Practical Tips for Safer C++

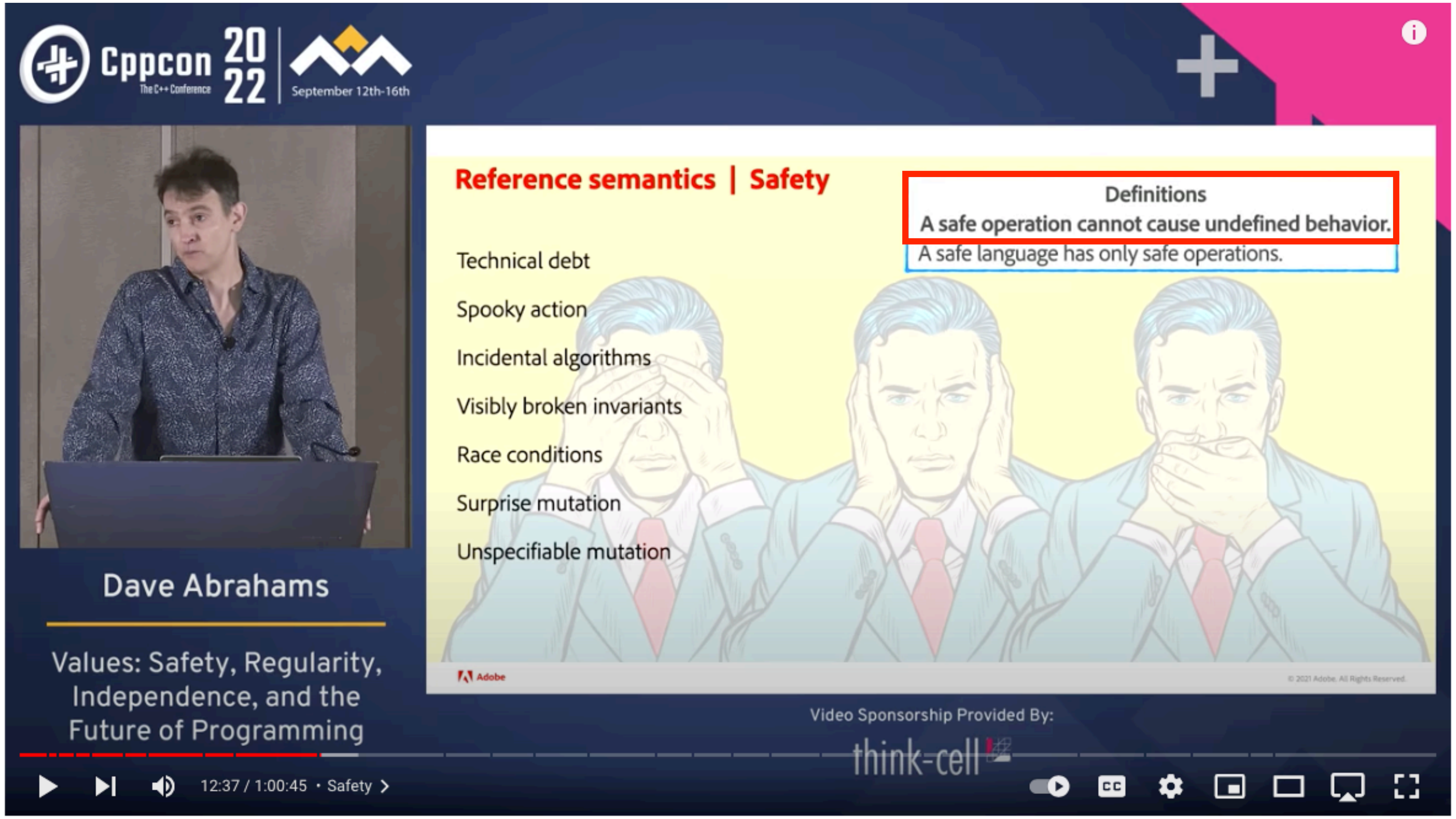
CppNorth 🇨🇦 2024

Tristan Brindle

Agenda

- What is safety?
- Numeric safety
- Bounds safety
- Memory safety
- Testing
- Questions

What is Safety?



Value Semantics: Safety, Independence, Projection, & Future of Programming - Dave Abrahams CppCon 22

CppCon
129K subscribers
Subscribed
318
Share
Clip
Save

A safe operation cannot cause undefined behaviour.

Dave Abrahams, CppCon 2022

What is Safety?

- We cannot reason about our programs in the presence of undefined behaviour
- It gets worse: optimisers take advantage of UB, meaning that we cannot reason about a program that has UB *anywhere*

Undefined Behavior in C++: What Every Programmer Should Know and Fear - Fedor Pikus - CppCon 2023

What is Safety?

- The C++ language and standard library make it easy to accidentally stumble into UB
- We cannot remove undefined behaviour from C++ without drastic language and library changes
- ...but we can take steps to make things safe**r**

Disclaimer (1)

- I’m not a safety or security expert
- This talk is based on personal experience
- It’s far from comprehensive…
- …but hopefully interesting :)

Disclaimer (2)

- I’m the author of Flux, a library which (among other things) aims to make coding in C++ less prone to accidental UB
- This wasn’t intended to be a Flux talk…
- …but I’m going to end up mentioning it quite a bit
- Find it at https://github.com/tcbrindle/flux

Numeric Safety

Numeric Safety

- Consider the following function:

```cpp
int add(int a, int b)
{
    return a + b;
}
```

- Is this a safe function in standard C++?

- No

- For example, `add(INT_MAX, 1)` causes **signed integer overflow** — undefined behaviour!

Numeric Safety

- Leaving signed overflow undefined allows the compiler to assume that it will “never happen”
- “Obvious” optimisations can be applied
- For example:
    - `3 + x - 3` can be optimised to `x`
    - `x < x + 10` can be optimised to `true`
    - `(x + y) + z` is the same as `x + (y + z)`

Numeric Safety

- Is this a safe function?

```
unsigned int add(unsigned int a, unsigned int b)
{
    return a + b;
}
```

- Unlike with signed integers, *unsigned* overflow is well-defined in C++
- This means that most operations on unsigned integers are not UB...
- ...but that doesn't mean they're not problematic
- Who hasn't accidentally tried to allocate an array of 18446744073709551615 elements?

Numeric Safety

- In Rust:
  - Integer overflow (signed and unsigned) panics in debug mode
  - Wraps in release mode
  - Library functions for explicit wrapped, checked and unchecked operations
- In Swift:
  - Integer overflow causes a runtime error unless compiled with -Ounchecked
  - Built-in operators (`&+` etc) for explicit wrapping operations
  - `addWithOverflow()` etc for explicit overflow checks
  - No (?) explicitly unchecked operations

Numeric Safety

- In GCC and Clang, `-fwrapv` gives signed integers *wrapping* semantics, as with unsigned
- Like release mode in Rust
- No unsafe behaviour on overflow…
- …but a few optimisations can no longer be applied
  - e.g. `x < x + 10` is not always `true`

Numeric Safety

- In Clang, compiling with `-ftrapv` makes signed overflow *trap* (i.e. crash your program)
- `-ftrapv` is also available in GCC, but doesn't work as reliably
- With both compilers, UB Sanitizer is a more comprehensive alternative:
  - `g++ my_file.cpp -fsanitize=signed-integer-overflow -fsanitize-trap`
- Can also use `-fsanitize=unsigned-integer-overflow` to get equivalent checks for unsigned integers
- Equivalent to debug mode in Rust, or the default mode in Swift
- Unlike most sanitizer options, these seem to be fine to use in production builds

Numeric Safety

- Checked arithmetic inhibits certain optimisations
  - https://lemire.me/blog/2020/09/23/how-expensive-is-integer-overflow-trapping-in-c/
- In particular, checked arithmetic can prevent auto-vectorisation
- In tight inner loops, we might prefer wrapping or even unsafe semantics

Numeric Safety

- Various libraries are available which provide safe integer operations, e.g.
  - SafeInt
  - Boost.SafeNumerics
  - Flux
- Some libraries also provide replacement integer types which overload the usual arithmetic operators with safe versions
- Unlike compiler flags, these retain the ability to use unsafe operations when absolutely necessary

Numeric Safety

- At least floating point numbers are fine!
  - ...aren't they?
- Is this a safe function?

```cpp
void sort_floats(std::vector<float>& vec)
{
    std::sort(vec.begin(), vec.end());
}
```

Numeric Safety

- Problem: in IEEE 754, NaNs are unordered:
  - `a == b`, `a < b` and `b < a` are all false if either `a` or `b` is NaN
- This means that `operator<` on floats is not a *strict weak order*
  - => it is undefined behaviour to call `std::sort()` on an array containing NaNs using the default comparator
- This can be a problem in real world code
  - https://gcc.gnu.org/bugzilla/show_bug.cgi?id=41448

Numeric Safety

- Fortunately, IEEE 754 also defines a *total order* for all floats, including NaNs
- In C++20, `std::strong_order()` on floats will use the IEEE total order

```
void sort_floats(std::vector<float>& vec)
{
    std::sort(vec.begin(), vec.end(), [](float a, float b) {
        return std::is_lt(std::strong_order(a, b));
    });
}
```

Numeric Safety: Summary

- Signed overflow is UB — compilers can and do optimise around this
- Unsigned overflow is not UB, but usually wrong
- GCC/Clang can use compiler options to change the defaults. Trapping in debug mode, wrapping in release mode is probably reasonable for most use cases
- Alternatively, libraries are available which can perform safe integer operations
- Floating point data containing NaNs can cause UB with certain standard library algorithms
- Use a custom comparator wrapping `std::strong_order()` to avoid this

Bounds Safety

Bounds Safety

- In C++, array access is unchecked:

```
int main()
{
    int arr[] = {1, 2, 3};

    return arr[100'000]; // Oops
}
```

- This is generally also the case for `operator[]` on `std::array`, `std::vector`, `std::span` etc...

Bounds Safety

- Out of bounds reads and writes are among the most common software vulnerabilities:

2023 CWE Top 10 KEV Weaknesses

KEV Weaknesses Rank	CWE-ID	Weakness Name	Analysis Score	Number of Mappings in the KEV Dataset	Average CVSS
1	CWE-416	Use After Free	73.99	44	8.54
2	CWE-122	Heap-based Buffer Overflow	56.56	32	8.79
3	CWE-787	Out-of-bounds Write	51.96	34	8.19
4	CWE-20	Improper Input Validation	51.38	33	8.27
5	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	49.44	25	9.36
6	CWE-502	Deserialization of Untrusted Data	29.00	16	9.06
7	CWE-918	Server-Side Request Forgery (SSRF)	27.33	16	8.72
8	CWE-843	Access of Resource Using Incompatible Type ('Type Confusion')	26.24	16	8.61
9	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	19.90	14	8.09
10	CWE-306	Missing Authentication for Critical Function	12.98	8	8.86

Source: https://cwe.mitre.org/top25/archive/2023/2023_kev_list.html#top10list

Bounds Safety

- The good news: most standard library containers provide a bounds-checked `at()` function which throws an exception if the requested index is out of bounds
- Just use it!
- Problem solved?
- `std::span` doesn’t provide `at()` before C++26 (I tried…)
- Still doesn’t help for raw arrays, `std::initializer_list`, random-access views…

Bounds Safety

- Workaround (1): use a bounds-checked `at()` function for all random-access, sized ranges:

```
template <std::ranges::random_access_range R>
    requires std::ranges::sized_range<R> &&
             std::ranges::borrowed_range<R>
constexpr auto at(R&& rng, std::ranges::range_size_t<R> idx)
    -> std::ranges::range_reference_t<R>
{
    if (idx < std::ranges::range_size_t<R>{} || idx >= std::ranges::size(rng)) {
        throw std::out_of_range{
            std::format("Requested index {} is out of bounds "
                        "for range of size {}", idx,
                        std::ranges::size(rng))};
    }
    if constexpr (std::ranges::contiguous_range<R>) {
        return std::ranges::data(rng)[idx];
    } else {
        return std::ranges::begin(rng)[idx];
    }
}
```

Bounds Safety

- Workaround (2): Use Flux!

```cpp
#include <flux.hpp>

int main()
{
    int arr[] = {1, 2, 3};

    int i = flux::read_at(arr, 100'000); // bounds checked

    int j = flux::read_at_unchecked(arr, 100'000); // Danger!
}
```

Bounds Safety

- An even better alternative is to eliminate the need for bounds checks by using looping constructs that don’t expose indices/iterators

- Using algorithms and range adaptors can often avoid the need for manual indexing, without any runtime overhead

- Using well-tested, reusable algorithms and adaptors also reduce the probability of indexing errors and off-by-ones

- “No raw loops” — Sean Parent

Bounds Safety

- Compilers are surprisingly good at eliding bounds checks
- However, there can be run-time overhead in some cases
- If you really need to drop to unchecked access for performance reasons, try to make it obvious in your code
- There are techniques available for avoiding bounds checking overhead without compromising safety (using Rust, but applicable to C++ as well):
  - https://shnatsel.medium.com/how-to-avoid-bounds-checks-in-rust-without-unsafe

Bounds Safety: Summary

- Array access is unchecked in C++, including vectors etc by default
- This is one of the most common safety problems
- Use `.at()` when you can
- For other types, use a generic bounds-checking function, or a library that does this for you
- Even better: use well-tested generic algorithms which don’t risk bounds violations
- If bounds checking really adds unacceptable overhead, try to make the use of unchecked indexing obvious in your code

Memory Safety

Memory Safety

- To state the obvious: C++ is not a memory safe language

- To take just one example, pointers become dangling when the object they refer to is destroyed:

```cpp
int main()
{
    int* ptr = nullptr;
    {
        int i = 10;
        ptr = &i;
    }
    std::print("{}", *ptr);
}
```

Memory Safety

- But it’s not just raw pointers. There are many “reference-like” types in C++ which can also easily become invalidated:
  - Language references
  - Iterators
  - `std::string_view`
  - `std::span`
  - `std::reference_wrapper`
  - Lambdas with reference-like captures
  - …

Memory Safety

- Many forms of reference invalidation can be hard to spot unless you know what to look for:

```
auto iter = vec.cbegin();

vec.push_back(99);

while (iter != vec.cend()) {
    // ...
}
```

Memory Safety

- Rust, Swift and Hylo achieve their safety using the “Law of Exclusivity”
  - Modification requires exclusive access to an object
- In the Rust world, this principle is often known as “shared XOR mutable”
- At any point, there can be:
  - At most one mutable reference to an object
  - **Or** any number of const references
  - But not both at the same time

Memory Safety

- Obeying the Law of Exclusivity would be sufficient to prevent memory errors in C++ as well

```
auto iter = vec.cbegin(); // iterators are "reference-like"
                          // we now have a reference to the
                          // vector

vec.push_back(99); // Law of Exclusivity violation:
                   // modification of vector while
                   // reference is in use

while (iter != vec.cend()) { // UB
    // ...
}
```

Memory Safety

- In the absence of language and compiler support, strictly adhering to the Law of Exclusivity in large scale, real-world C++ codebases is phenomenally difficult
- But holding the LoE in mind while writing code and designing APIs can go a long way
- In particular, APIs in which the formation of references is explicit can make data and lifetime dependencies more obvious, and can highlight LoE violations (and thus potential problems)
- Example: parameter passing in the Flux library

Memory Safety

- In Flux, sequence adaptors are *sinks*, taking ownership of the sequences passed to them.
- Adaptor functions take their arguments (as if) by value
- In order for an adaptor to operate on a *reference* to a sequence, we need to explicitly create a reference wrapper, which then gets passed “by value"
- The `flux::ref()` function creates a read-only reference object referring to some other sequence
    - like `std::reference_wrapper<const S>`
- The `flux::mut_ref()` function creates a mutable reference object referring to some other sequence
    - like `std::reference_wrapper<S>`, but move-only

Memory Safety

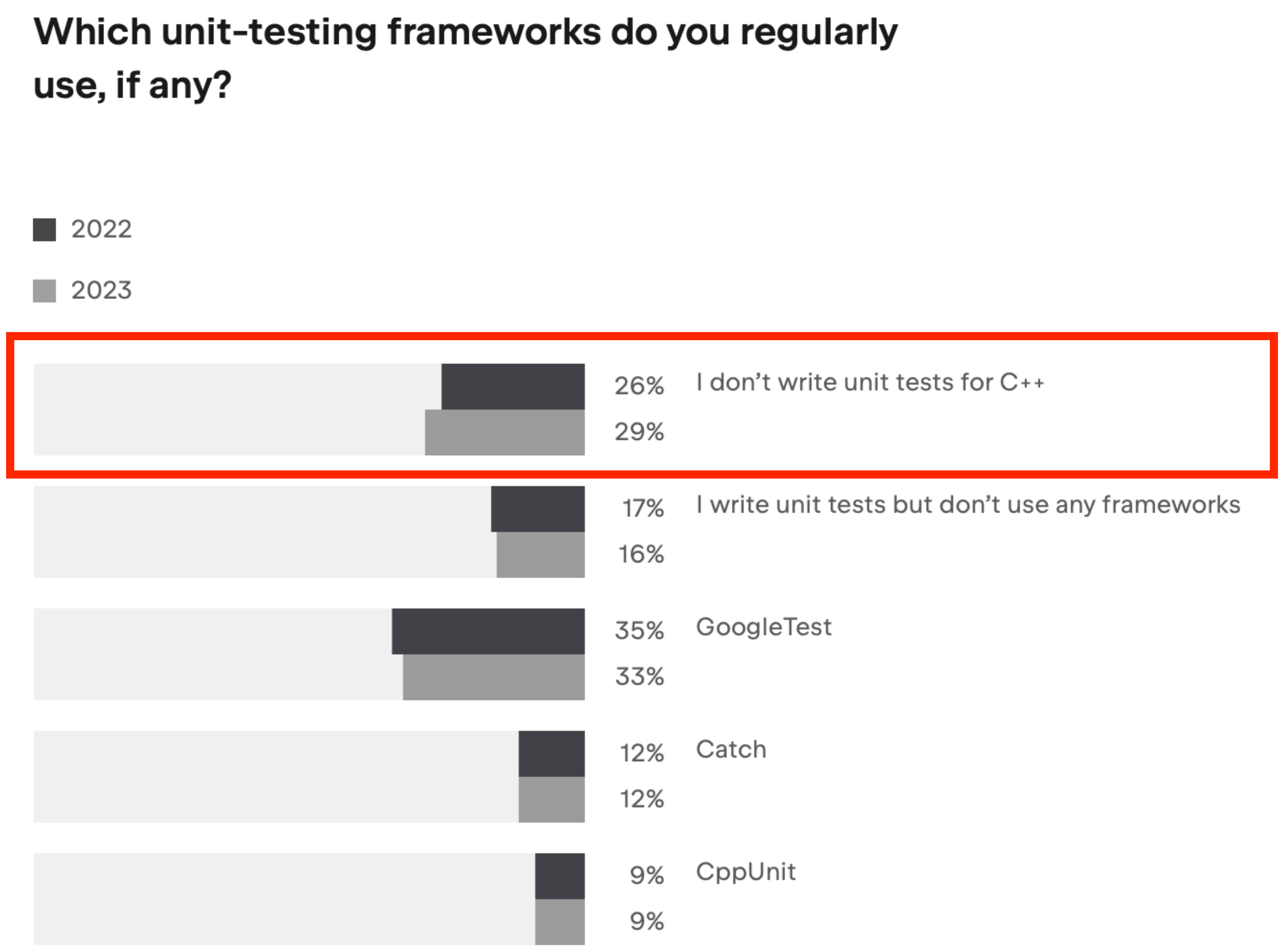
```
auto flux::zip(flux::sequence auto... seqs) -> flux::sequence auto;

auto z1 = flux::zip(auto(vec));
    // explicit copy, no data/lifetime dependency

auto z2 = flux::zip(std::move(vec));
    // explicit move, no data/lifetime dependency

auto z3 = flux::zip(flux::ref(vec));
    // explicit (const) reference, z3 has a dependency on vec

auto z4 = flux::zip(flux::ref(vec), flux::ref(another_vec));
    // z4 has dependencies on vec and another_vec

auto z5 = flux::zip(flux::ref(vec), flux::ref(vec));
    // Okay, two const references to the same object

auto z6 = flux::zip(flux::ref(vec), flux::mut_ref(vec));
    // Suspicious: const reference and mutable reference to the same object
    // Are we 100% sure that this is correct?
```

Memory Safety: Summary

- C++ is (obviously) a memory unsafe language
- The Law of Exclusivity is used by other native languages to ensure memory safety
- In principle, adhering to the LoE in C++ would give us memory safety too
  - Probably impossible in practice without language support
- API design can make data and lifetime dependencies more obvious in code, reducing the possibility of mistakes
- ...but there is no silver bullet (maybe Circle?)
- **Be careful out there**

Testing

Testing



- Source: JetBrains — The State of Developer Ecosystem survey 2023 (C++)
  - https://www.jetbrains.com/lp/devecosystem-2023/cpp/

Testing

- If surveys are accurate, a ~~terrifying~~ surprising number of C++ developers don’t write unit tests
- Don’t be one of them!
- Testing is important in all software domains
- But this is particularly true in a language like C++ that offers few safety guarantees

Testing

- There are many conference talks on how to write good unit tests
- They're packed with useful information

Kevlin Henney - Programming with GUTs - Meeting C++ 2021

Testing

- One particularly useful technique is **constexpr testing**

An expression E is a core constant expression unless the evaluation of E, following the rules of the abstract machine ([intro.execution]), would evaluate one of the following:

...

— an operation that would have undefined or erroneous behavior as specified in [intro] through [cpp], excluding [dcl.attr.assume] and [dcl.attr.noreturn];[70]

- Translated: the compiler is required to diagnose undefined behaviour that occurs during compile-time evaluation

Testing

- Given a constexpr function, we can write a constexpr test and evaluate it using `static_assert`
- We gain a free, compile-time UB check courtesy of the compiler!

Testing

```
constexpr int add(int a, int b)
{
    return a + b;
}

constexpr bool test_add()
{
    if (add(3, 4) != 7) { throw test_failure{}; }
    // ...
    if (add(INT_MAX, 1) != INT_MIN) { throw test_failure{}; }

    return true;
}
static_assert(test_add());
```

Testing

```
<source>:53:23: error: non-constant condition for static assertion
   53 | static_assert(test_add());
      |               ~~~~~~~~^~
<source>:53:23:   in 'constexpr' expansion of 'test_add()'
<source>:49:12:   in 'constexpr' expansion of 'add(2147483647, 1)'
<source>:42:14: error: overflow in constant expression [-fpermissive]
   42 |     return a + b;
      |            ~~^~~
Compiler returned: 1
```

Testing

- Suggestion: make functions `constexpr` when possible in order to facilitate compile-time testing
- If compile times are a concern, place constexpr tests within `#ifdef`s
- Constexpr tests may be our best defence against undefined behaviour
- Almost the whole of Flux is tested this way

Testing

- The next best defence against UB is testing using sanitizers:
  - UB Sanitizer: detects some forms of undefined behaviour at runtime
  - Address Sanitizer: detects certain memory errors, e.g. use-after-free, buffer overflow
  - Memory Sanitizer: detects the use of uninitialised memory
  - Thread Sanitizer: detects data races
- Run your tests regularly with sanitizers, in both debug and release configurations
- If possible, compile regularly with more than one major compiler
  - And, obviously, enable all the warnings you can. The compiler is trying to help you!

Testing

- Cannot stress this enough: these approaches can only find problems in code that they execute
- Thorough tests are **essential**
- Don’t be one of the 29%!

Testing: Summary

- If surveys are to be believed, an alarming number of C++ developers don’t write tests
- The importance of good unit tests cannot be overstated
- Constexpr testing can be used to detect undefined behaviour at compile time
- Regularly build and run your tests with sanitizers to help detect problems at run time
- Building with multiple compilers will help improve code quality

Conclusion

- The absence of undefined behaviour is a prerequisite for safe code
- Be aware of the many kinds of UB that can occur in C++
- There are various libraries and coding techniques that can help you avoid common UB pitfalls — use them!
- Write tests that help you detect accidental UB
  - At compile time or at run time using sanitisers
- Good luck out there!

Thanks for coming!

Questions?