std::forward from std::initializer_list

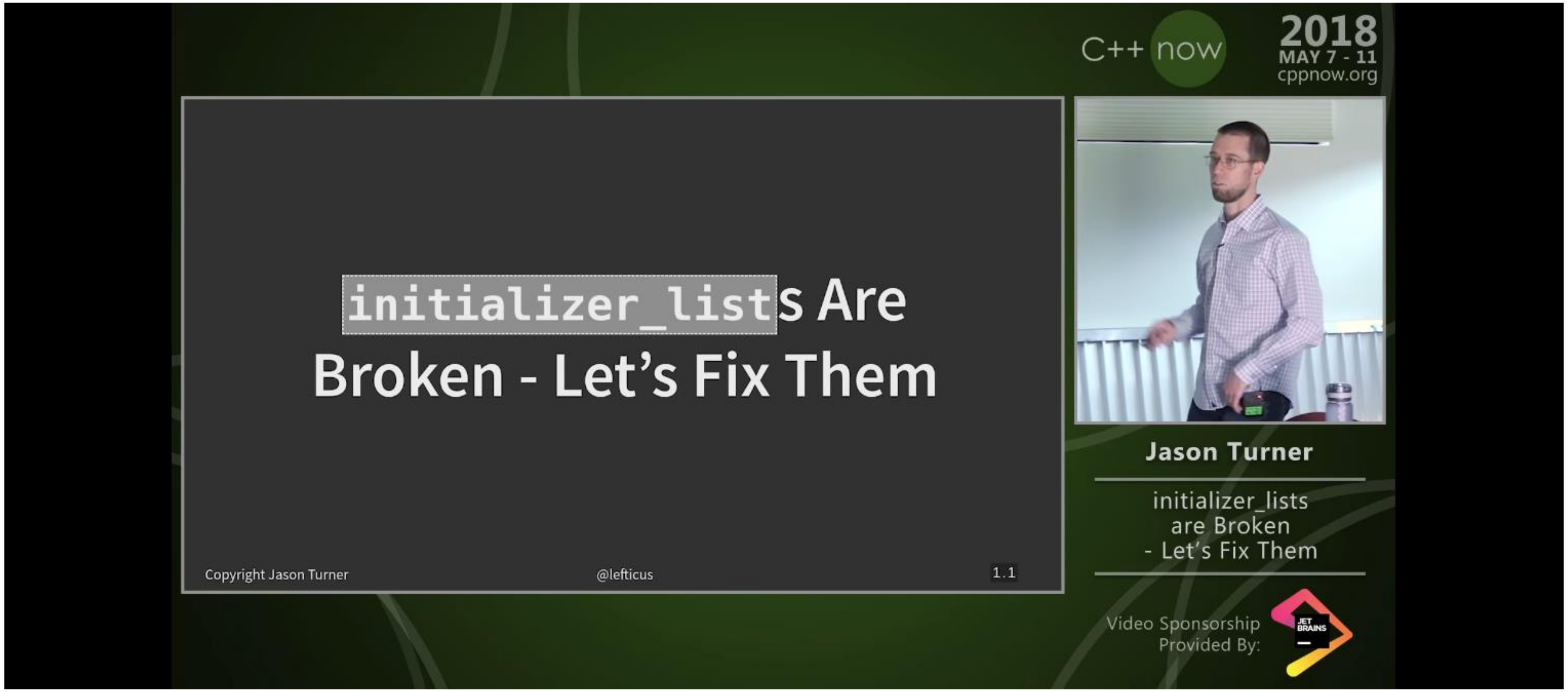
P1249 – A paper targeting C++20

Alex Christensen | Apple | achristensen@apple.com

Presented by: JeanHeyd Meneide | phdoftheouse@gmail.com

Motivation I

- `std::initializer_list` is fundamentally broken for a large class of language types, making it absolutely unusable in efficient contexts
  - Fixing it from the outside is hard and complicated



initializer_lists Are Broken - Let’s Fix Them

Copyright Jason Turner @lefticus 1.1

C++ now
2018
MAY 7 - 11
cppnow.org
Jason Turner
initializer_lists are Broken - Let’s Fix Them
Video Sponsorship Provided By:
JET BRAINS

Motivation II

```
void sadness() {
    // error: object of type
    // 'std::__1::unique_ptr<int,
    //          std::__1::default_delete<int> >’
    // cannot be assigned because its
    // copy assignment operator is implicitly deleted
    // buffer[size++] = element;
    Vector<std::unique_ptr<int>> v2({
        std::make_unique<int>(3),
        std::make_unique<int>(4)
    });
}
```

Prior Art

- Several papers went forward with `movable_initializer_list`/`own_initializer_list`
  - All have failed despite consensus encouraging them
  - Suffers from problem of “well we develop complex rules to let compiler pick owning version of not”

- No papers try to fundamentally change definition of `initializer_list`: none mention why they do not go this route
  - Previous authors assume immutability of Core Wording, always tackle problem from external/library view first

Scalable Solution

- Take `const` off both `initializer_list` and also off [§9.3.5, clause 5 [dcl.init.list]]
  - Non-const iterators now return non-const elements
  - Standardese would no longer require backing storage to be const

- Do not need compiler rules about when to create movable initializer lists
  - No special casing to bite user in generic contexts (!!)

Root of All Evil?

- Did the C++ Standards Committee prematurely optimize here?

- `const` storage to have things put in read-only memory (e.g. .data) too prematurely when this feature was first conceived?
  - No vocal objections from mailing list when paper was brought up
  - Nobody I talked to or discussed this with could give me a good reason
  - Standard has example that specifically calls out it is okay to have constructed initializer list in “read-only memory”.

Why?

- Consider

  ```
  std::vector<int> v1{
        1, 2, 3
  }
  ```

  - How many times does this appear outside of example / slide code?

- Consider:

  ```
  std::vector<int> v2{
        func_parameter + static_variable – variadic_arg_0, …
  }
  ```

  - Much more realistic: how would this ever get put into read-only memory?

Breaking Changes?

```
Vector(std::initializer_list<T>&& list) :
buffer(std::make_unique<T[]>(list.size())) {
    for (auto&& element : list) {
        // Calls a different function
        checkConst(element);
        buffer[size++] = element;

    }
}
void checkConst(T&) {
    std::cout << "non-const" << std::endl;
}
void checkConst(const T&) {
    std::cout << "const" << std::endl;
}
```

Breaking Changes: Current Forecast

- For containers which essentially divert to `insert`/`push_back`, likely to not be much of a problem
- For other use cases, if the user only wrote const-qualified functions to handle the difference, no change in runtime is observable
  - User cannot have written non-const qualified version without explicitly `const_cast`-ing, in which case they violated the standard to begin with and nothing we can do will help them

Wording Complete

- Wording is extraordinarily simple:
  - Just removes const from both the Core wording and the `initializer_list` specification
  - Will require a quick review in Core to see if we might be missing anything

Poll

- Forward to EWG?

Strongly in Favor	In Favor	Neutral	Against	Strongly Against