtis
- Guy Davidson
- Gabriel Dos Reis
- Niall Douglas
- Chris Guzak
- Howard Hinnant
- Odin Holmes
- Ben Kuhn
- Stephan T. Lavavej
- Phil Nash
- Gor Nishanov
- Michael Novak
- Arthur O'Dwyer
- Andreas Pokorny
- Ryan Shepherd
- Bjarne Stroustrup
- Tony Tye
- Tony Van Eerd
- Ville Voutilainen
- Titus Winters
- Michael Wong

Code review…

```
status_code [redacted](widget& a, gadget& b) {
    ...
    if (!process(a)) return widget_error();
    if (!dbwrite(b)) throw db_exception();
    ...
    return good_result();
}
```

- Q: What do you think of this code?

Pathology 101

```
status_code pathological(widget& a, gadget& b) {
    ...
    if (!process(a)) return widget_error();
    if (!dbwrite(b)) throw db_exception();
    ...
    return good_result();
}
```

- Q: What do you think of this code?
  - **A: “Pick a lane!”**
- Q2: What’s harder than getting callers to do decent error handling?
  - A2: Getting them to do it **twice**, two different ways.

Pathology 101

- But this is “normal” in today’s bifurcated world:

Library A

```
status_code process(widget& a) {
    if (...) return widget_error();
    ...
}
```

Library B

```
void process(gadget& b) {
    if (...) throw db_exception();
    ...
}
```

- “Pity the poor call site that uses A and B” – including all generic code:

```
template<typename T>
auto some_func(T& t) {
    … process(t) …     // ??? error handling ???
}
```

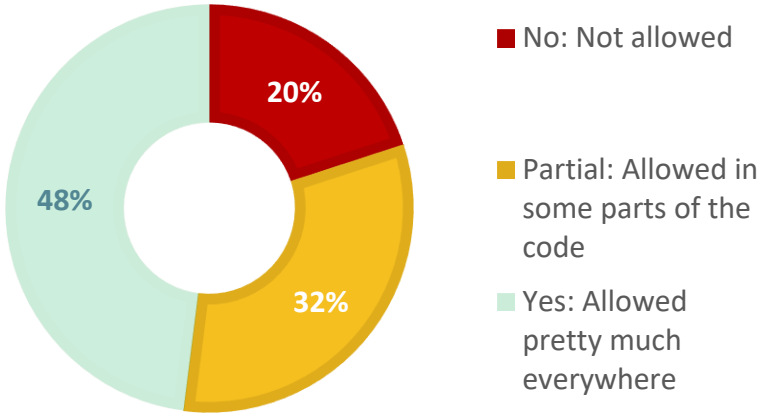
Roadmap

- Establishing the problem: Today's EH violates the zero-overhead principle
  - "I can't afford to enable exception handling" ⇒ paying for what you don't use
  - "I can't afford to throw an exception" ⇒ can write it more efficiently by hand
  - Bonus "I can't throw through this code" ⇒ lack of control, invisible vs. automatic propagation
- Key definition: What is an "error"?
  - Recoverable error != programming bug != abstract machine corruption
  - Exceptions/codes != pre/post contracts != stack and heap overflow
- 4 coordinated proposals
  1. Enable zero-overhead exception handling
  2&3. Throw fewer exceptions (~95% of all exceptions should not be)
  4. Support explicit "try" for visible propagation

isocpp.org 2018-02 survey

- **Most** "C++" projects ban exceptions in whole or in part.
  - ⇒ Not really using Standard C++, which requires exceptions.
  - Using a divergent incompatible language dialect with different idioms (e.g., factory functions instead of constructors).
  - Using a divergent incompatible std:: library dialect (e.g., EASTL, _HAS_EXCEPTIONS=0), or none at all (e.g., Epic).

Q7: [Are exceptions] allowed in your current project? (N=3,240)

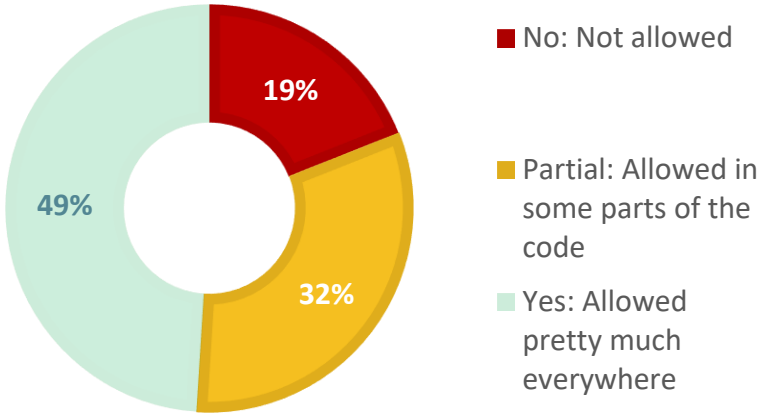


N = 3,240 of ~5.5MM: 1 in ~1,700
Norm for US national polls: 1 in 200,000

Microsoft 2018-09 survey

- **Most** "C++" projects ban exceptions in whole or in part.
  - ⇒ Not really using Standard C++, which requires exceptions.
  - Using a divergent incompatible language dialect with different idioms (e.g., factory functions instead of constructors).
  - Using a divergent incompatible std:: library dialect (e.g., EASTL, _HAS_EXCEPTIONS=0), or none at all (e.g., Epic).

Q: [Are exceptions] allowed in your current project? (N=437)



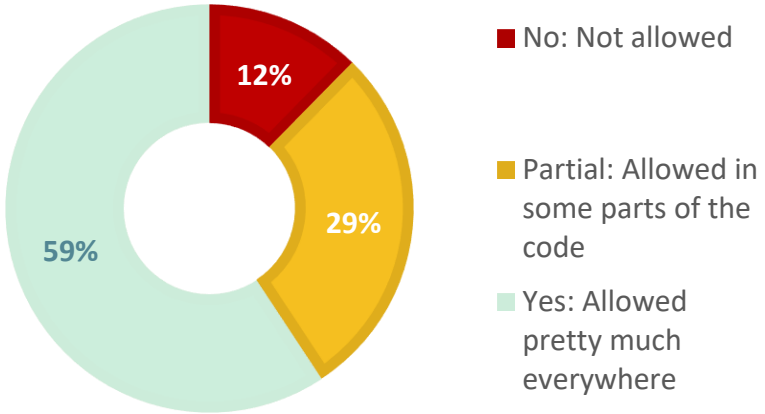
N = 437 of ~5.5MM: 1 in ~12,600

Norm for US national polls: 1 in 200,000

ACCU 2019-04 survey ☺

- **Most** "C++" projects ban exceptions in whole or in part.
  - ⇒ Not really using Standard C++, which requires exceptions.
  - Using a divergent incompatible language dialect with different idioms (e.g., factory functions instead of constructors).
  - Using a divergent incompatible std:: library dialect (e.g., EASTL, _HAS_EXCEPTIONS=0), or none at all (e.g., Epic).

Q: [Are exceptions] allowed in your current project? (N=81)

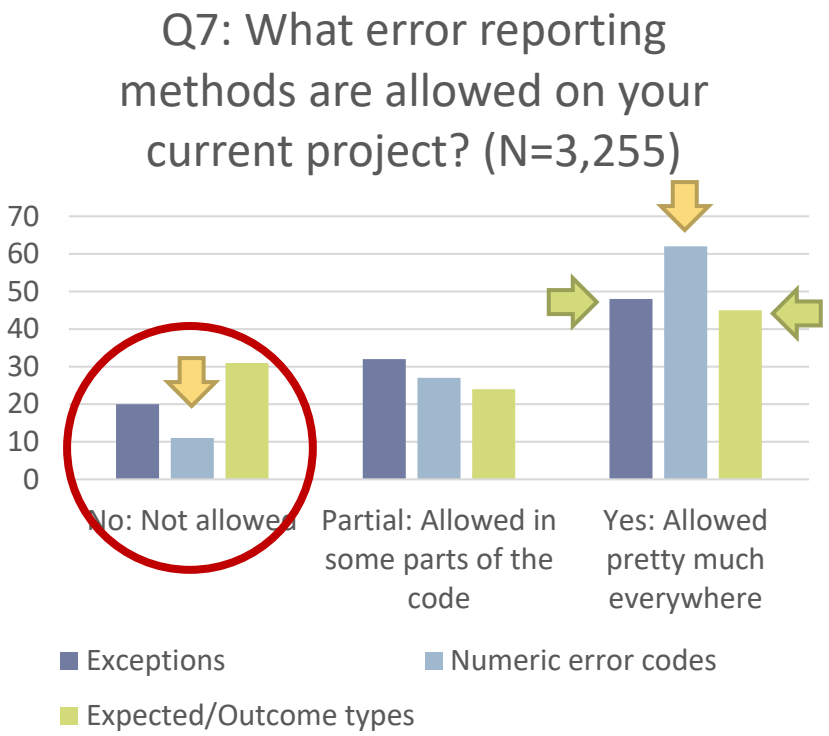


N = 81 of ~5.5MM: 1 in ~67,900

Norm for US national polls: 1 in 200,000

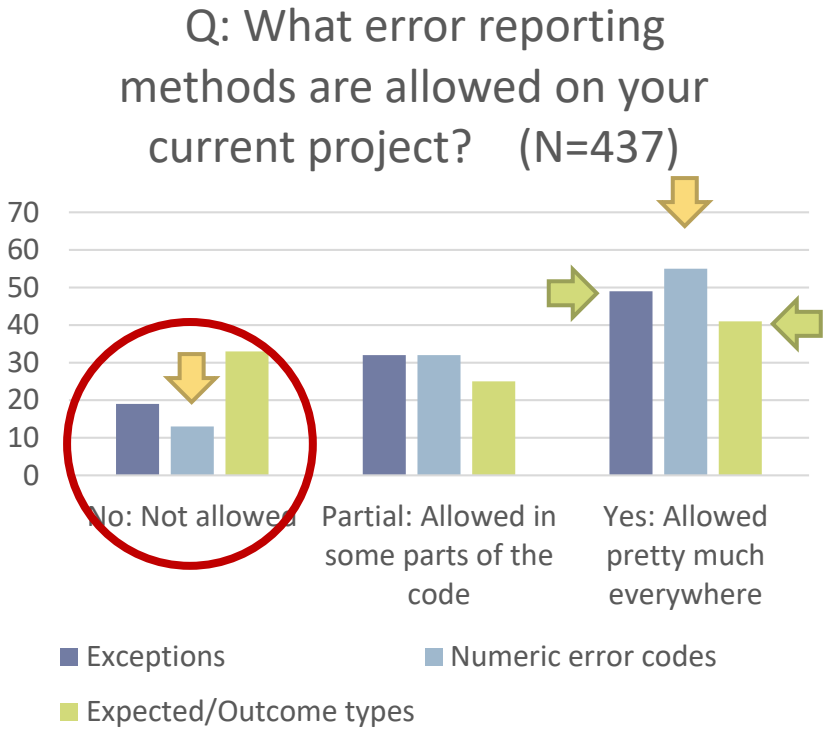
Fragmentation: isocpp.org 2018-02 survey

- **Error codes** have strongest support of any error reporting method.
- **Expected/Outcome types** are “allowed everywhere” almost equally to exceptions.
- **Every method is banned** outright in >10% of projects.
  - A measure of fragmentation into dialects.



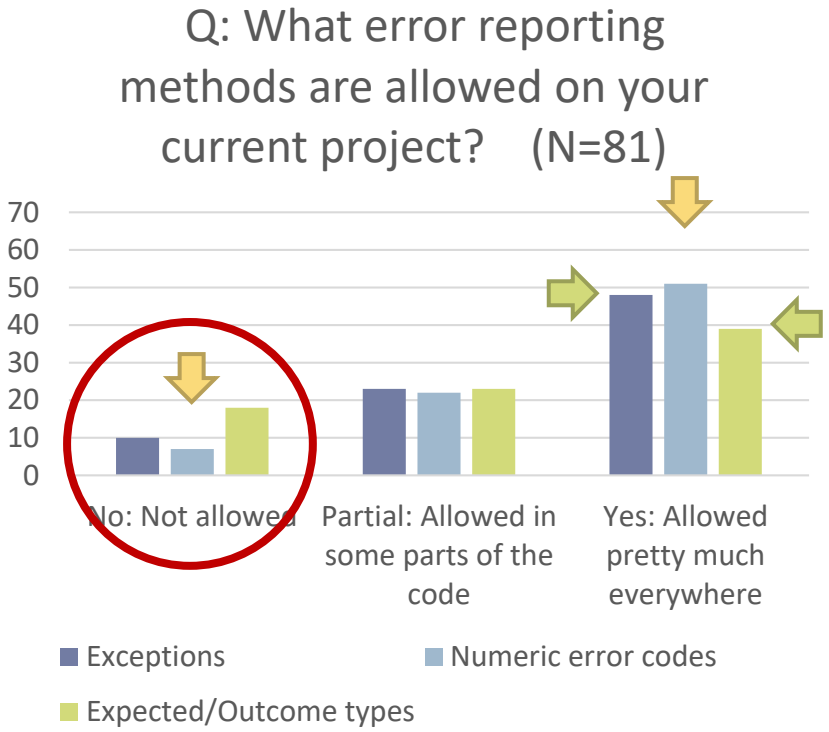
Fragmentation: Microsoft 2018-09 survey

- **Error codes** have strongest support of any error reporting method.
- **Expected/Outcome types** are “allowed everywhere” almost equally to exceptions.
- **Every method is banned** outright in >10% of projects.
  - A measure of fragmentation into dialects.



Fragmentation: ACCU 2019-04 survey ☺

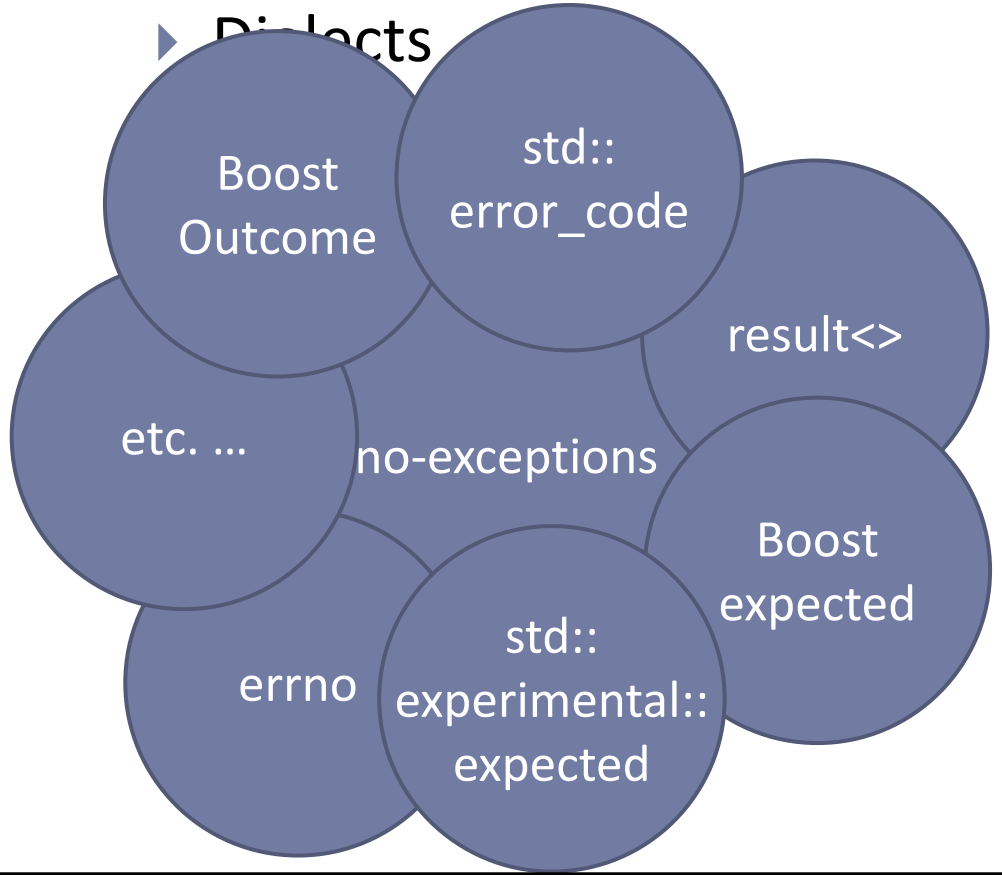
- **Error codes** have strongest support of any error reporting method.
- **Expected/Outcome types** are “allowed everywhere” almost equally to exceptions.
- **Every method is banned** outright in >10% of projects.
  - A measure of fragmentation into dialects.



Divided, we fall

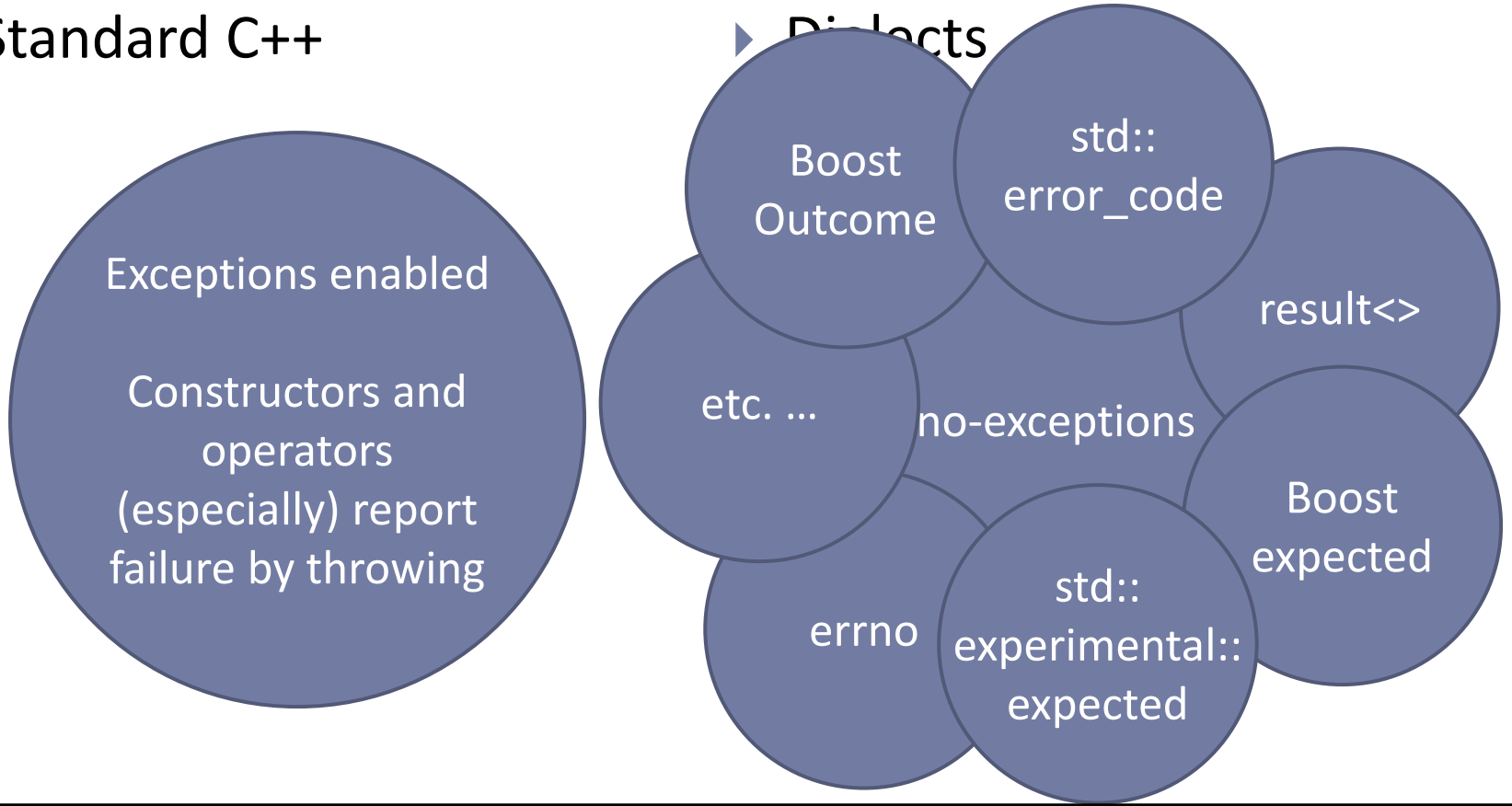
- Standard C++

- Dialects



Q: Which of these does std::filesystem use?

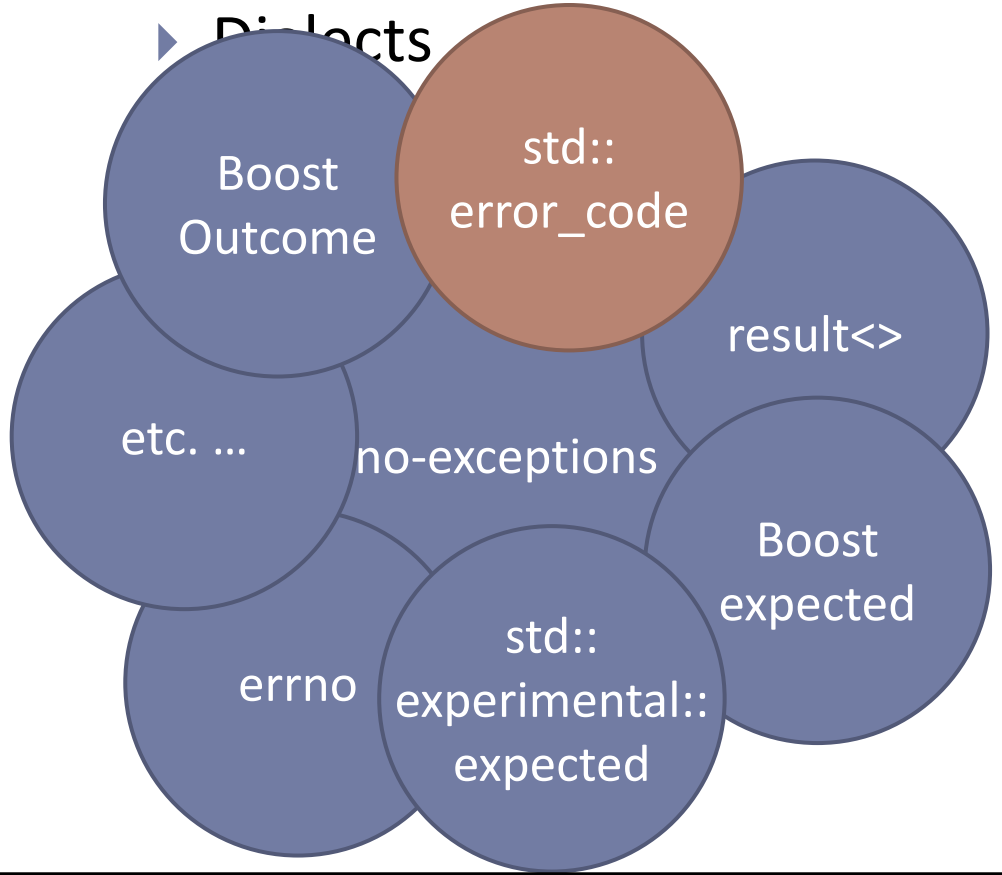
- Standard C++
- Dialects



Q: Which of these does std::filesystem use?

- Standard C++

- Dialects



A funny thing happened on the way to breakfast…

Today @6:45am:

Me: “So, what do you work on?”

Björn Fahller: “High-end embedded systems. Not the ones with constrained memory, but network switches, that sort of thing.”

Me: “Cool. …

A funny thing happened on the way to breakfast…

Today @6:45am:

Me: “So, what do you work on?”

Björn Fahller: “High-end embedded systems. Not the ones with constrained memory, but network switches, that sort of thing.”

Me: “Cool. Say, on your current project, are exceptions enab—”

BF: “No.”

Me: “—ed? …

A funny thing happened on the way to breakfast…

Today @6:45am:

Me: “So, what do you work on?”

Björn Fahller: “High-end embedded systems. Not the ones with constrained memory, but network switches, that sort of thing.”

Me: “Cool. Say, on your current project, are exceptions enab—”

BF: “No.”

Me: “—ed? … Oh. So, do you use the standard library?”

BF: “No. Well, we cheat. Algorithms don’t throw…”

Root cause: Today's EH not "zero-overhead"

- Violates C++'s zero-overhead principle in two ways.

  1. "I can't afford to **enable** exception handling."
     - **Just turning on EH incurs space overhead.**
     - Zero overhead principle, part 1: "Don't pay for what you don't use."
  2. "I can't afford to **throw** an exception."
     - **Throwing an exception incurs not-statically-boundable space and time overhead.**
     - **Throwing an exception usually less efficient than returning code/expected<> by hand.**
     - Zero overhead principle, part 2: "When you do use it you can't reasonably write it better by hand" including by using alternatives.

- Bonus problem: "I can't throw through **this** code."
  - **Lack of control:** Automatic propagation is great, but **invisible** control flow makes writing exception-safe code harder. More on this later…

Roadmap

- Establishing the problem: Today's EH violates the zero-overhead principle
  - "I can't afford to enable exception handling" ⇒ paying for what you don't use
  - "I can't afford to throw an exception" ⇒ can write it more efficiently by hand

  Bonus "I can't throw through this code" ⇒ lack of control, invisible vs. automatic propagation
- Key definition: What is an "error"?
  - Recoverable error != programming bug != abstract machine corruption
  - Exceptions/codes != pre/post contracts != stack and heap overflow
- 4 coordinated proposals
  1. Enable zero-overhead exception handling
  2&3. Throw fewer exceptions (~95% of all exceptions should not be)
  4. Support explicit "try" for visible propagation

Program-recoverable errors

***error**: "an act that … fails to achieve what should be done."*

— [Merriam-Webster]

- P0709: "recoverable error" ≡ "a function couldn't do what it advertised."
  - Its preconditions were met.
  - It could not achieve its successful-return postconditions.
  - The calling code can be told and can programmatically recover.

- Errors (and only errors) should be reported to the calling **code**.
  - Regardless of mechanism: "Prefer exceptions" but applies to any reporting style.

Abstract machine corruption ≠ recoverable error

- Abstract machine corruption causes a corrupted state that cannot be recovered from programmatically.
  - So it should never be reported to calling code as an error (e.g., via exception).

- Example: **Stack exhaustion** is always an abstract machine corruption.
  - It can happen to any function.
    ⇒ If we tried to report it using an exception then every function could throw.
  - We cannot continue running normal code. (NB: Destructors are "normal code.")
    ⇒ The callee can't run code to report it to the caller…
    … and the caller couldn't run code to recover anyway.
  - Conclusion: Reporting this as a runtime error would be a category error.

Programming bug ≠ recoverable error

- A programming bug (e.g., out-of-bounds access, null dereference) causes a corrupted state that cannot be recovered from programmatically.
  - Therefore it should never be reported to the calling code as an error (e.g., it should not be reported via an exception).
- Examples:
  - A **precondition** (e.g., [[pre...]]) violation is always a bug in the caller (it shouldn't make the call).
    - Corollary: std::logic_error and its derivatives should never be thrown (§4.2), its existence is itself a "logic error"; use assertions/contracts/… instead.
  - A **postcondition** (e.g., [[post...]]) violation on "success" return is always a bug in the callee (it shouldn't return success).
    - Violating a noexcept declaration is also a form of postcondition violation.
  - An **assertion** (e.g., [[assert...]]) failure is always a bug in the function.

Taxonomy

	What to use	Report-to handler	Handler species
A. Corruption of the abstract machine (e.g., stack exhaustion)	Terminate	User	Human
B. Programming bug (e.g., precondition violation)	Asserts, log checks, contracts, ...	Programmer	Human
C. Recoverable error (e.g., host not found)	Throw exception, error code, etc.	Calling code	Code

Roadmap

- Establishing the problem: Today's EH violates the zero-overhead principle
  - "I can't afford to enable exception handling" ⇒ paying for what you don't use
  - "I can't afford to throw an exception" ⇒ can write it more efficiently by hand
  - Bonus "I can't throw through this code" ⇒ lack of control, invisible vs. automatic propagation
- Key definition: What is an "error"?
  - Recoverable error != programming bug != abstract machine corruption
  - Exceptions/codes != pre/post contracts != stack and heap overflow
- 4 coordinated proposals
  1. Enable zero-overhead exception handling
  2&3. Throw fewer exceptions (~95% of all exceptions should not be)
  4. Support explicit "try" for visible propagation

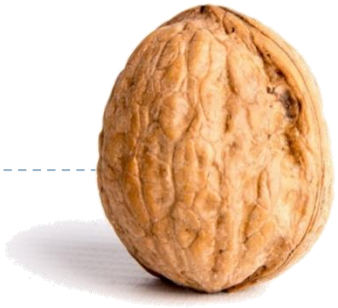
Core issues: Zero-overhead + **determinism**

- Exceptions are great: Distinct "error" paths, can't ignore, auto propagation.
  - But: Inherently not zero-overhead, not deterministic.
  - "Throwing **objects of dynamic types**… ⇒ dynamic allocation + type erasure
    … and catching **using RTTI**." ⇒ dynamic casting (special)

- Proposal:
  - "Throwing **values of static types**… ⇒ stack allocation, share return channel
    … and catching **by value**." ⇒ no dynamic casting, just value comparison
  - Isomorphic to error codes, identical space/time overhead and **predictability**.
  - Share return channel ⇒ potential for negative overhead abstraction.
  - If a function agrees (**opts in**) that any exceptions it emits are **values of one statically known type**, we can implement it with zero dynamic/non-local overheads.

not a breaking change

1. Throw values, not types



- As-if returning **union{ Success; Error; } + bool**, using the same return channel (incl. registers + CPU flag for discriminant).
  - Best of exceptions and error codes (and fully prior-art):
    Exactly exceptions' programming model (throw, try, catch).
    Exactly error codes' return-value implementation (w/o monopolizing channel).
  - Doubles down on **value semantics**. (Cf: C++11 move semantics.)
- If you love:
  - Exceptions: Can use them more widely, removing perf reasons to avoid/ban.
  - Expected/Outcome: Gets language support, propagates automatically.
  - Error codes: Doesn't monopolize return channel, propagates automatically, and the caller can't forget to check it and gets distinct success/error paths.
  - Termination (fail-fast): Hook the propagation notification (see §4.1.4).

Core proposal summary

- A *static-exception-specification* **throws** ⇒ function can throw **std::error**,
  an evolution of std::error_code + SG14-driven improvements already underway.

```
string f() throws {
   if (flip_a_coin()) throw arithmetic_error::something;
   return “xyzzy”s + “plover”;                    // bad_alloc → std::errc::ENOMEM
}
string g() throws { return f() + “plugh”; }       // bad_alloc → std::errc::ENOMEM
int main() {
   try {
      auto result = g();
      cout << “success, result is: ” << result;
   } catch(error err) {                           // catch by value
      cout << “failed, error is: ” << err.error();
   }
}
```

Core proposal summary

- A *static-exception-specification* **throws** ⇒ function can throw **std::error**,
  an evolution of std::error_code + SG14-driven improvements already underway.

```
string f() throw
   if (flip_a_coi
   return "xyzzy
}
string g() throw
int main() {
   try {
      auto result
      cout << "su
   } catch(error
      cout << "fa
   }
}
```

Default and recommended std::error usage == purely local return values:

- Always allocated as an ordinary stack value
- Share (not waste) the return channel
- Statically known type, so never need RTTI

Zero-overhead: No extra static overhead in the binary image.
No dynamic allocation. No need for RTTI.

Determinism: Identical space and time cost as if returning an error code by hand.

Note: For compatibility, std::error can also wrap an exception_ptr, but this is a compatibility mode where the overheads come from using today's model, which are just passed through

Dynamic type(-*erased*) vs. static type

Today (pseudocode)

```
// throw site: “throw MyException(value)”
return (void*) new MyException(value);


// …
// propagate
// …


// catch site: “catch (EBase& e) {/*…*/}” by reference
if (auto e = special_dynamic_cast<EBase*>(pvoid); e)
   {/*…*/}
```

Proposed (pseudocode)

```
// throw site: “throw std::error(domain,value)”
return std::error(domain,value);      // no alloc


// …
// propagate
// …


// catch site: “catch (std::error e)” by value
if (e.failed) {/*…*/}                 // no RTTI
```

1. Simplifications

- What are the benefits?
  - Unification: All projects can turn on exception handling.
    - Zero overhead principle, part 1: “Don’t pay for what you don’t use.”
  - Unification: All code can report errors using exceptions.
    - Zero overhead principle, part 2: “When you do use it you can’t reasonably write it better by hand” including by using alternatives.
    - Even space- and time-constrained code that need statically boundable costs.
  - **Simplification:** Can teach “every function should be declared with exactly one of **noexcept** or **throws**.”
    - Just like we now can teach “every virtual function should be declared with exactly one of **virtual**, **override**, or **final**.”

Roadmap

- Establishing the problem: Today's EH violates the zero-overhead principle
  - "I can't afford to enable exception handling" ⇒ paying for what you don't use
  - "I can't afford to throw an exception" ⇒ can write it more efficiently by hand

  Bonus "I can't throw through this code" ⇒ lack of control, invisible vs. automatic propagation
- Key definition: What is an "error"?
  - Recoverable error != programming bug != abstract machine corruption
  - Exceptions/codes != pre/post contracts != stack and heap overflow
- 4 coordinated proposals
  1. Enable zero-overhead exception handling

  2&3. Throw fewer exceptions (~95% of all exceptions should not be)

  4. Support explicit "try" for visible propagation

Language-independent fact

"[With contracts,] 90-something% of the typical uses of exceptions in .NET and Java became preconditions. All of the ArgumentNullException, ArgumentOutOfRangeException, and related types and, more importantly, the manual checks and throws were gone."

— [Duffy 2016]

2. Bugs ⇒ contracts

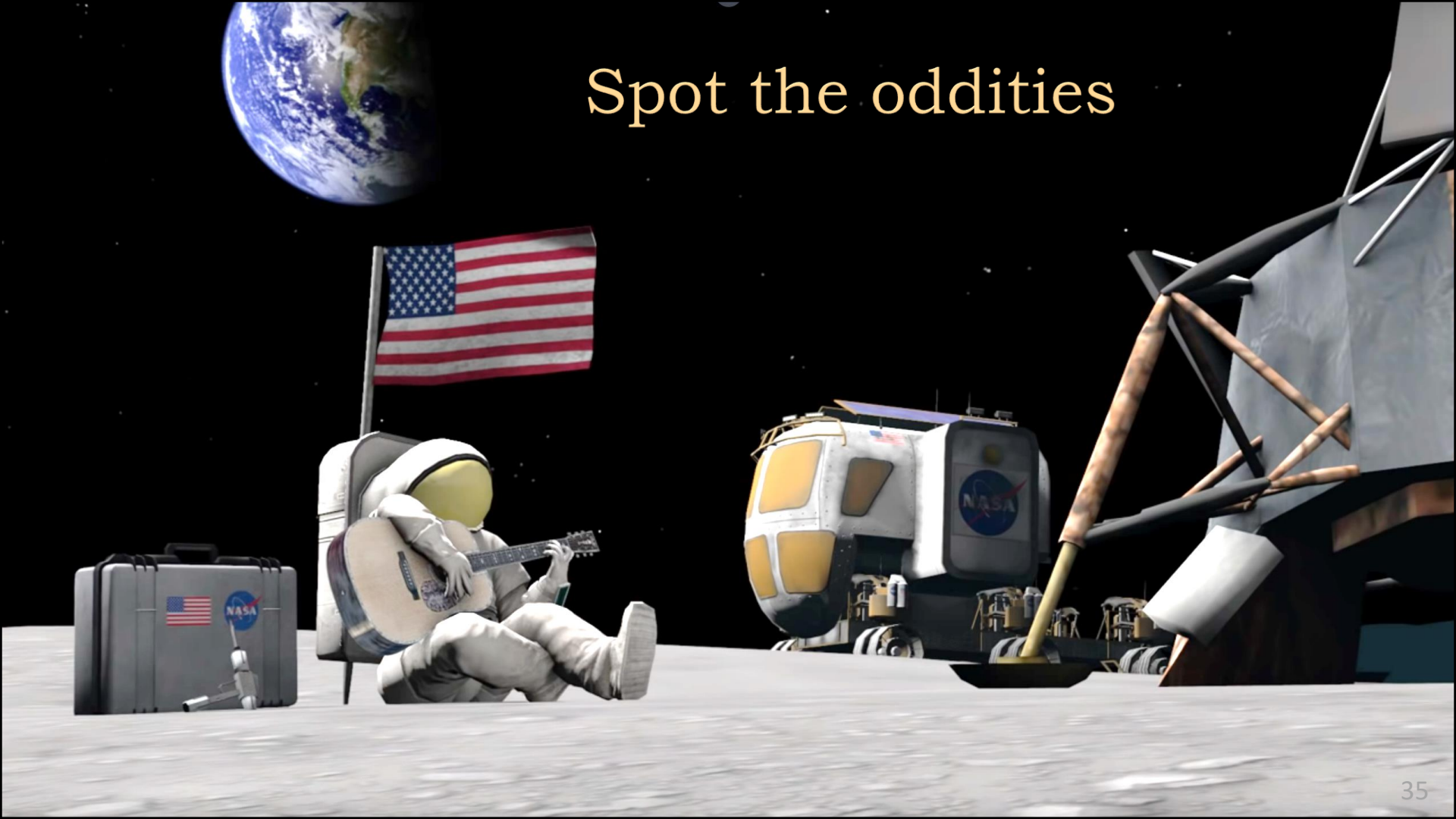
Summary

- Precondition violations are bugs, not program-recoverable errors
- Don't report them using error handling (exceptions or codes)
  - Calling code can't recover programmatically
  - Shared state must already be presumed corrupt
- Use assertions, contracts, or similar instead
  - Report to a human programmer who can fix the bug

Status / Proposal

- WG21:
  - Supported by standard library maintainers
  - Migration planned to move *logic_error* & derived types to not be exceptions
    - When used as preconditions
    - Multi-release migration period

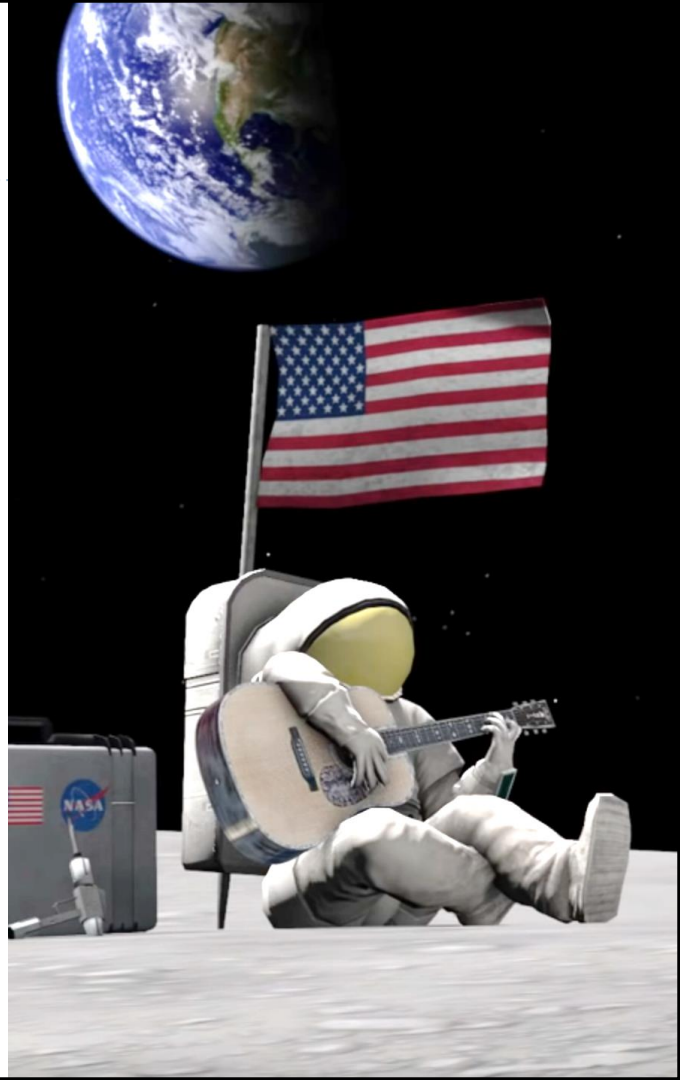
Spot the oddities



Spot the oddities

- Today:
  - 1. Exceptions must be dynamically allocated.
  - 3. Dynamic allocation failures are reported using exceptions.

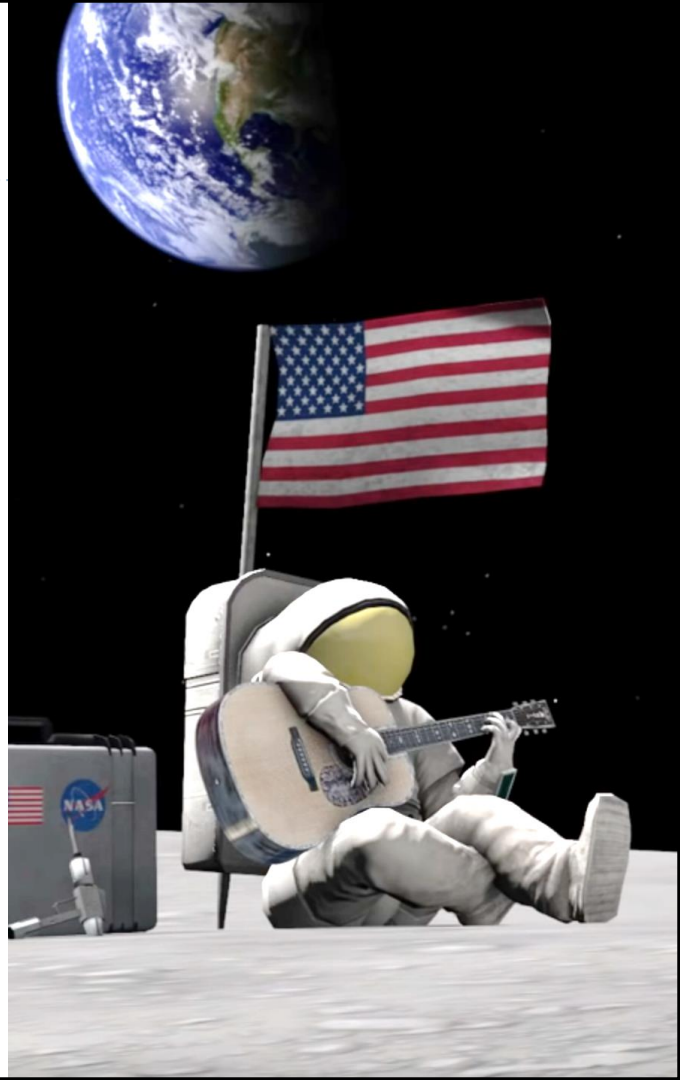
- **Q: How does this statement describe two independent issues?**
  - 1. (see prev) Exceptions shouldn't need to be dynamically allocated.
  - 3. (see next) Allocation failures shouldn't be reported as program-recoverable errors (exceptions or otherwise)…



Spot the oddities

- Today:
  - ~~1. Exceptions must be dynamically allocated.~~
  - 3. Dynamic allocation failures are reported using exceptions.

- **Q: How does this statement describe two independent issues?**
  - ~~1. (see prev) Exceptions shouldn't need to be dynamically allocated.~~
  - 3. (see next) Allocation failures shouldn't be reported as program-recoverable errors (exceptions or otherwise)…



Key Q: Can I continue running “normal code”?

*Q: If I hit **stack overflow**, can I continue running ordinary code?*

*A: **No.***

We have exhausted/corrupted the abstract machine.

Key Q: Can I continue running “normal code”?

*Q: If I hit a **memory allocation failure**, can I continue running ordinary code?*

*A: **No, if it’s a “small” allocation failure…***

We have exhausted/corrupted the abstract machine.

Language-independent fact

- Many heap allocation failures (aka out of memory, OOM) are unrecoverable
  - Appears to be inherent: e.g., impossible to thoroughly test, or must be written carefully
  - But some current/future code is OOM-safe, and we don't want to lose that

- Case 1: Failure to allocate big buffer || opt-in allocator (e.g., ***new[100000]***, ***MyAlloc***)
  - Causes: Optimistic or unsanitized size input
  - **Unwinding+recovering by running "normal code" is possible: Throwing/returning is OK**
  - Recovery possible: Fall back to smaller buffer, or fail requested operation
- Case 2: Failure to perform "small" allocation && default allocator (e.g., ***new int***)
  - Cause: Resource limit (actual exhaustion or fragmentation): **So like stack exhaustion**
  - **Unwinding+recovering) by running "normal code" not possible: Throwing/returning not OK**
  - ⇒ By default: Don't throw, terminate (with *terminate_handler* support to opt out)

3. OOM ⇒ terminate (99%) + harden (1%)

Summary

- "Small" allocation failure:
  - Not testable: Too pervasive
  - Nonportable: Requires OS-specific settings to enable on common OSes
  - Not unwindable: Can't run "normal code"
- "Large" failure can write fallback:
  - Texture load, big work buffer…
- Proposal
  - For "99%": Terminate by default, treat same as abstract machine failure
  - For "1%": Use new(nothrow) + provide try_reserve/…

Status / Proposal

- WG21:
  - Change default new_handler from "throw bad_alloc" to "terminate"
  - Groundswell of support, but some opposition
- What to do:
  - Texture load, big buffer unavailable: Explicitly test, implement fallback (e.g., don't show texture, use smaller buffer)
  - File|Open: Do all the work off to the side in an isolated arena, commit using nofail operations only

Taxonomy

	What to use	Report-to handler	Handler species
A. Corruption of the abstract machine (e.g., stack **or heap** exhaustion)	Terminate	User	Human
B. Programming bug (e.g., precondition violation)	Asserts, log checks, contracts, ...	Programmer	Human
C. Recoverable error (e.g., host not found, **large allocation failure**)	Throw exception, error code, etc.	Calling code	Code

2 & 3. Simplifications

- What are the benefits?
  - Correctness: Exceptions are not appropriate for reporting non-errors.
    - Bugs (e.g., preconditions) and corruption (e.g., abstract machine failures).
  - Correctness and performance: Eliminate ~95% of all exceptions.
    - The vast majority of the standard library would not throw.
    - (Recall: Language-independent. Also true of Java and C#.)
  - **Simplification:** Eliminate ~95% of the *invisible* control flow paths.
    - (Which today dominate the visible ones.)
    - Clear code is easier to write correctly and reason about.
    - Example: See GotW #20, a 4-line function with 3 normal (and visible) control flow paths and *20 exceptional (and invisible) control flow paths*.

Roadmap

- Establishing the problem: Today's EH violates the zero-overhead principle
  - "I can't afford to enable exception handling" ⇒ paying for what you don't use
  - "I can't afford to throw an exception" ⇒ can write it more efficiently by hand
  - Bonus "I can't throw through this code" ⇒ lack of control, invisible vs. automatic propagation
- Key definition: What is an "error"?
  - Recoverable error != programming bug != abstract machine corruption
  - Exceptions/codes != pre/post contracts != stack and heap overflow
- 4 coordinated proposals
  1. Enable zero-overhead exception handling
  2&3. Throw fewer exceptions (~95% of all exceptions should not be)
  4. Support explicit "try" for visible propagation

4. Proposed extension: *try* expressions

- Good news: Exceptional control flow is **automatic**.
- Bad news: Exception control flow is **invisible**.
  - Hard to reason about exceptions, especially in legacy code.
- Proposal: **try** before an expression/statement where a subexpression can throw.
  - Makes exceptional paths visible.
  - If we required it in new code: **Compile-time guarantees** (e.g., no "throw" ⇒ noexcept).

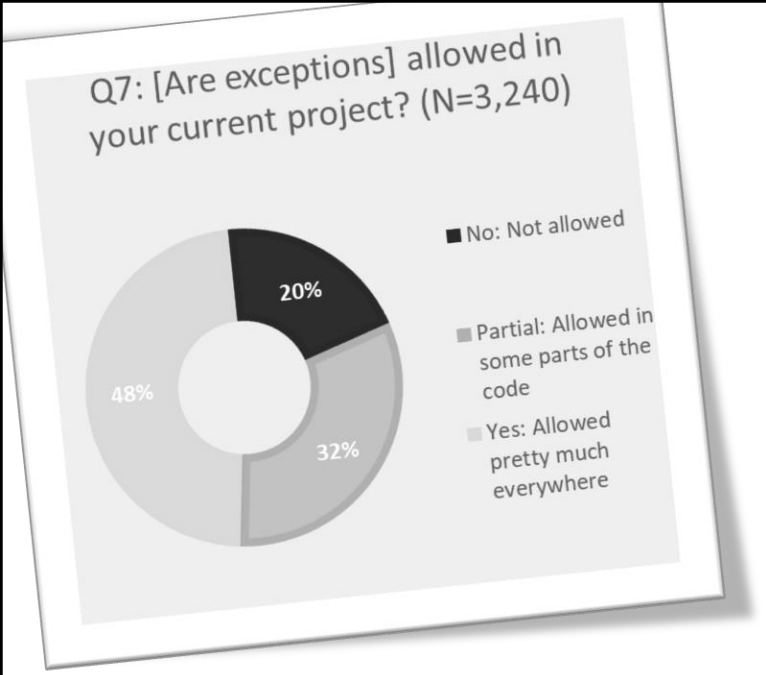
```
string f() throws {
   if (flip_a_coin()) throw arithmetic_error::something;
   return try "xyzzy"s + "plover";                    // greppable

   try string s("xyzzy");                             // same, just showing statement form too
   try return s + "plover";
}

string g() throws { return try f() + "plugh"; }
```

4. Simplifications

- What are the benefits?
  - Convenience (as today): **Automatic** exception propagation.
  - Correctness (new): **Visible** (still convenient) propagation.

1+2+3+4. Simplifications

- "One more thing"… Sets the stage for a potential new world:
  - 1: Enables "declare every function either noexcept or throws."
  - 2+3: Enables "~95% of all functions are noexcept."
  - 1+2+3+4: Enables "require **try** on every expression that can throw."
  - **Simplification:** Enables using C code in C++ projects with confidence.
    - Can take any C code, compile it as C++, and (automatically) add **try** on every expression that could throw ⇒ feasible to inspect and validate the code is exception-safe.



De-fragmenting C++

Making exceptions more affordable and usable

Herb Sutter