P3745R0
Slides for EWG presentation of P3100R2 Implicit Contract Assertions

Timur Doumler
Joshua Berne

ISO C++ Meeting, Sofia, 16-21 June 2025

History

- Idea first published in P1995R0 in March 2020
- P3100R0 published in May 2024
- P3100R1 reviewed by SG21 in Wrocław (November 2024)
  Poll: We support the direction of P3100R1 and encourage the authors to come back with a fully specified proposal.
  19 / 6 / 0 / 0 / 0 (Consensus)
- P3100R2 reviewed by SG23 in Sofia (June 2025)
  Poll: We should promise more committee time to pursuing P3100R2.
  18 / 3 / 1 / 0 / 2 (Consensus)

Goal

A standardised framework for runtime detection and mitigation of undefined behaviour across the entire C++ language.

Goal

A standardised framework for runtime detection and mitigation
of undefined behaviour across the entire C++ language.

Target ship vehicle

The "core language UB" white paper
that EWG agreed in Hagenberg to pursue,
and C++29

Initial draft proposal for core language UB white paper: Process and major work items

Doc #	P3656 R1
Authors	Herb Sutter, Gašper Ažman
Date	2025-03-23
Audience	EWG

Abstract: Background and scope

At Hagenberg 2025-02, EWG encouraged the following work:

> Poll: Pursue a language safety white paper in the C++26 timeframe containing systematic treatment of core language Undefined Behavior in C++, covering Erroneous Behavior, Profiles, and Contracts. Appoint Herb and Gašper as editors.

SF	F	N	A	SA
32	31	6	4	4

Initial draft proposal for core language UB white paper: Process and major work items

Doc #	P3656 R1
Authors	Herb Sutter, Gašper Ažman
Date	2025-03-23
Audience	EWG

Abstract: Background and scope

At Hagenberg 2025-02, EWG encouraged the following work:

> Poll: Pursue a language safety white paper in the C++26 timeframe containing systematic treatment of core language Undefined Behavior in C++, covering Erroneous Behavior, Profiles, and Contracts. Appoint Herb and Gašper as editors.

SF	F	N	A	SA
32	31	6	4	4

"Major work items" proposed in P3656R1

- Enumerate cases of language UB
- Perform basic categorisation:
  - Which ones are security-related?
  - Which are efficiently locally diagnosable?
- List possible tools for handling these UB cases
- Take a first pass at penciling in which tool to use for each UB case
- Group cases (profile names / contract labels)
- Suggested guidance for tags & descriptions in the Standard document

What P3100R2 does

- Enumerate all cases of explicit language UB in C++
- Group them into 12 categories
- Classify them according to relevant criteria:
  - (Relevance for security)
  - Local checkability
  - Cost of diagnosis
  - (Non-)existence of well-defined fallback behaviour
  - Discussion of mitigation strategies
- Proposal (with wording) for how to specify, in the C++ Standard, optional runtime checks **and** fallback behaviour wherever possible

What P3100R2 does

- Enumerate all cases of explicit language UB in C++
- Group them into 12 categories
- Classify them according to relevant criteria:
  - (Relevance for security)
  - Local checkability
  - Cost of diagnosis
  - (Non-)existence of well-defined fallback behaviour
  - Discussion of mitigation strategies
- Proposal (with wording) for how to specify, in the C++ Standard, optional runtime checks **and** fallback behaviour wherever possible

Enumerate all UB

- List of UB cases created independently from scratch
  (*not* based on Shafik Yaghmour's work – P1705R1, P3075R0)
- We only consider UB specified explicitly, not implicit UB
  (wording must contain the word "undefined")
- We only consider language UB, not library UB
- We do not consider IFNDR
- Stable identifiers for each specified case of UB
  e.g. {lifetime.outside.pointer.static.cast}
- PR open against ub-ifndr branch on https://github.com/cplusplus/draft
  (to merge our list with the one based on Shafik Yaghmour's)

90 cases of explicit language UB

12 categories

I. Initialisation
II. Bounds
III. Type and Lifetime
IV. Arithmetic
V. Threading
VI. Sequencing
VII. Assumptions
VIII. Control Flow
IX. Replacement Functions
X. Coroutines
XI. Templates
XII. Preprocessor

12 categories

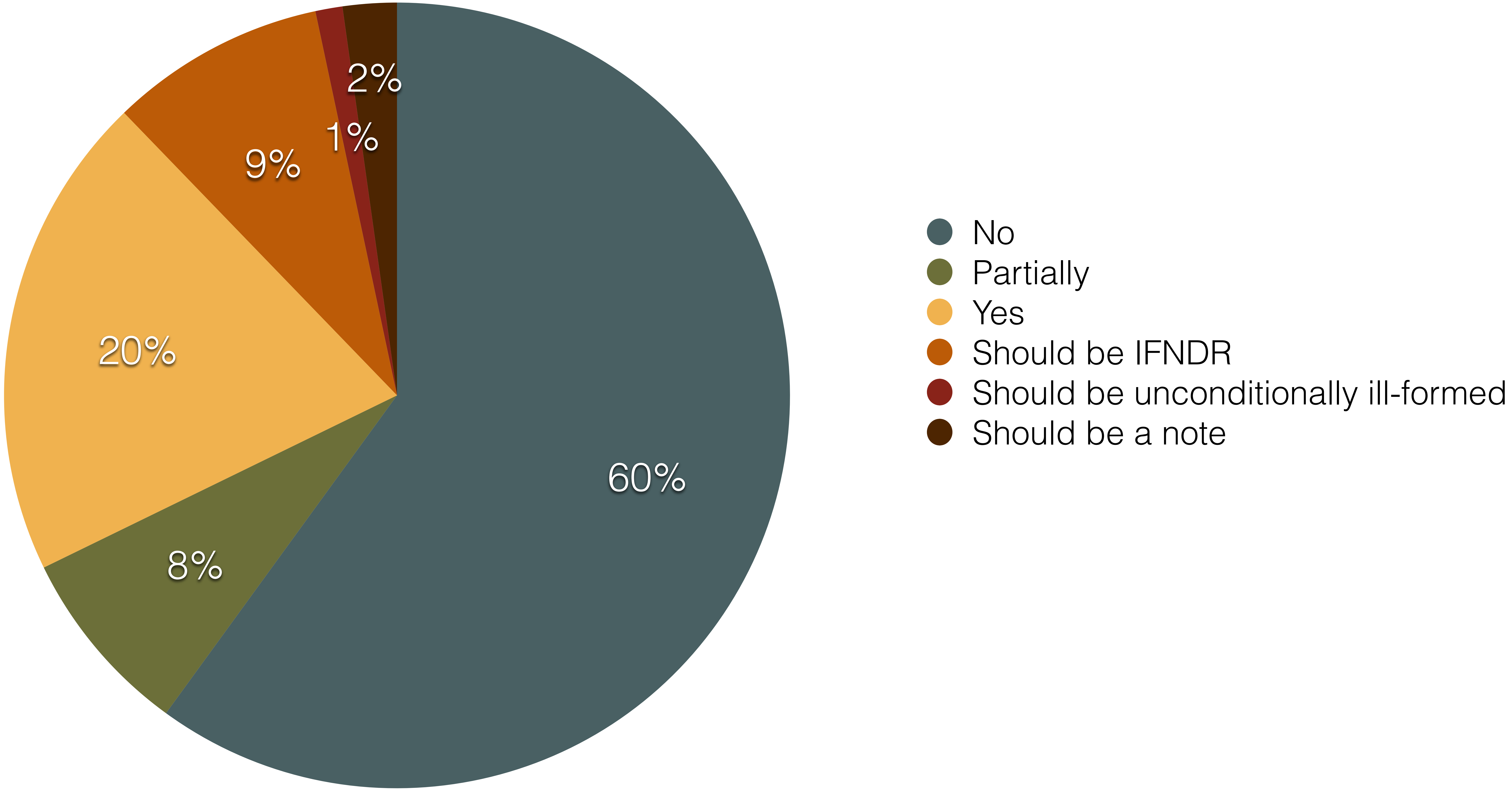
I. Initialisation – 1 case
II. Bounds – 5 cases
III. Type and Lifetime – 52 cases
IV. Arithmetic – 9 cases
V. Threading – 1 case
VI. Sequencing – 1 case
VII. Assumptions – 1 case
VIII. Control Flow – 6 cases
IX. Replacement Functions – 3 cases
X. Coroutines – 2 cases
XI. Templates – 1 case
XII. Preprocessor – 8 cases

Security-relevant?

Commonly associated with security vulnerabilities

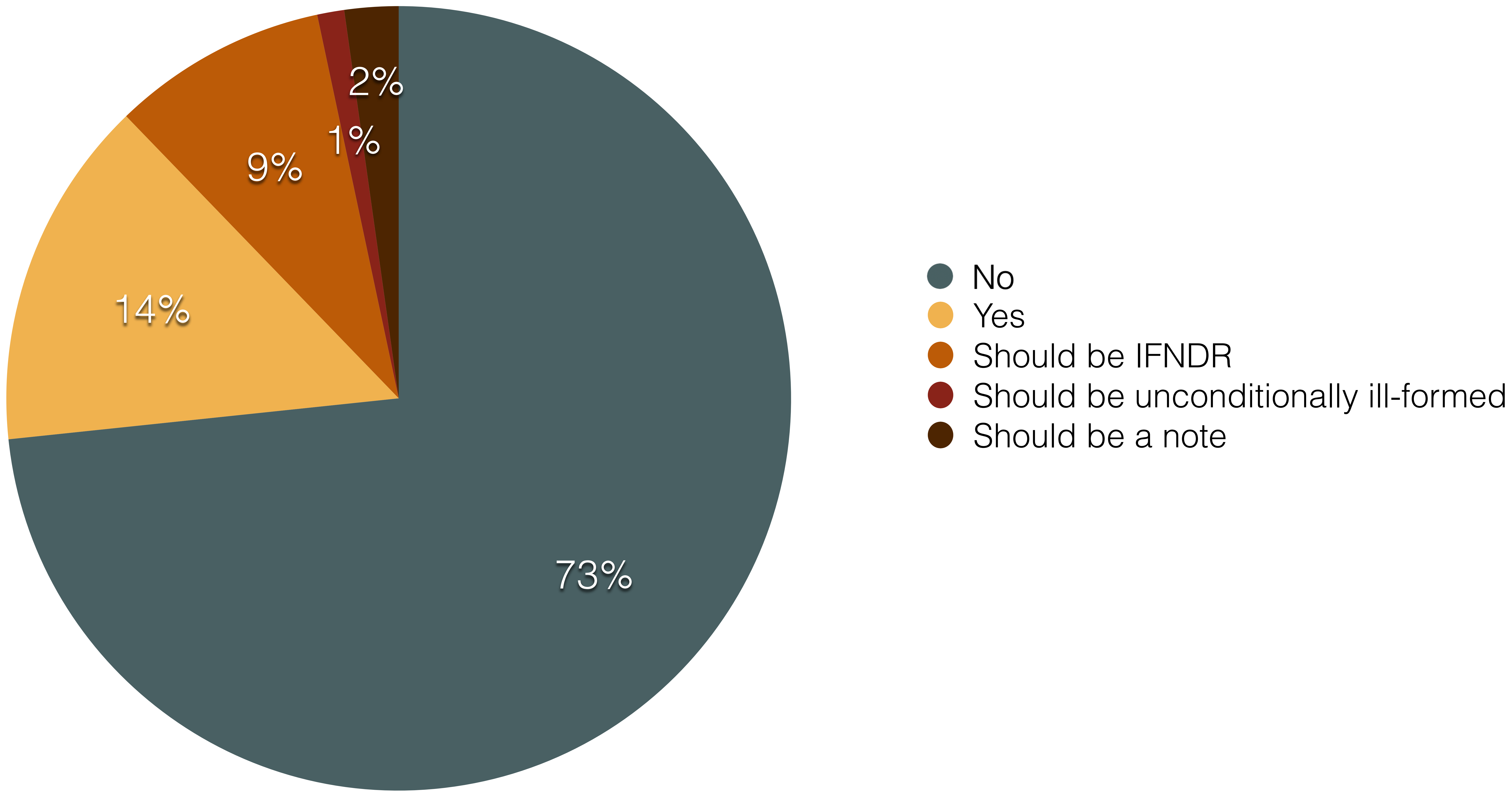
I. Initialisation – 1 case
II. Bounds – 5 cases
III. Type and Lifetime – 52 cases
IV. Arithmetic – 9 cases
V. Threading – 1 case
VI. Sequencing – 1 case
VII. Assumptions – 1 case
VIII. Control Flow – 6 cases
IX. Replacement Functions – 3 cases
X. Coroutines – 2 cases
XI. Templates – 1 case
XII. Preprocessor – 8 cases

Locally diagnosable at runtime?



 | @timur_audio | https://timur.audio

Well-defined fallback behaviour?



Proposed design

Part 1:
systematically introduce runtime checks

Part 2:
systematically replace UB by well-defined fallback behaviour

Part 3:
Provide opt-out

Part 1:
systematically introduce runtime checks

Part 2:
systematically replace UB by well-defined fallback behaviour

Part 3:
Provide opt-out

Refresher: C++26 Contracts

```
T& operator[] (size_t index)
  pre (index < size());   // contract assertion
```

Refresher: C++26 Contracts

```
T& operator[] (size_t index)
  pre (index < size());   // contract assertion
```

- Evaluated with one of four possible evaluation semantics:
  ignore, *observe*, *enforce*, *quick-enforce*
- If predicate is checked (*observe*, *enforce*, *quick-enforce*) and check fails:
  - contract violation handler is called (*observe*, *enforce*)
  - program is terminated (*enforce*, *quick-enforce*)
- default contract violation handler can be replaced at link time

Refresher: C++26 Contracts

```
T& operator[] (size_t index)
  pre (index < size());   // explicit contract assertion
```

- Evaluated with one of four possible evaluation semantics: *ignore*, *observe*, *enforce*, *quick-enforce*
- If predicate is checked (*observe*, *enforce*, *quick-enforce*) and check fails:
  - contract violation handler is called (*observe*, *enforce*)
  - program is terminated (*enforce*, *quick-enforce*)
- default contract violation handler can be replaced at link time

Strategy

- If it is in principle possible to insert a runtime check for a case of UB (even if it's expensive and/or requires global instrumentation), we specify that check as an **implicit contract assertion**.

Proposed wording transformation

"If *X* is not true, the behaviour of operation *A* is undefined"

→

"Operation *A* has an implicit precondition assertion that *X* is true"

Strategy

- If it is in principle possible to insert a runtime check for a case of UB (even if it's expensive and/or requires global instrumentation), we specify that check as an **implicit contract assertion**.
  - behaves the same as an explicit contract assertion
  - except that it is inserted by the implementation

```
enum class assertion_kind : unspecified {
  pre = 1,
  post = 2,
  assert = 3,
  implicit = 4
}
```

Strategy

- If it is in principle possible to insert a runtime check for a case of UB (even if it's expensive and/or requires global instrumentation), we specify that check as an **implicit contract assertion**.
  - behaves the same as an explicit contract assertion
  - except that it is inserted by the implementation

Strategy

- If it is in principle possible to insert a runtime check for a case of UB (even if it's expensive and/or requires global instrumentation), we specify that check as an **implicit contract assertion**.
  - behaves the same as an explicit contract assertion
  - except that it is inserted by the implementation
  - we do not require an implementation to provide all possible checks (*ignore* is always a valid choice)

Strategy

- If it is in principle possible to insert a runtime check for a case of UB (even if it's expensive and/or requires global instrumentation), we specify that check as an **implicit contract assertion**.
  - behaves the same as an explicit contract assertion
  - except that it is inserted by the implementation
  - we do not require an implementation to provide all possible checks (*ignore* is always a valid choice)
  - but we require an implementation to document the selection mechanism (which semantic is chosen *is implementation-defined*)

Benefits

- Bringing compiler flags, sanitisers, etc. that implement these checks already today into scope of the C++ Standard
- Enabling things like a Standard callback API for diagnosed runtime UB
- Codifying standard names and categories for UB in the Standard
- Enables seamless integration with Contract labels, Profiles, etc.

Part 1:
systematically introduce
runtime checks

**Part 2:
systematically replace UB by
well-defined fallback behaviour**

Part 3:
Provide opt-out

Introduce well-defined fallback behaviour

"If *X* is not true, the behaviour of operation *A* is undefined"

Does fallback behaviour exist?

"Operation *A* has an implicit precondition assertion that *X* is true; continuing execution past a violation of this precondition is undefined behaviour."

"Operation *A* has an implicit precondition assertion that *X* is true; after a violation of this precondition, <*fallback behaviour*> happens".

Part 1:
systematically introduce runtime checks

Part 2:
systematically replace UB by well-defined fallback behaviour

Part 3:
Provide opt-out

Four evaluation semantics

- ***ignore***: do not check predicate
- ***observe***: check predicate; if `false`, call contract-violation handler; when handler returns, continue
- ***enforce*** check predicate; if `false`, call contract-violation handler; when handler returns, terminate
- ***quick-enforce***: check predicate; if `false`, terminate immediately

Five evaluation semantics

- ***assume***: do not check predicate but assume it is **`true`**; if predicate is *not* **`true`**, the behaviour is undefined
- ***ignore***: do not check predicate
- ***observe***: check predicate; if **`false`**, call contract-violation handler; when handler returns, continue
- ***enforce*** check predicate; if **`false`**, call contract-violation handler; when handler returns, terminate
- ***quick-enforce***: check predicate; if **`false`**, terminate immediately

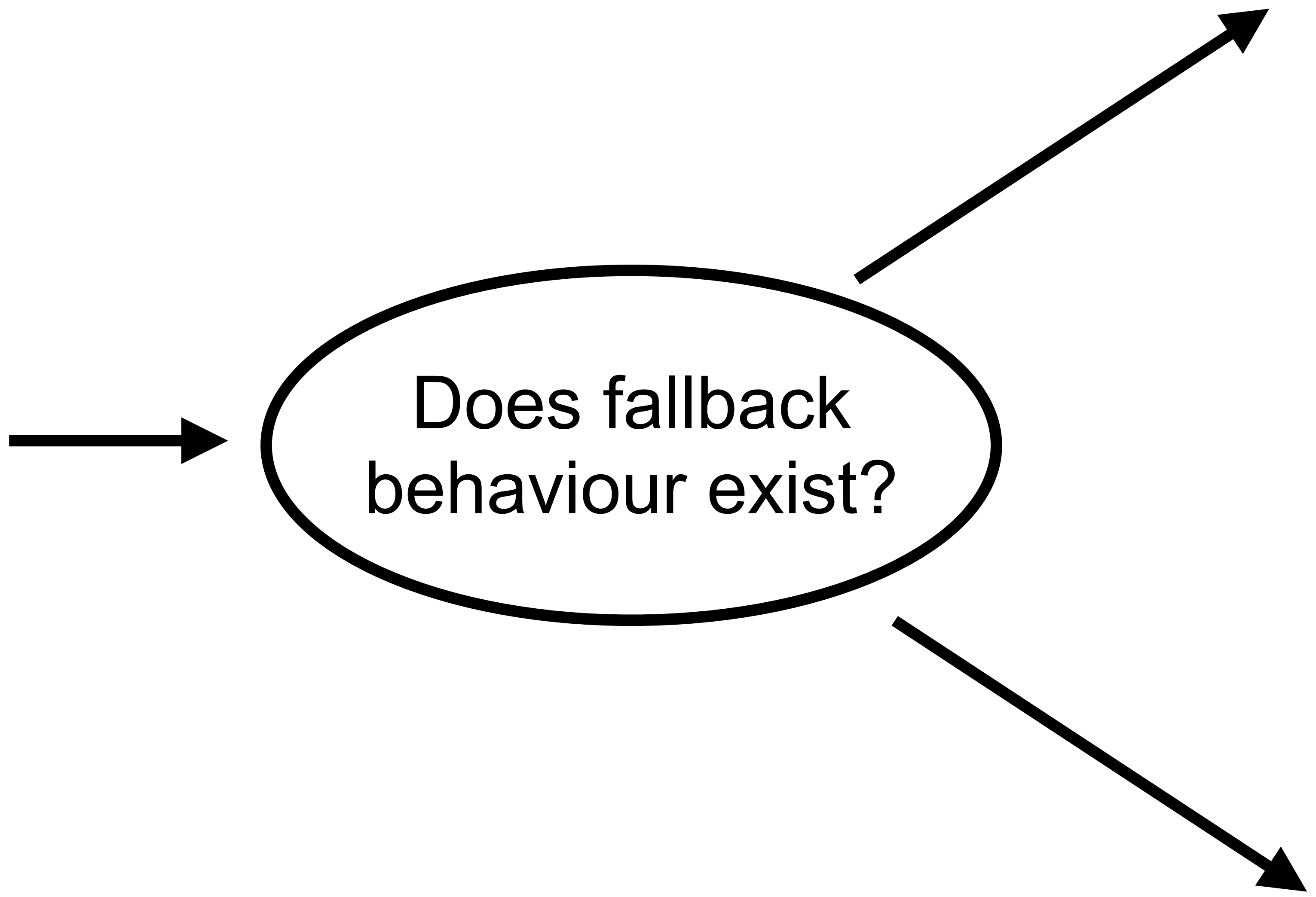
Introducing *assume*

- We do not propose to allow assume for **explicit** contract assertions
- Only for **implicit** contract assertions
- Every implementation is already conforming with this proposal today
- Because *assume* is already the default for these assertions today!

Examples

Introduce well-defined fallback behaviour

"If *X* is not true, the behaviour of operation *A* is undefined"

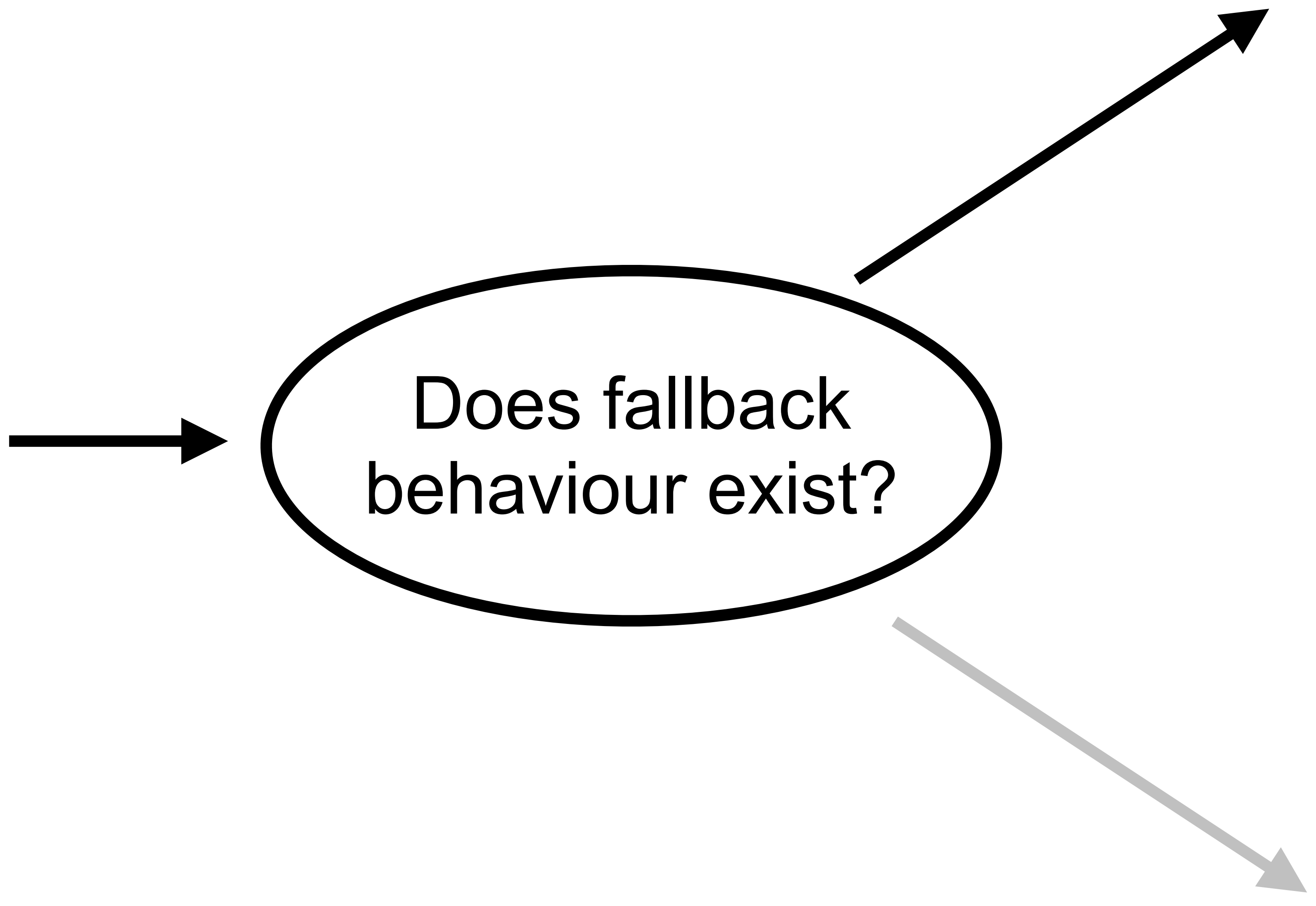


"Operation *A* has an implicit precondition assertion that *X* is true; continuing execution past a violation of this precondition is undefined behaviour."

"Operation *A* has an implicit precondition assertion that *X* is true; after a violation of this precondition, <*fallback behaviour*> happens".

Introduce well-defined fallback behaviour

"If *X* is not true, the behaviour of operation *A* is undefined"



"Operation *A* has an implicit precondition assertion that *X* is true; continuing execution past a violation of this precondition is undefined behaviour."

"Operation *A* has an implicit precondition assertion that *X* is true; after a violation of this precondition, <*fallback behaviour*> happens".

Example 1: Out of bounds array access

```
int main() {
  int a[10] = { 1, 1, 2, 3, 5 };  // array of known bounds
  std::size_t i;
  std::cin >> i;
  return a[i];  // potential UB here
}
```

Example 1: Out of bounds array access

```
int main() {
  int a[10] = { 1, 1, 2, 3, 5 };  // array of known bounds
  std::size_t i;
  std::cin >> i;
  return a[i];  // potential UB here
}

template <typename T, std::size_t N>
T& __index_into_array(T (&a)[N], std::size_t i) {
  return *(&a + i);
}
```

Example 1: Out of bounds array access

```
int main() {
  int a[10] = { 1, 1, 2, 3, 5 };  // array of known bounds
  std::size_t i;
  std::cin >> i;
  return a[i];  // potential UB here
}

template <typename T, std::size_t N>
T& __index_into_array(T (&a)[N], std::size_t i)
pre (i < N) {   // implicit contract assertion
  return *(&a + i);
}
```

Example 1: Out of bounds array access

```
int main() {
  int a[10] = { 1, 1, 2
  std::size_t i;
  std::cin >> i;
  return a[i];  // pote
}

template <typename T, std::size_t N>
T& __index_into_array(T (&a)[N], std::size_t i)
pre (i < N) {   // implicit contract assertion
  return *(&a + i);
}
```

ignore == status quo

enforce for all arrays == AddressSanitizer

quick-enforce for arrays of known bound == Clang's -fbounds-safety

Introduce well-defined fallback behaviour

"If *X* is not true, the behaviour of operation *A* is undefined"

Does fallback behaviour exist?

"Operation *A* has an implicit precondition assertion that *X* is true; continuing execution past a violation of this precondition is undefined behaviour."

"Operation *A* has an implicit precondition assertion that *X* is true; after a violation of this precondition, <*fallback behaviour*> happens".

Example 2: Signed integer overflow

```
int g(int i, int j) {
  return i + j;
}

// We pretend that built-in integer addition was performed as-if by:
int operator+(int a, int b)
pre ((b >= 0 && a <= INT_MAX - b) || (b < 0 && a >= INT_MIN - b));
```

Example 2: Signed integer overflow

assume == status quo

ignore == GCC **`-fwrapv`** (one possibility)

quick-enforce == GCC **`-ftrapv`**

enforce == UBSan

```
int g(int i, int j) {
  return i + j;
}

// We pretend that built-in integer addition was performed as-if by:
int operator+(int a, int b)
pre ((b >= 0 && a <= INT_MAX - b) || (b < 0 && a >= INT_MIN - b)) {
  // well-defined behaviour
}
```

Introduce well-defined fallback behaviour

"If *X* is not true, the behaviour of operation *A* is undefined" → **Does fallback behaviour exist?**

→ "Operation *A* has an implicit precondition assertion that *X* is true; continuing execution past a violation of this precondition is undefined behaviour."

→ "Operation *A* has an implicit precondition assertion that *X* is true; after a violation of this precondition, <*fallback behaviour*> happens".

Additional features with Labels

(P3400, <u>not</u> this paper)

How labels extend P3100: Categories

- Implicit contract assertions have implicit labels
- These labels have standard names
- These labels are organised in standard categories (i.e., "bounds")
- You can add your own contract assertions to those categories:

```
MyVector::operator[] (size_t i)
  pre <category::bounds> (i < size());
```

- You can name the cases / categories of UB in the contract-violation handler, branch on them, etc.

How labels extend P3100: In-source semantic control

- Directive that adds labels to specified implicit contract assertions in a scope (file, class, function, block ....)
- These labels can control the semantic of these assertions (*quick-enforce* all lifetime assertions, *observe* all arithmetic assertions...)

How labels extend P3100: In-source semantic control

- Directive that adds labels to specified implicit contract assertions in a scope (file, class, function, block ....)
- These labels can control the semantic of these assertions (*quick-enforce* all lifetime assertions, *observe* all arithmetic assertions...)

```
int f(int a, int b) {
  contract_assert implicit arithmetic |= always_enforce;
  return a + b;
}
```

How does all this fit into the bigger picture?

Towards Safe C++

Configurable

Profiles
Named configuration presets for the features below

P3100

- Redefined behaviour (e.g., P2644R1 range-based for loop fix)
- Erroneous behaviour (e.g., return an erroneous value)
- Runtime checks (e.g., -ftrapw, sanitisers)
- Language subsetting (e.g., disallow C-style casts)
- Annotations (e.g., lifetime annotations, Clang's __counted_by)
- New features (e.g., borrow checking, std:: saturate_cast)

Legacy code

New code /
code you can change