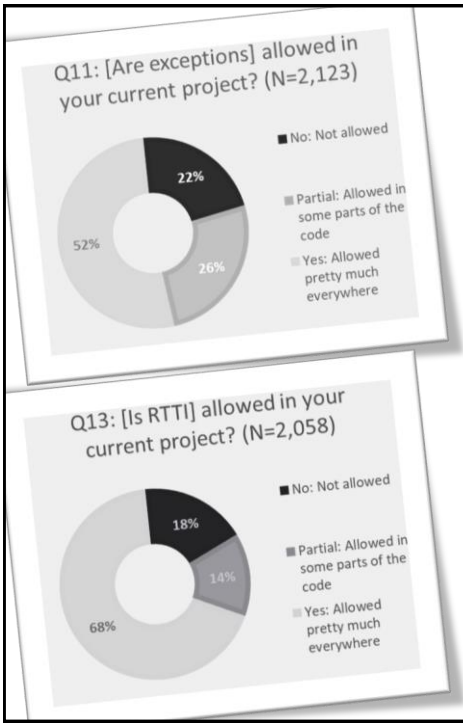


De-fragmenting C++

Making **exceptions** and **RTTI** more affordable and usable

Herb Sutter

1

In honor of Andrei Alexandrescu

- This ***is*** a known algorithm, mentioned at a previous CppCon, just in assembler.
- Please raise your hand when you recognize it. You have 15 seconds.

```
488b11          mov     rdx, qword ptr [rcx]
4c8d05ce8f0500 lea     r8, [codegentest!MyChild1::`vftable']

; Now do the range check. Jump to the AppCompat check if the r
492bc0          sub     rdx, r8
4883f820        cmp     rdx, 20h
7715            ja      codegentest!DoCast+0x3b       ; Jump to
```

2

2

From CppCon 2018

So what *is* C++ / "C++ic"?

- Core principles:
  - **Zero-overhead abstraction** — don't pay for what you don't use, what you use is as efficient as you can reasonably write by hand
  - **Determinism & control** — over time & space, close to hardware, leave no room for a lower language, trust the programmer
  - **Link compatibility** — with C95 and C++prev
- Useful pragmatics for adoption and library consumption:
  - **Backward source compat with C** — mostly C95, read-only, consume headers & seamlessly call
  - **Backward source compat with C++prev** — C++98 and later, read-mostly, to use & to maintain
- What is *not* core C++:
  - **Specific syntax** — we've been adding & recommending C-incompatible syntax since 1979
  - **Tedium** — most modern abstractions (e.g., range-*for*, *override*) compatible with zero-overhead
  - **Lack of good defaults** — good defaults are fine, as long as the programmer can override them
  - **Sharp edges** — e.g., brittle dangling pointers are not necessary for efficiency and control

3

3

So what *is* C++ / "C++ic"?

Historical strength: **static**
typing, compilation, linking

Major current trend (IME): ↓dynamic, ↑**static**
concepts, contracts, constexpr, static error types (fs_error), virtual → templates, RTTI's typeid → reflection, …

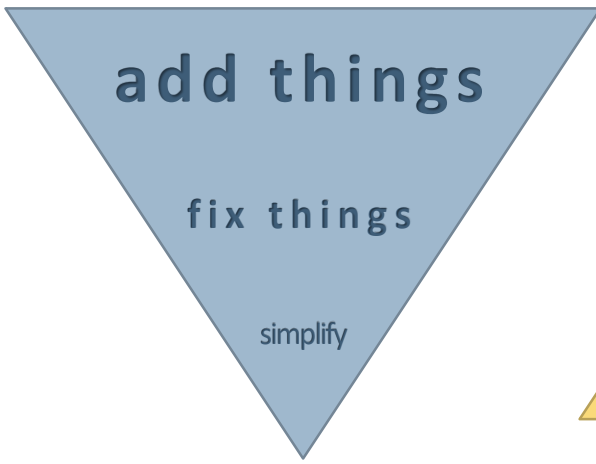
- Core principles:
  - **Zero-overhead abstraction** — don't pay for what you don't use, what you use is as efficient as you can reasonably write by hand
  - **Determinism & control** — over time & space, close to hardware, leave no room for a lower language, trust the programmer
  - **Link compatibility** — with C95 and C++prev
- Useful pragmatics for adoption and library consumption:
  - **Backward source compat with C** — mostly C95, read-only, consume headers & seamlessly call
  - **Backward source compat with C++prev** — C++98 and later, read-mostly, to use & to maintain
- What is *not* core C++:
  - **Specific syntax** — we've been adding & recommending C-incompatible syntax since 1979
  - **Tedium** — most modern abstractions (e.g., range-*for*, *override*) compatible with zero-overhead
  - **Lack of good defaults** — good defaults are fine, as long as the programmer can override them
  - **Sharp edges** — e.g., brittle dangling pointers are not necessary for efficiency and control

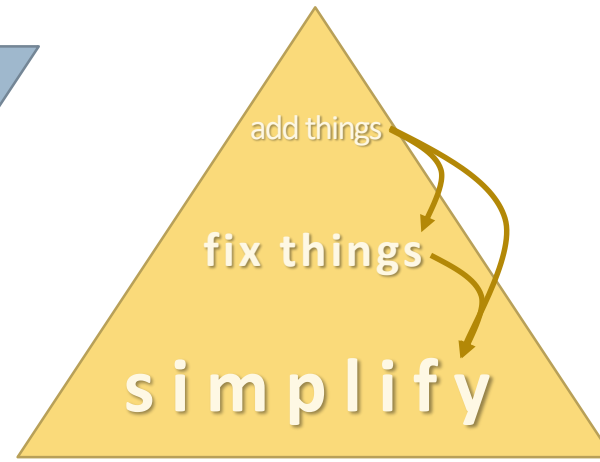
4

4

C++'s evolution priorities

Historical





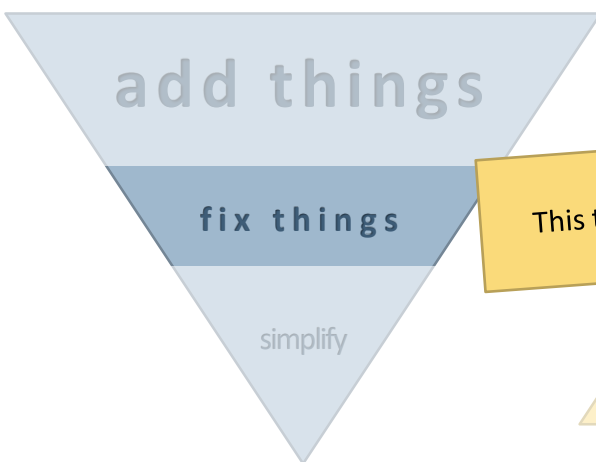
A future worth considering?

5

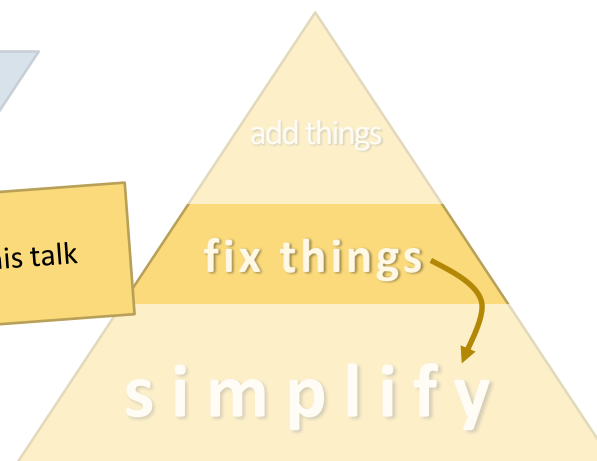
5

C++'s evolution priorities

Historical



This talk



A future worth considering?

6

6

A tale of two -compiler /switches

We 💔 exceptions

```
-fno-exceptions
#define _HAS_EXCEPTIONS 0
```

We 💔 RTTI

```
-fno-rtti
/GR-
```

7

7

A tale of two -compiler /switches

We 💔 exceptions

```
-fno-exceptions
#define _HAS_EXCEPTIONS 0
```

We 💔 RTTI

```
-fno-rtti
/GR-
```

One common cause:

"RT" == Run Time

Dynamic semantics that must be available, used or not

Spoilers

Today's EH requires **type erasure**, and today's RTTI requires binaries to **store metadata** — whether needed/used or not.

Optimizations are possible, but are about digging partway out of the hole we already jumped into.

Alternate idea: How about not digging until we need to?
⇒ **more static + opt-in**, less dynamic default/always-on?

8

8

A "pay only for what you use" principle

static by default

dynamic by opt-in

9

9

Part 1: Exception Handling (EH)

- 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 a "recoverable error"?
  - Recoverable error != programming bug != abstract machine corruption
  - Exceptions/codes != pre/post contracts != stack and heap overflow
- Four coordinated proposals
  1. Enable zero-overhead exception handling
  2&3. Throw fewer exceptions (~90% of all exceptions should not be)
  4. Support explicit "try" for visible propagation

10

10

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?

11

11

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.

12

12

Pathology 101

- But this is "normal" in today's bifurcated world:

Library A	Library B

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

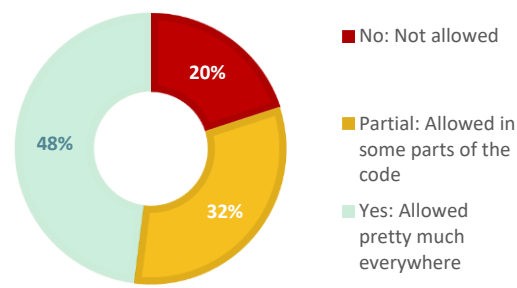
13

13

isocpp.org 2018-02 survey

- **About half** of "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

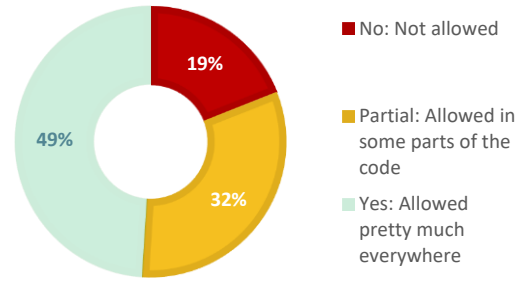
14

14

Microsoft 2018-09 survey

- **About half** of "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)



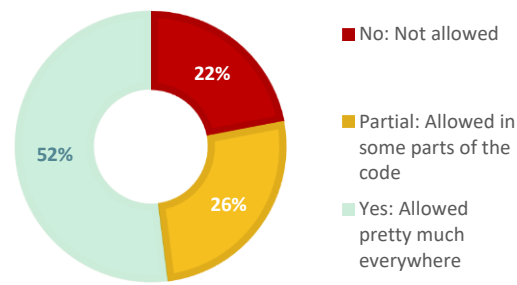
15

15

isocpp.org 2019-04 survey

- **About half** of "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).

Q11: [Are exceptions] allowed in your current project? (N=2,123)

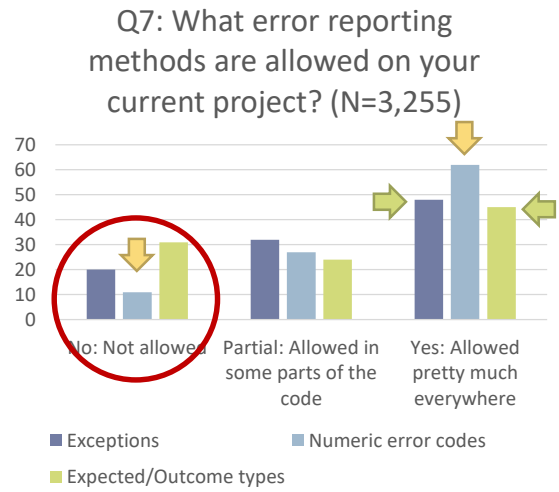


16

16

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.

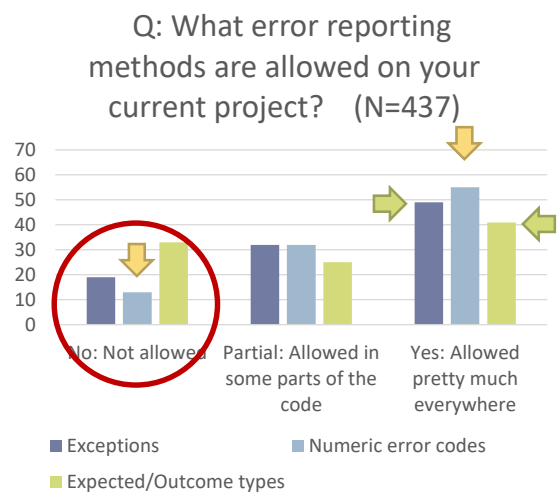


17

17

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.

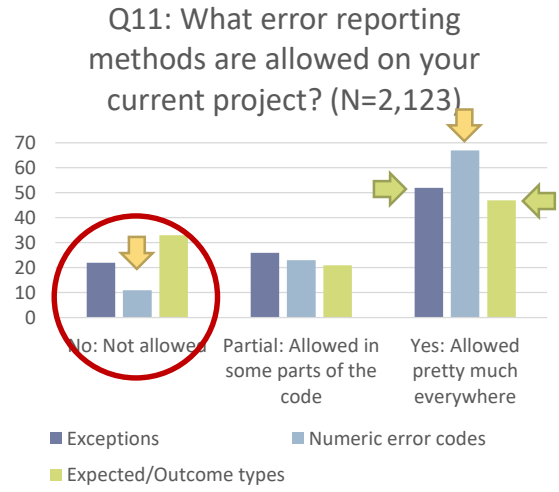


18

18

Fragmentation: isocpp.org 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.

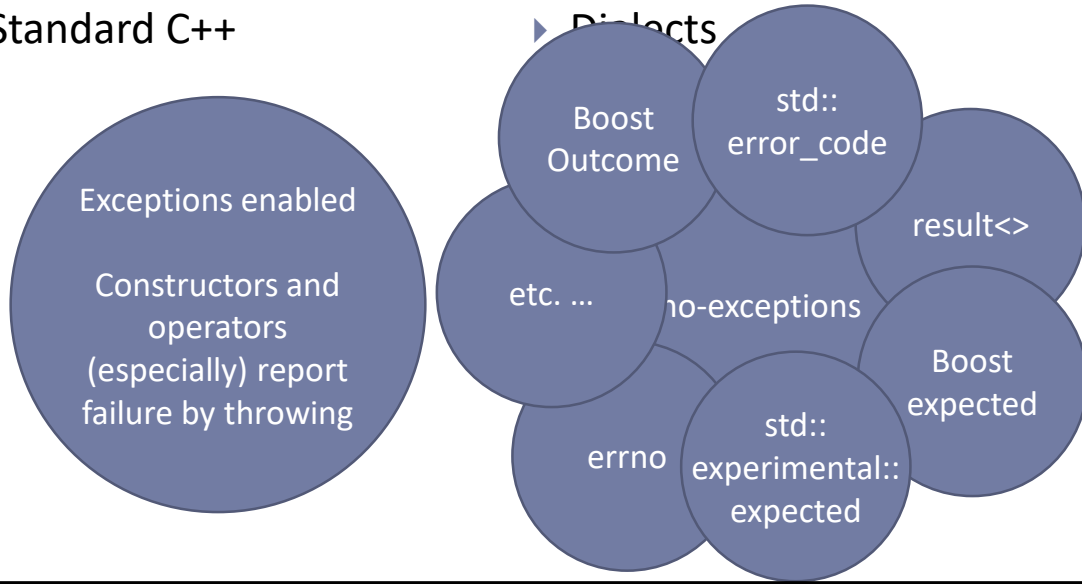


19

19

Divided, we fall

- Standard C++
- Dialects

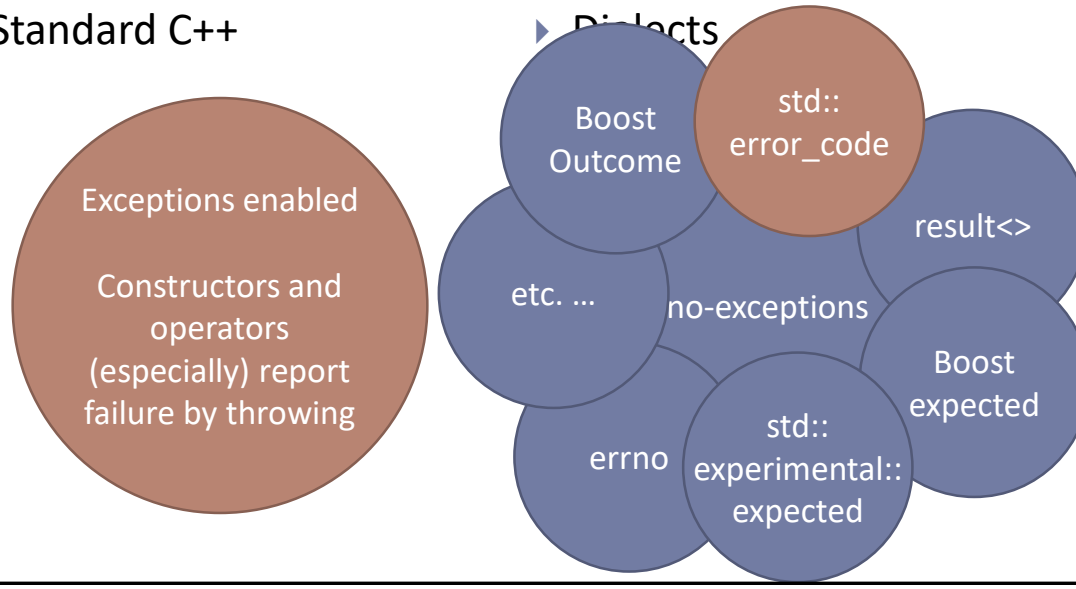


20

20

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

- Standard C++
- Dialects



21

21

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…

22

22

Part 1: Exception Handling (EH)

- 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. auto propagtn
- Key definition: What is a "recoverable error"?
  - Recoverable error != programming bug != abstract machine corruption
  - Exceptions/codes != pre/post contracts != stack and heap overflow
- Four coordinated proposals
  1. Enable zero-overhead exception handling
  2&3. Throw fewer exceptions (~90% of all exceptions should not be)
  4. Support explicit "try" for visible propagation

SF	WF	N	WA	SA
22	20	5	1	0
22	17	4	2	1
14	13	0	3	4

23

23

Part 1: Exception Handling (EH)

- 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 a "recoverable error"?
  - Recoverable error != programming bug != abstract machine corruption
  - Exceptions/codes != pre/post contracts != stack and heap overflow
- Four coordinated proposals
  1. Enable zero-overhead exception handling
  2&3. Throw fewer exceptions (~90% of all exceptions should not be)
  4. Support explicit "try" for visible propagation

24

24

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

25

25

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.

26

26

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.

27

27

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

28

28

Part 1: Exception Handling (EH)

- 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 a "recoverable error"?
  - Recoverable error != programming bug != abstract machine corruption
  - Exceptions/codes != pre/post contracts != stack and heap overflow
- Four coordinated proposals
  1. Enable zero-overhead exception handling

  2&3. Throw fewer exceptions (~90% of all exceptions should not be)

  4. Support explicit "try" for visible propagation

29

29

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

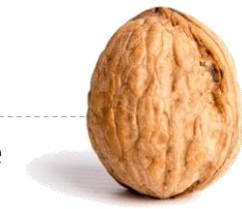
- Exceptions are great: Distinct "error" paths, can't ignore, auto propagation.
  - But: Inherently not zero-overhead, not deterministic.
  - "Throwing **objects of dynamic types**... ⇒ dynamic allocation + type erasure
    ... and catching **using RTTI**." ⇒ dynamic casting (special)
- Proposal:
  -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

30

30

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).

31

31

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();
   }
}
```

32

32

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

33

33

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
```

34

34

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**."

35

35

Part 1: Exception Handling (EH)

- 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 a "recoverable error"?
  - Recoverable error != programming bug != abstract machine corruption
  - Exceptions/codes != pre/post contracts != stack and heap overflow
- Four coordinated proposals
  1. Enable zero-overhead exception handling
  2&3. Throw fewer exceptions (~90% of all exceptions should not be)
  4. Support explicit "try" for visible propagation

36

36

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]

37

37

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

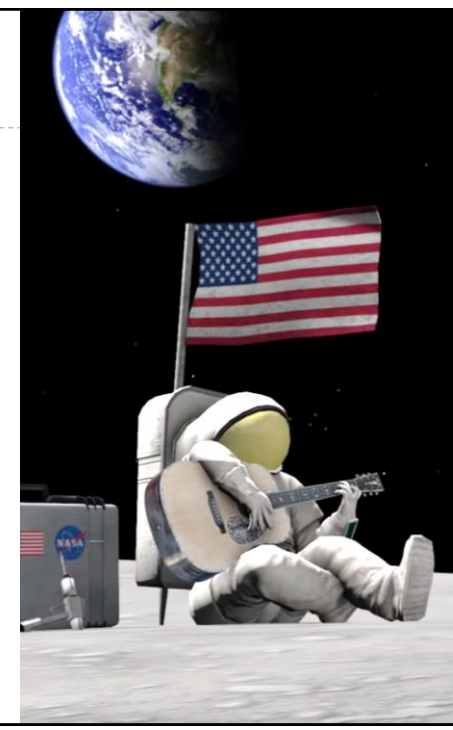
38

38

39

Spot the oddities

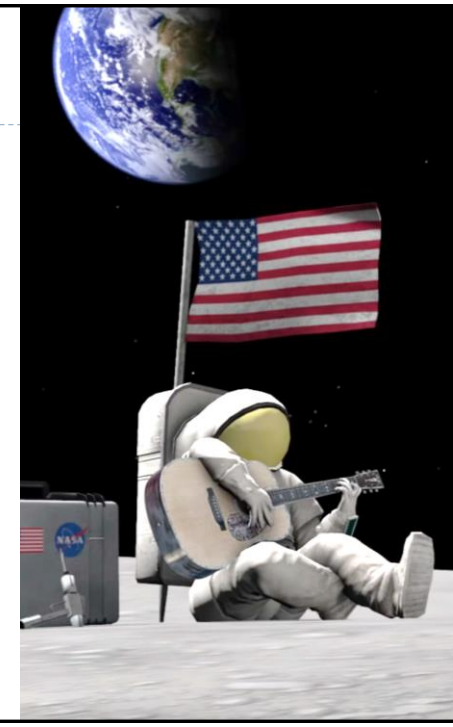
- Today:
  - 1. Exceptions must be dynamically allocated.
  - 3. Dynamic allocation failures are reported using exceptions.
- **Q: How does this statement describe two independent issues?**
  - 1. (see prev) Exceptions shouldn't need to be dynamically allocated.
  - 3. (see next) Allocation failures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



40

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 always be reported as program-recoverable errors (exceptions or otherwise)…



41

3. Allocation failure: Let the allocator decide

- Cologne (2019-07) Library Evolution subgroup (LEWG) direction:
  - Add a noexcept-queryable **allocator property** for "reports vs. fails-fast" on allocation failure. (Unanimous)
  - Recommend **conditional noexcept** based on the relevant allocator) pervasively on standard library functions that could only throw bad_alloc. (Unanimous)
  - Fail-fast for default std::allocator and default operator new. **(19 11 4 1 0)**
    - **NB: Even though a breaking change.**
- Cologne (2019-07) Language Evolution subgroup (EWG) direction:
  - Fail-fast for default std::allocator and default operator new. **(0 17 6 10 5)**
    - **NB: Even though a breaking change.**
    - **Updated proposal: Let the programmer decide for global new.**

42

42

Taxonomy

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

43

43

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 ~90% of all exceptions.
    - The vast majority of the standard library would not throw.
    - (Recall: Language-independent. Also true of Java and C#.)
  - **Simplification:** Eliminate ~90% 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*.

44

44

Part 1: Exception Handling (EH)

- 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 a "recoverable error"?
  - Recoverable error != programming bug != abstract machine corruption
  - Exceptions/codes != pre/post contracts != stack and heap overflow
- Four coordinated proposals

  1. Enable zero-overhead exception handling

  2&3. Throw fewer exceptions (~90% of all exceptions should not be)

  4. Support explicit "try" for visible propagation

45

45

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 "try/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"; }
```

46

46

4. Simplifications

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

47

47

1+2+3+4. Simplifications

- "One more thing"… Sets the stage for a potential new world:
  - 2+3: Enables "~90% of all functions are noexcept."
  - 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.

48

48

Part 1: Exception Handling (EH)

- 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. auto propagtn
- Key definition: What is a "recoverable error"?
  - Recoverable error != programming bug != abstract machine corruption
  - Exceptions/codes != pre/post contracts != stack and heap overflow
- Four coordinated proposals
  1. Enable zero-overhead exception handling
  2&3. Throw fewer exceptions (~90% of all exceptions should not be)
  4. Support explicit "try" for visible propagation

SF	WF	N	WA	SA
22	20	5	1	0
22	17	4	2	1
14	13	0	3	4

SF	WF	N	WA	SA
15	16	6	6	3
(see earlier)				
5	5	6	15	16

49

49

(but basically the same)

50

Part 2: Run-Time Type Information (RTTI)

- Establishing the problem: Today's RTTI violates the zero-overhead principle
  - "I can't afford to enable RTTI" ⇒ paying for what you don't use
  - "I can't afford to use dynamic_cast" ⇒ can write it more efficiently by hand
- Two coordinated proposals (coming over the next year-ish)
  1. Adopt **static reflection** ⇒ can make typeid/type_info consteval
  2. Adopt ***down_cast*** ⇒ can ask for only the work you need

51

51

Code review…

```
void myfunc(base* a) {
   ...
   if (something) {
      auto concrete = static_cast<derived*>(a);
      use(concrete);
   }
   ...
}
```

- Q: What do you think of this code?

52

52

Pathology 102… … why so downcast?

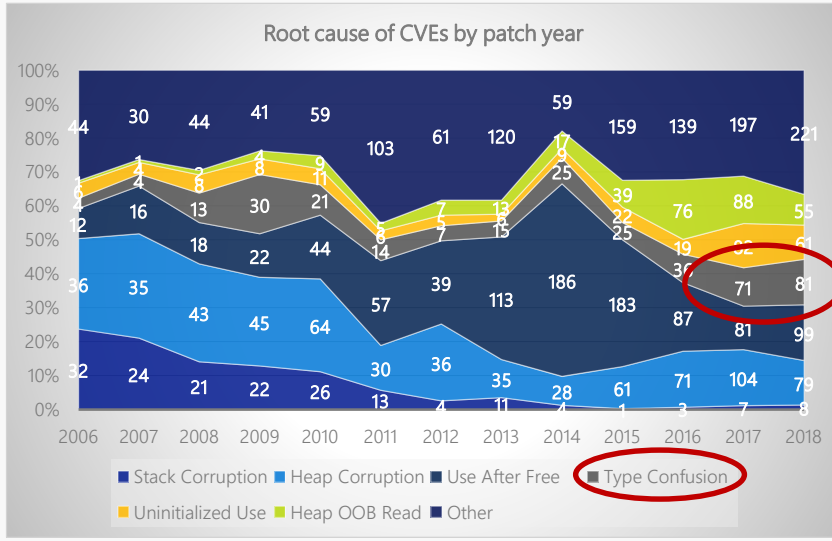
```
void myfunc(base* b) {
   ...
   if (something) {
      auto concrete = static_cast<derived*>(b);
      use(concrete);
   }
   ...
}
```

- Q: What do you think of this code?
  - **A: "Wait, that's an unchecked down-cast!"**

53

53

From Wednesday:



Joe Bialek (Microsoft)

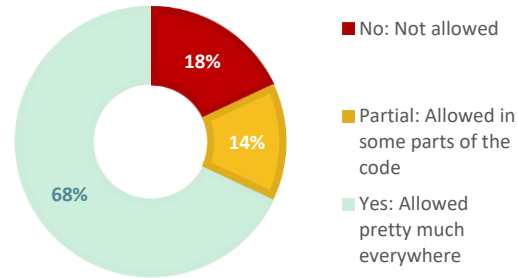
Shayne Hiet-Block (Microsoft)

54

isocpp.org 2019-04 survey

- **Many** "C++" projects ban RTTI in whole or in part.
  - One root cause of security vulnerabilities is "didn't use dynamic_cast" because:
    - "We can't, RTTI not enabled because it's too expensive."
    - "We can't, RTTI is enabled but *dynamic_cast* is too expensive."
    - **"And we tested so we know this downcast is safe…"**

Q13: [Is RTTI] allowed in your current project? (N=2,058)



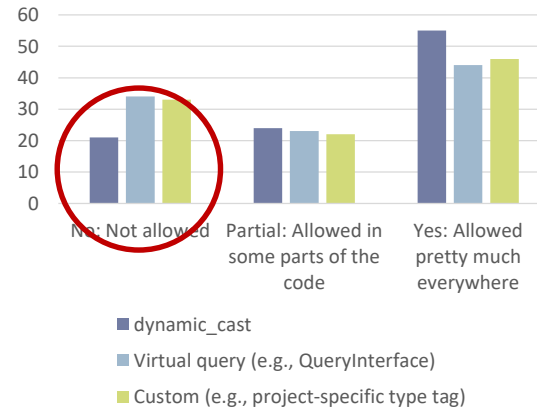
55

55

Fragmentation: isocpp.org 2019-04 survey

- **All nonstandard methods** are "allowed everywhere" only a little less than *dynamic_cast*.
- **Every method is banned** outright in >20% of projects.
  - A measure of fragmentation into dialects.
  - **NB: Worse than EH's 10%!**

Q12: What run-time casting methods are allowed on your current project? (N=2.083)



56

56

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

- Violates C++'s zero-overhead principle in two ways.
  1. "I can't afford to **enable** RTTI."
     - **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 **call dynamic_cast**."
     - **dynamic_cast incurs not-statically-boundable space and time overhead.**
     - **dynamic_cast is usually less efficient than a custom type-tag solution.**
     - Zero overhead principle, part 2: "When you do use it you can't reasonably write it better by hand" including by using alternatives.

57

57

Part 2: Run-Time Type Information (RTTI)

- Establishing the problem: Today's RTTI violates the zero-overhead principle
  - "I can't afford to enable RTTI" ⇒ paying for what you don't use
  - "I can't afford to use dynamic_cast" ⇒ can write it more efficiently by hand
- Two coordinated proposals (coming over the next year-ish)
  1. Adopt **static reflection** ⇒ can make typeid/type_info consteval
  2. Adopt ***down_cast*** ⇒ can ask for only the work you need

58

58

Type information: It's great! When you need that…

- **Prefer static** (know exactly what this program asks to use) to dynamic (gotta store it in case someone might ask).
  - Type information is great! But don't give it to me unless I ask for it.
- Observation 1: **Static reflection** makes run-time *type_info* redundant.
- Observation 2: Anything that's static, **we can easily make dynamic**.
  - Just store it somewhere.
  - Only pay for what you store ⇒ **zero-overhead**.
- **Proposal: Adopt static reflection + consteval programming, make *typeid*/*type_info* consteval.**

Andrew Sutton (Lock3)

59

59

dynamic_cast: It's great! When you need that…

- Observation 1: **Don't ask for more work than you need.**
  - *dynamic_cast* can do a lot, including cross-casts and virtual inheritance.
  - Example: *std::equal_range* vs. *std::lower_bound*.
- Observation 2: A frequent use case is **down-cast** without virtual inheritance.
  - Conjecture: A *down_cast* that doesn't compile unless the cast is downward and not across a virtual inheritance link can be much more efficient.
    - Why? Because we're asking for less work.
    - We get to stop being the demanding boss.
  - You only pay for what you use ⇒ **zero-overhead**.

60

60

Clang CFI and Windows CastGuard


Peter Collingbourne (Google)


Jim Radigan (Microsoft)


Kostya Serebryany (Google)


Joe Bialek (Microsoft)

- CastGuard injects **checks for *static_cast* down-casts**.
  - Based on Clang Control Flow Integrity (CFI).
  - No change to ABI incl. vtables. Doesn't use RTTI.
  - **Check is a simple range check.** No tree walks. Vtables are arranged in memory so "IS-A related type" is a SUB.
  - Limitation: Intra-DLL, doesn't check cross-DLL casts.
- Preliminary: Low impact on binary size and run time.
  - Current worst case: One Windows DLL has +1.5% binary size…
    - … **vs. unchecked static_cast** (not dynamic_cast, which is banned).
- **Proposal: Provide *down_cast* to ask for only what you need.**
  - To compile, must be base-to-derived and not cross a virtual inheritance link.
  - **Constant-time** within a DLL.
  - **Zero-overhead:** Don't pay if you don't use it + when you use it you can't do better by hand.

61

61

15km (~50,000') comparison: **instructions executed**

dynamic_cast, simple SI

```
71472 c744242000000000 mov     dword ptr [rsp+20h], 0 ss:0000001f`baeff890=567bf330
bd60e 1bc0        sbb     eax, eax
bd14d 741f        je      test3!FindSITargetTypeInstance+0x52 (bd16e) [br=0]
bd68f 4c8bc8      mov     r9, rax
bd692 4885c0      test    rax, rax
bd695 7510        jne     test3!__RTDynamicCast+0xf7 (bd6a7)   <-- take this jump

bd6a7 4139590c    cmp     dword ptr [r9+0Ch], ebx
bd6ab 7c13        jl      test3!__RTDynamicCast+0x110 (bd6c0) [br=1]   <-- take this jump

bd6c0 49634108    movsxd  rax, dword ptr [r9+8] ds:c9bd0=00000000
bd6c4 4803c3      add     rax, rbx
bd6c7 4903c6      add     rax, r14
bd6ca e90dffffff  jmp     test3!__RTDynamicCast+0x2c (bd5dc)          <-- jump

bd5dc 4c8d5c2450  lea     r11, [rsp+50h]
bd5e1 498b5b38    mov     rbx, qword ptr [r11+38h]
bd5e5 498b7340    mov     rsi, qword ptr [r11+40h]
bd5e9 498be3      mov     rsp, r11
bd5ec 415f        pop     r15
bd5ee 415e        pop     r14
bd5f0 415d        pop     r13
bd5f2 415c        pop     r12
bd5f4 5f          pop     rdi
bd5f5 c3          ret
```

CastGuard, simple SI

```
Start of CastGuard check
; rcx == The right-hand side object pointer.

; First do the nullptr check. This could be optimized away but is not today.
; N.B. If the static_cast has to adjust the pointer base, this nullptr check
; already exists.
4885c9          test    rcx, rcx
7416            je      codegentest!DoCast+0x26

; Next load the RHS vftable and the comparison vftable.
488b11          mov     rdx, qword ptr [rcx]
4c8d05ce8f0500 lea     r8, [codegentest!MyChild1::`vftable']

; Now do the range check. Jump to the AppCompat check if the range check fails.
492bc0          sub     rdx, r8
4883f820        cmp     rdx, 20h
7715            ja      codegentest!DoCast+0x3b       ; Jump to app-compat check
```

62

62

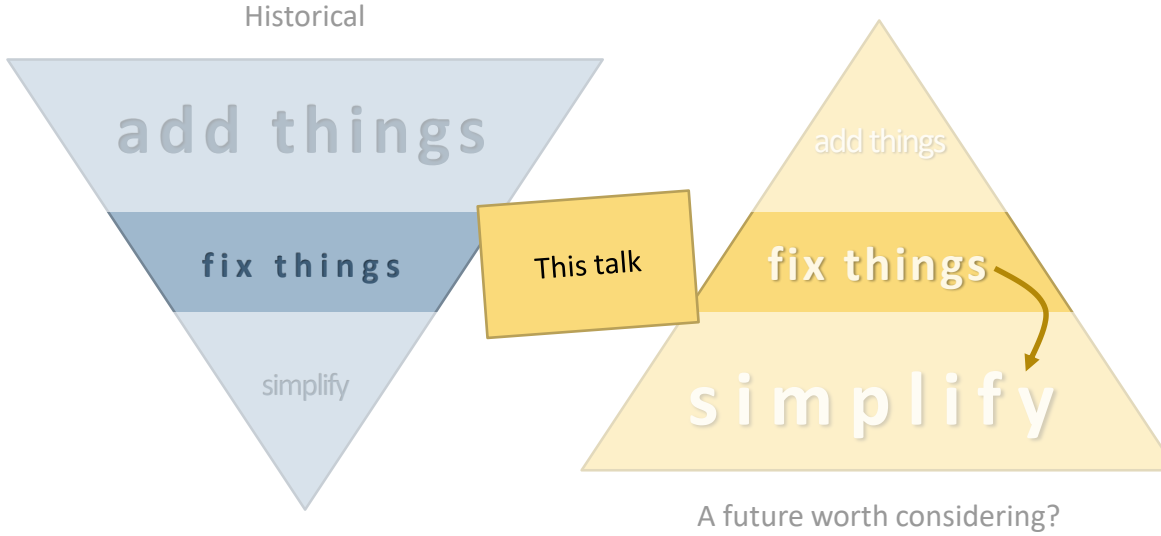
CastGuard, simple SI

```
Start of CastGuard check
; rcx == The right-hand side object pointer.

; First do the nullptr check. This could be optimized away but is not today.
; N.B. If the static_cast has to adjust the pointer base, this nullptr check
; already exists.
4885c9          test    rcx, rcx
7416            je      codegentest!DoCast+0x26

; Next load the RHS vftable and the comparison vftable.
488b11          mov     rdx, qword ptr [rcx]
4c8d05ce8f0500 lea      r8, [codegentest!MyChild1::`vftable']

; Now do the range check. Jump to the AppCompat check if the range check fails.
492bc0          sub     rdx, r8
4883f820        cmp     rdx, 20h
7715            ja      codegentest!DoCast+0x3b       ; Jump to app-compat check
```

63

63

C++'s evolution priorities

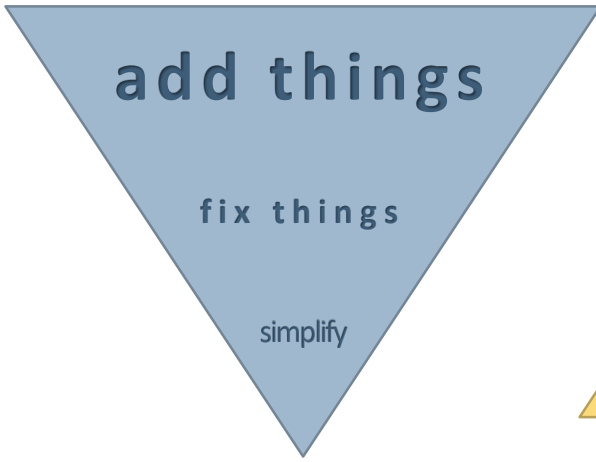


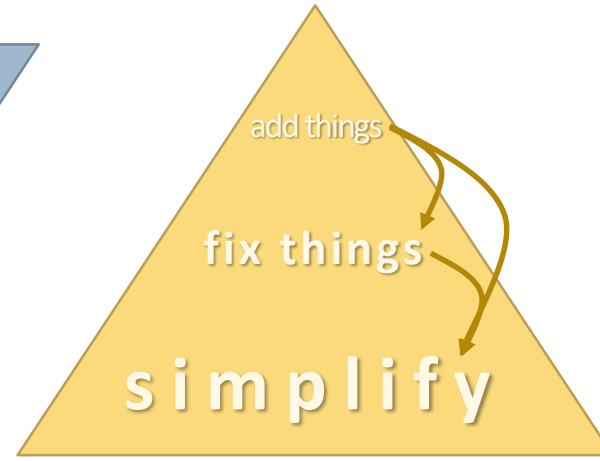
64

64

C++'s evolution priorities

Historical





A future worth considering?

65

65

A "pay only for what you use" principle

static by default

dynamic by opt-in

66

66

So what *is* C++ / "C++ic"?

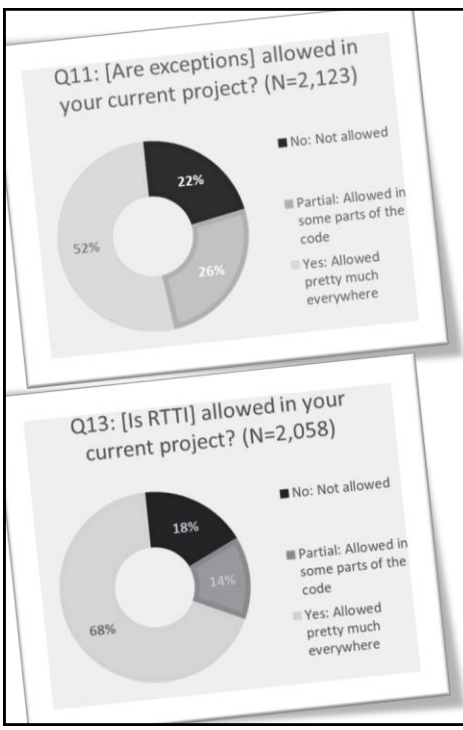
Historical strength: **static**
typing, compilation, linking

Major current trend (IME): ↓dynamic, ↑**static**
concepts, contracts, constexpr, static error types (fs_error), virtual → templates, RTTI's typeid → reflection, …

- Core principles:
  - **Zero-overhead abstraction** — don't pay for what you don't use, what you use is as efficient as you can reasonably write by hand
  - **Determinism & control** — over time & space, close to hardware, leave no room for a lower language, trust the programmer
  - **Link compatibility** — with C95 and C++prev
- Useful pragmatics for adoption and library consumption:
  - **Backward source compat with C** — mostly C95, read-only, consume headers & seamlessly call
  - **Backward source compat with C++prev** — C++98 and later, read-mostly, to use & to maintain
- What is *not* core C++:
  - **Specific syntax** — we've been adding & recommending C-incompatible syntax since 1979
  - **Tedium** — most modern abstractions (e.g., range-*for*, *override*) compatible with zero-overhead
  - **Lack of good defaults** — good defaults are fine, as long as the programmer can override them
  - **Sharp edges** — e.g., brittle dangling pointers are not necessary for efficiency and control

67

67



De-fragmenting C++

Making **exceptions** and **RTTI** more affordable and usable

Herb Sutter

68