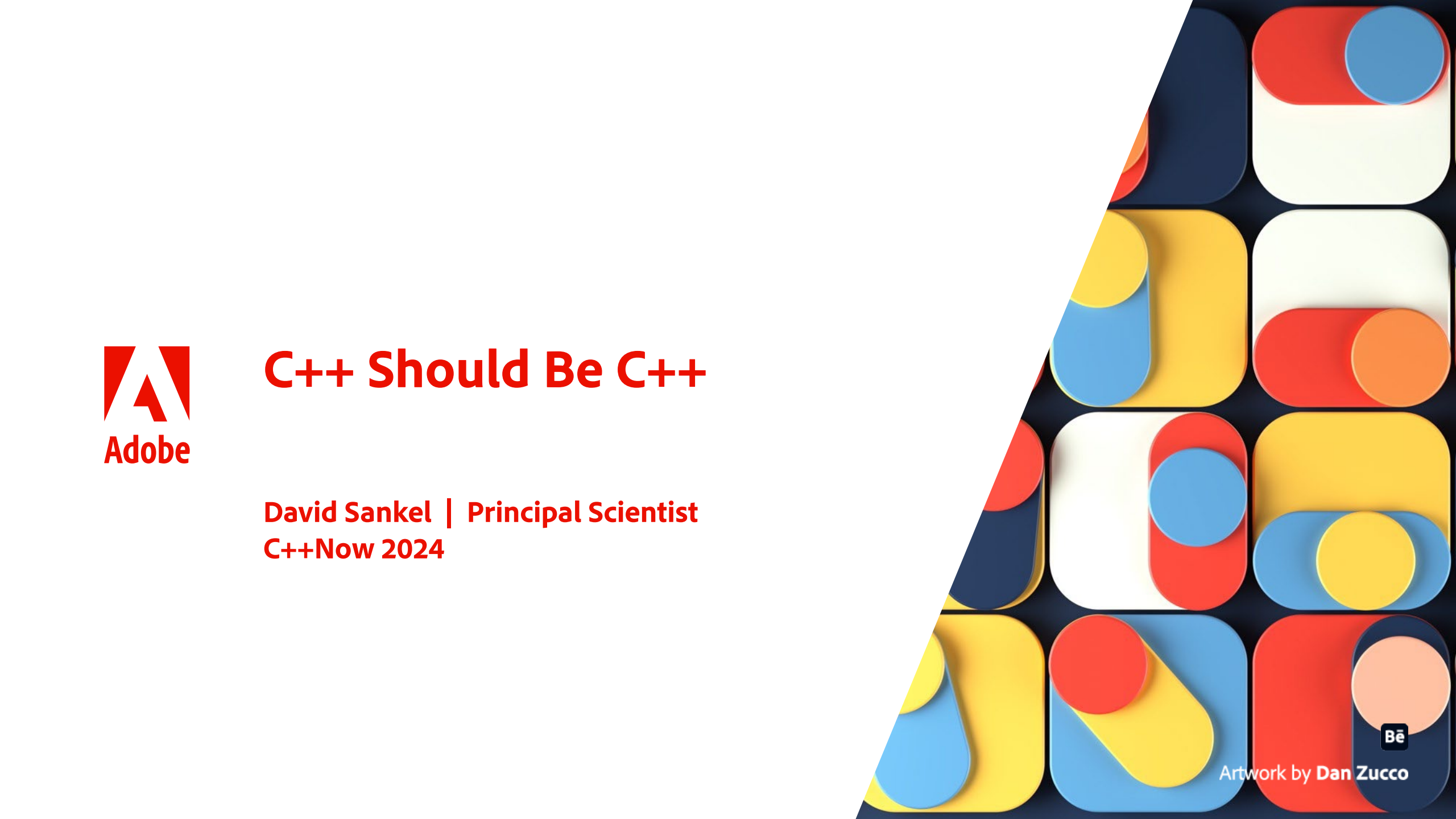
C++ now

C++ Should Be C++

David Sankel

2024

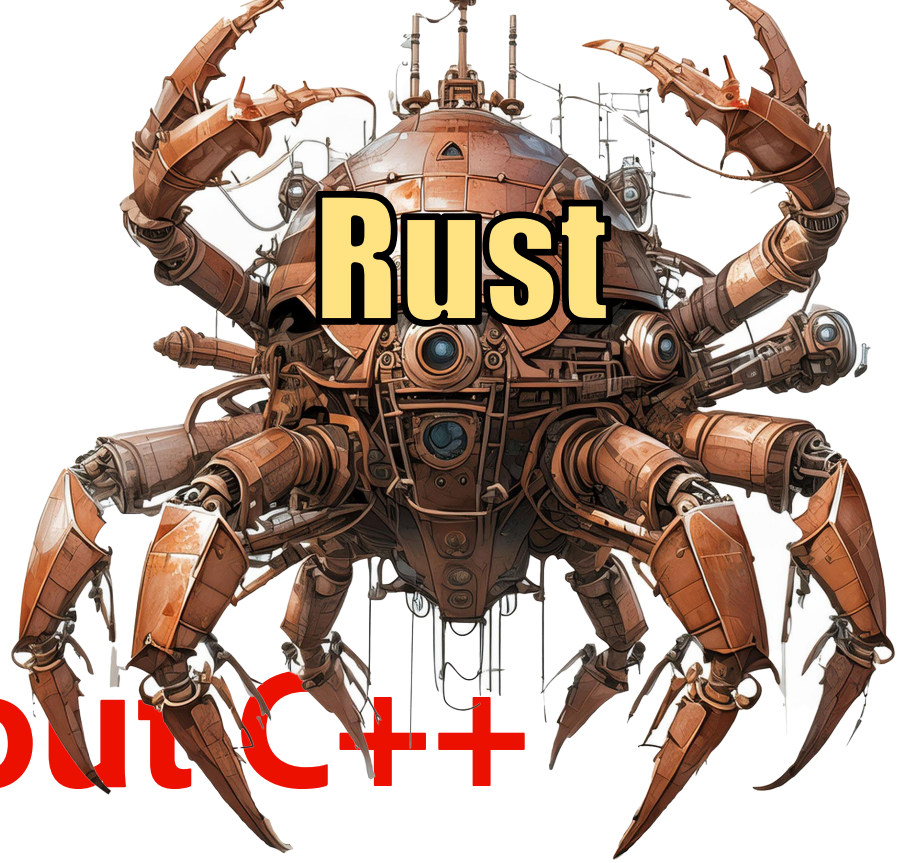


C++ Should Be C++

David Sankel | Principal Scientist

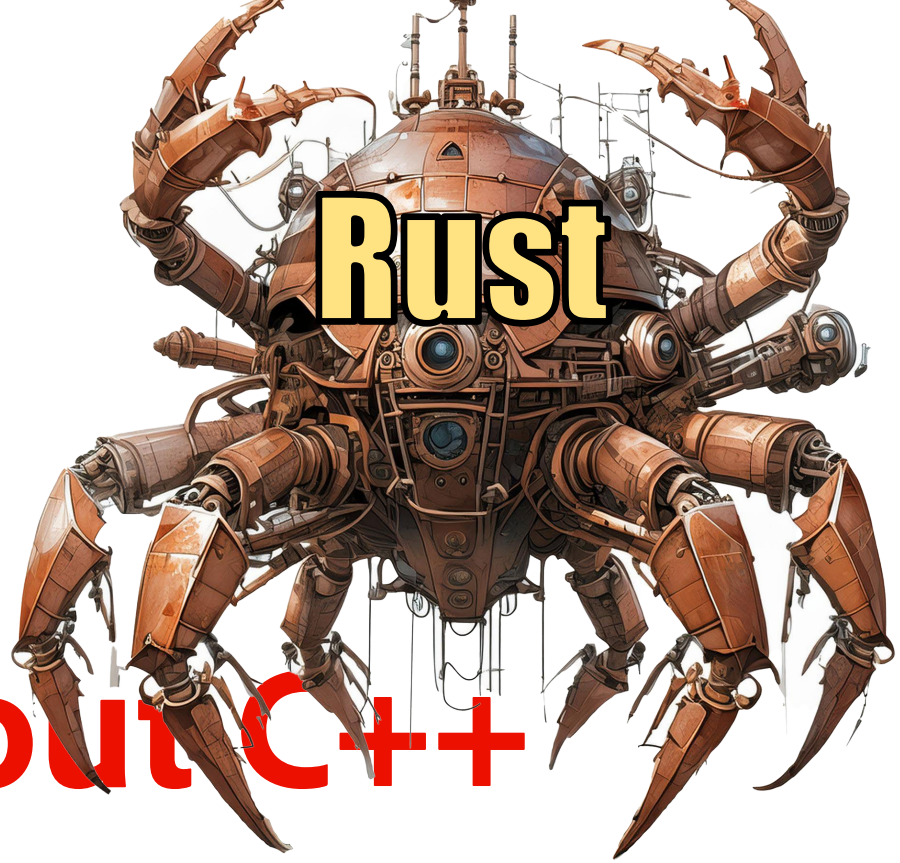
C++Now 2024

Let’s talk about C++



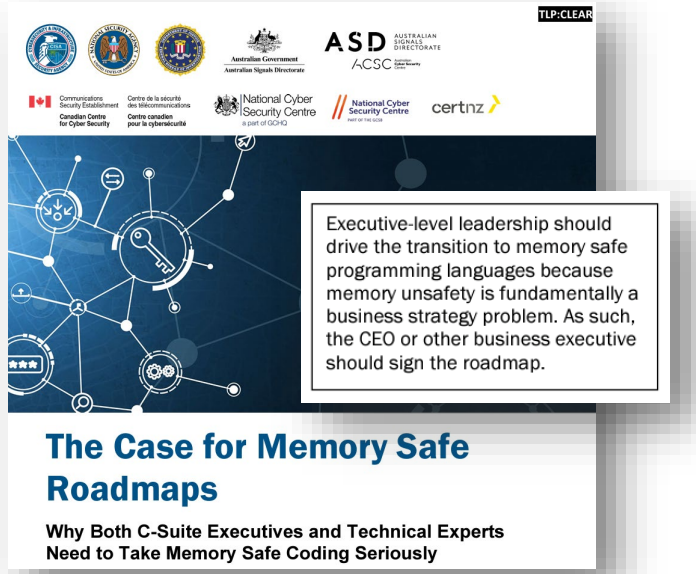
Let’s talk about C++

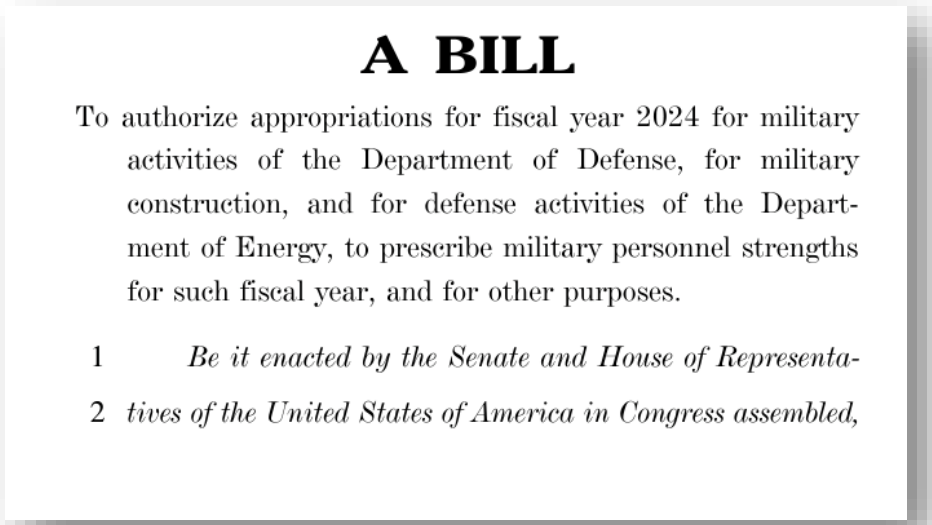
Let’s talk about C++



Let’s talk about C++

"C and C++…are not memory safe languages…[unlike] Rust."

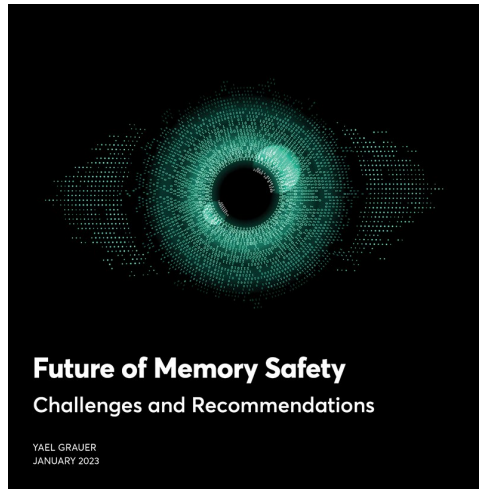




Rust

Requires new DOD policy establishing "conditions and associated approval process" for … buying software using memory-unsafe languages

"[S]ome resistance to moving off C/C++ is due to the sunk cost fallacy…using Rust for new projects will ultimately result in higher productivity."

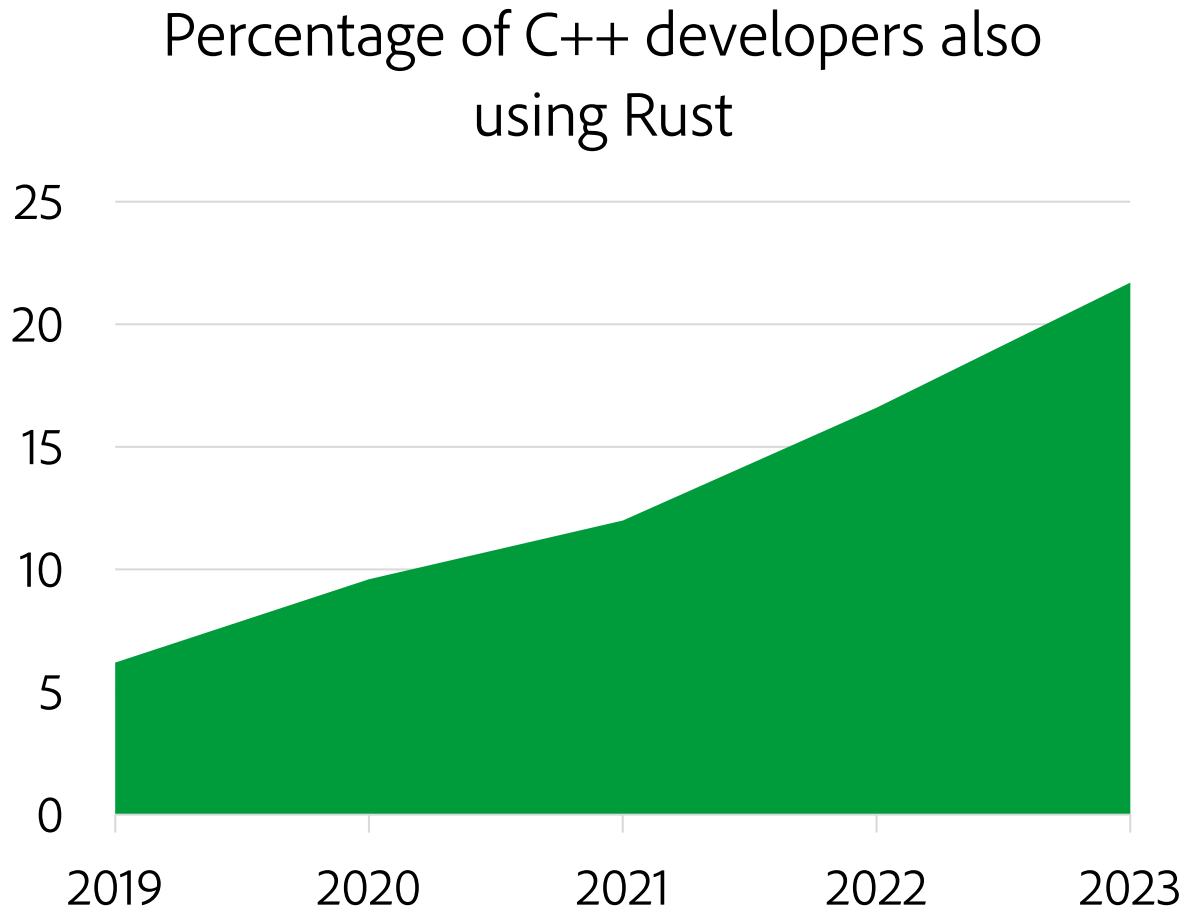


Bjarne: [I]f I considered any of those “safe” languages superior to C++ for the range of uses I care about, I wouldn’t consider the fading out of C/C++ as a bad thing, but that’s not the case

Herb: in C++ a 98% improvement in the four most common problem areas is achievable in the medium term.

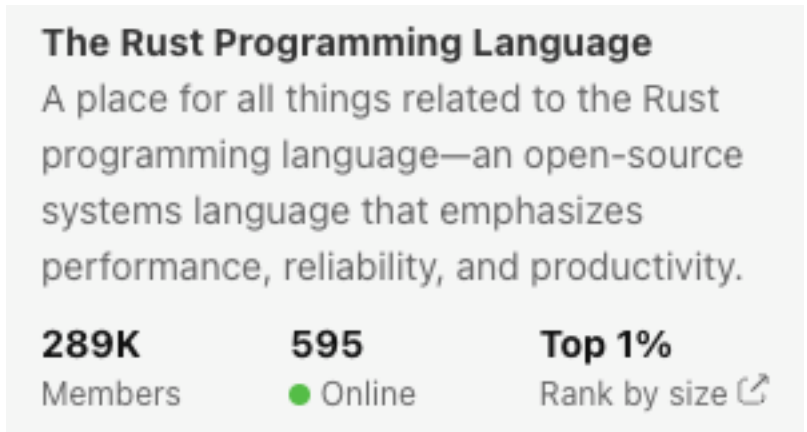

Rust
C++

- StackOverflow annual survey
- 90,000 people



2024 C++ Foundation Survey

18.7% of polled C++ engineers said Rust is another tool they use



For every 5 C++ PRs on GitHub, there's 1 Rust PR


THIS IS
FINE

Let’s talk about C++

Let’s talk about C

Let's talk about C

- Like C++, but with a lot less stuff
- Quite popular (~1 C PR for every 2 C++ PRs)

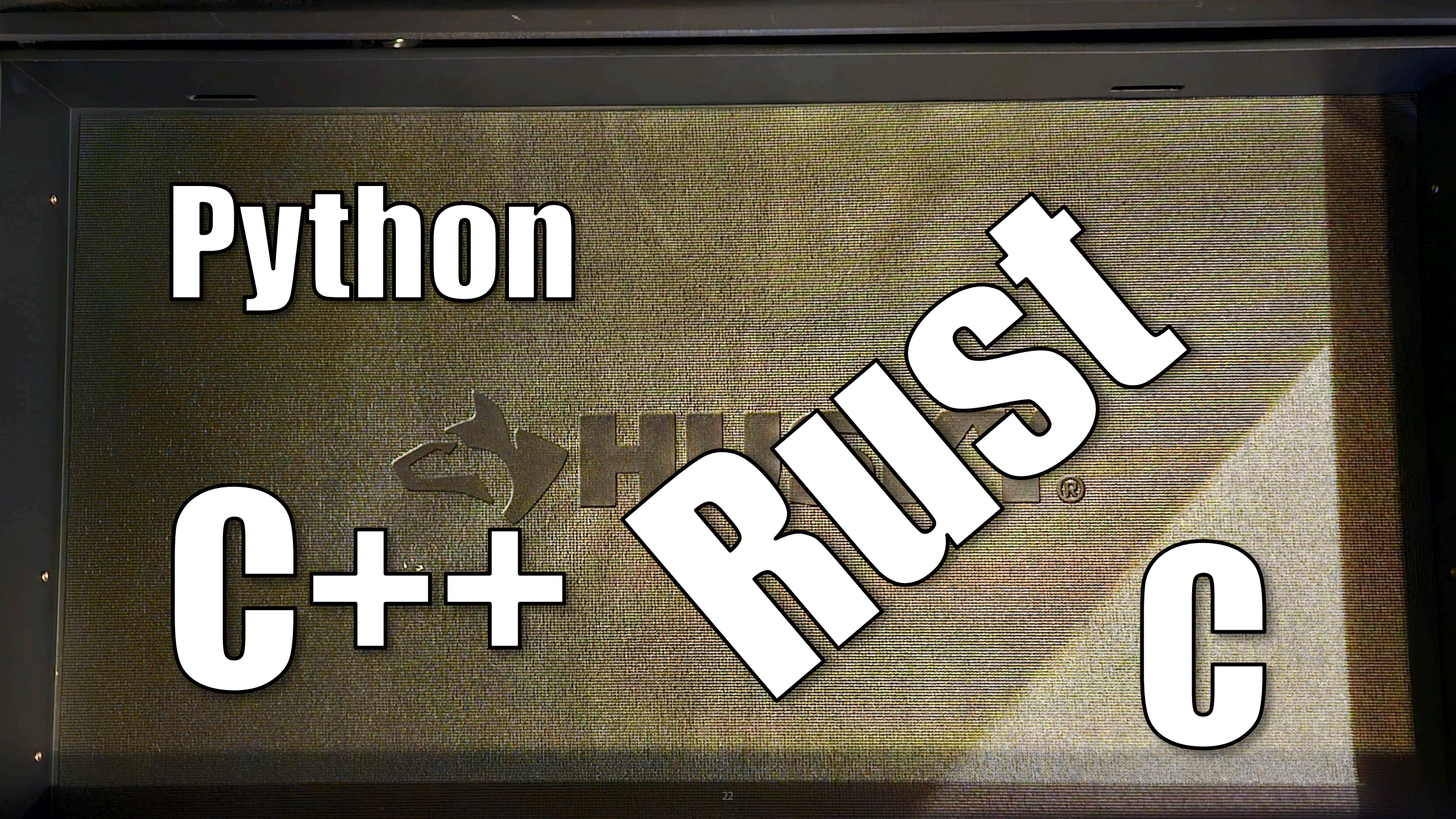
Why is C used so much?

- Libraries with C interfaces are very accessible
  - Latin of Programming Languages
  - OpenGL, Operating Systems, POSIX, etc.
- Easy to learn/use
- Low-level access without too much fuss
- Fast
- Stable

Why didn’t C++ supplant C?


RIKON
Life is better on the tips

LINCOLN ELECTRIC


Python
C++
RUST
C

Do we need C++ in our toolbox if we also have Rust?

Yes!

- There's all that existing C++ code
- There's all that existing "unsafe" C code
- Rust's design has a bad tradeoff set for some problems

Case study: Rust gamedev

- Higher up-front design cost
- Idea exploration not rapid
- Overly abstract

C++ is uniquely fit for several domains

- Abstracting low-level C stuff
  - Embedded
  - Real time
- Scientific computing
- Performance-critical rapid prototyping
- What else?

Key observation: new tools don’t make old tools less capable

But, there is something that can make C++ less capable…

How is the committee a danger to C++'s utility?

First X3J16 meeting

Somerset, NJ, USA (1990)

Completed C++11

Madrid, Spain (2011)

Why do people join the committee?

Why do people join the committee?

- Represent corporate interests
- Desire to change C++'s direction
- Stewardship of the language
- Integrate particular features
- Notoriety/recognition

When stewardship gets overshadowed…

- Incoherence
- Complication
- Scattered direction

Case study: std::simd

Fundamentals of Generic Programming (James C. Dehnert and Alexander Stepanov) 1998

1. T a = b; assert(a==b);
2. T a; a = b; ⇔ T a = b;
3. T a = c; T b = c; a = d; assert(b==c)
4. T a = c; T b = c; zap(a); assert(b==c && a!=b) where zap always changes its operand's value.

Regularity buys us a lot

- Algorithms (e.g. std::find)
- Language features (e.g. bool operator==(const T&) const = default)
- Lower teaching costs

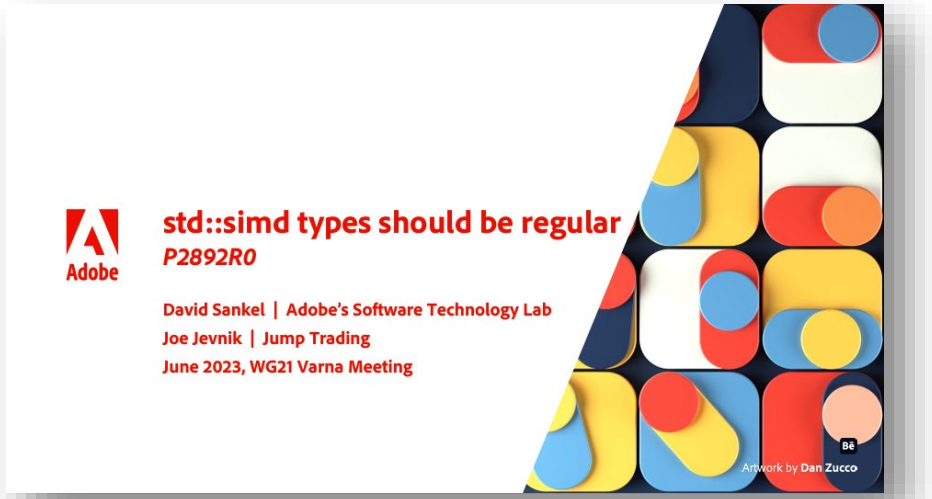
std::simd proposal has operator== return a mask

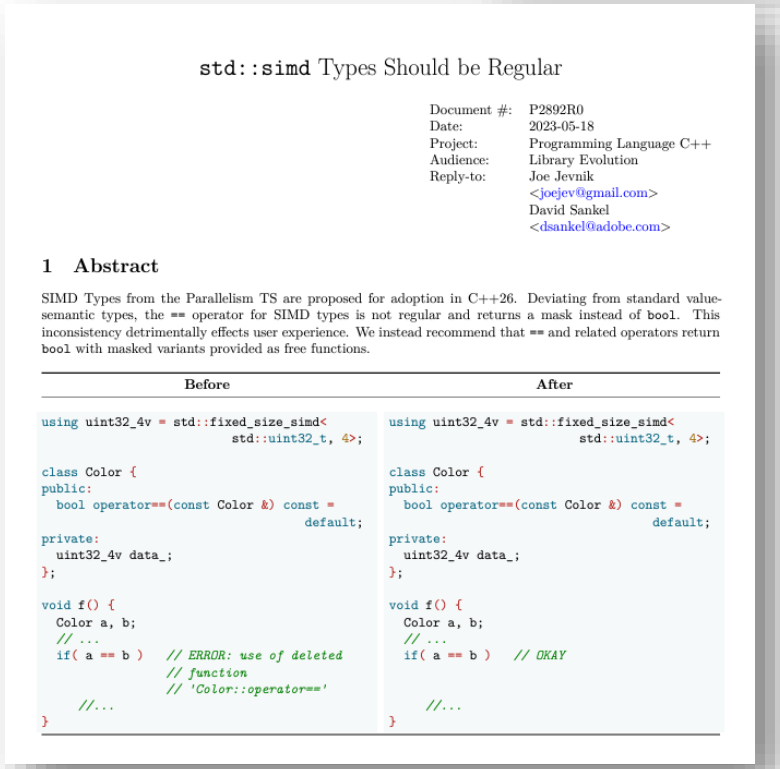
Might make sense *if looking only at SIMD*

- Consistent with +
- Other SIMD libraries do this
- Maybe a different value semantic category

Regularity didn't win out

- Attempts to address this were consistently met with calls to avoid discussion
- No comparative direction polls
- Non-regular SIMD is what will likely be in C++26





`std::simd` Types Should be Regular

Document #:	P2892R0
Date:	2023-05-18
Project:	Programming Language C++
Audience:	Library Evolution
Reply-to:	Joe Jevnik <joejev@gmail.com>, David Sankel <dsankel@adobe.com>

1 Abstract

SIMD Types from the Parallelism TS are proposed for adoption in C++26. Deviating from standard value-semantic types, the == operator for SIMD types is not regular and returns a mask instead of bool. This inconsistency detrimentally effects user experience. We instead recommend that == and related operators return bool with masked variants provided as free functions.

Before

```
using uint32_4v = std::fixed_size_simd<
                        std::uint32_t, 4>;

class Color {
public:
  bool operator==(const Color &) const =
                                    default;
private:
  uint32_4v data_;
};

void f() {
  Color a, b;
  // ...
  if( a == b )   // ERROR: use of deleted
                 // function
                 // 'Color::operator=='
    //...
}
```

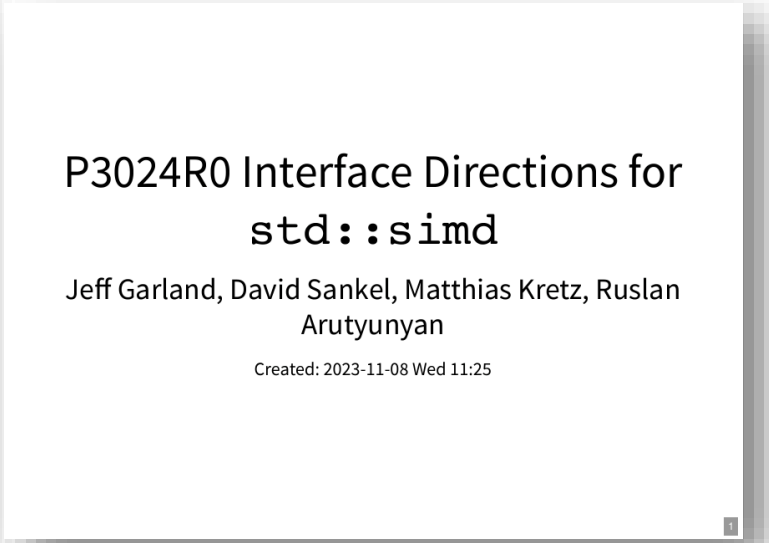
After

```
using uint32_4v = std::fixed_size_simd<
                        std::uint32_t, 4>;

class Color {
public:
  bool operator==(const Color &) const =
                                    default;
private:
  uint32_4v data_;
};

void f() {
  Color a, b;
  // ...
  if( a == b )   // OKAY

    //...
}
```



P3024R0 Interface Directions for `std::simd`

Jeff Garland, David Sankel, Matthias Kretz, Ruslan Arutyunyan

Created: 2023-11-08 Wed 11:25

A troubling trend: expert friendly

[The] “average programmer”…is seriously underrepresented on the committee

- H. Hinnant et al., Direction for ISO C++

Case study: thread attributes

Thread attributes

Document #:	P2019R5
Date:	2024-01-13
Programming Language C++	
Audience:	LEWG
Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>

Abstract

We propose a way to set a thread stack size and name before the start of its execution, both of which are, as we demonstrate, current practices in many domains.

The absence of these features make `std::thread` and `std::jthread` unfit or unsatisfactory for many use cases.

Wording details…

```
template<class F, class... Args>
explicit thread(F&& f, Args&&... args);
```

Constraints:

- `sizeof...(Args) > 0` is `true`, and
- `remove_cvref_t<F Args[0]>` is not the same type as `thread`.

Mandates: The following are all `true`:

- No type is present more than once in the pack `remove_cvref_t<Attrs>`. [*Editor's note:* // is there a better way to say that ?]

`attrs` can be used to tailor the thread with additional implementation-defined behaviors. (see [thread.attributes]).

- Let i be the smallest value such that `decay_t<Args...[i]>` is not a thread attribute type. [*Editor's note:* We need to define that]

  If no such i exists, the program is ill-formed.
- Let `F` be `Args...[i]`.
- Let `f` be `args...[i]`.
- Let `attrs` a pack of the expressions `args...[j]` for each j such that $0 \leq j < i$.
- Let `FArgs` a pack of the types `Args...[j]` for each j such that $i \leq j \leq$ `sizeof...(args)`.
- Let `fargs` a pack of the expressions `args...[j]` for each j such that $i \leq j \leq$ `sizeof...(args)`.

Wording details…

```
template<class F, class... Args>
explicit thread(F&& f, Args&&... args);
```

Constraints:

- `sizeof...(Args) > 0` is `true`, and
- `remove_cvref_t<F Args[0]>` is not the same type as `thread`.

- Let i be the smallest value such that `decay_t<Args...[i]>` is not a thread attribute type. [*Editor's note:* We need to define that]

  If no such i exists, the program is ill-formed.
- Let `F` be `Args...[i]`.
- Let `f` be `args...[i]`.
- Let `attrs` a pack of the expressions `args...[j]` for each j such that $0 \leq j < i$.
- Let `FArgs` a pack of the types `Args...[j]` for each j such that $i \leq j \leq$ `sizeof...(args)`.
- Let `fargs` a pack of the expressions `args...[j]` for each j such that $i \leq j \leq$ `sizeof...(args)`.

Mandates: The following are all `true`:

- No type is present more than once in the pack `remove_cvref_t<Attrs>`. [*Editor's note:* // is there a better way to say that ?]

`attrs` can be used to tailor the thread with additional implementation-defined behaviors. (see [thread.attributes]).

Wording details…

```
template<class F, class... Args>
explicit thread(F&& f, Args&&... args);
```

Constraints:

- `sizeof...(Args) > 0` is `true`, and
- `remove_cvref_t<F Args[0]>` is not the same type as `thread`.

Mandates: The following are all `true`:

- No type is present more than once in the pack `remove_cvref_t<Attrs>`. [*Editor's note:* // is there a better way to say that ?]

`attrs` can be used to tailor the thread with additional implementation-defined behaviors. (see [thread.attributes]).

- Let i be the smallest value such that `decay_t<Args...[i]>` is not a thread attribute type. [*Editor's note:* We need to define that]

  If no such i exists, the program is ill-formed.
- Let `F` be `Args...[i]`.
- Let `f` be `args...[i]`.
- Let `attrs` a pack of the expressions `args...[j]` for each j such that $0 \leq j < i$.
- Let `FArgs` a pack of the types `Args...[j]` for each j such that $i \leq j \leq$ `sizeof...(args)`.
- Let `fargs` a pack of the expressions `args...[j]` for each j such that $i \leq j \leq$ `sizeof...(args)`.

Wording details…

```
template<~~class F,~~ class... Args>
explicit thread(~~F&& f,~~ Args&&... args);
```

Constraints:

- `sizeof...(Args) > 0` is `true`, and
- `remove_cvref_t<~~F~~ Args[0]>` is not the same type as `thread`.

Mandates: The following are all `true`:

- No type is present more than once in the pack `remove_cvref_t<Attrs>`. [*Editor's note:* // is there a better way to say that ?]

`attrs` can be used to tailor the thread with additional implementation-defined behaviors. (see [thread.attributes]).

- Let i be the smallest value such that `decay_t<Args...[i]>` is not a thread attribute type. [*Editor's note:* We need to define that]

  If no such i exists, the program is ill-formed.
- Let `F` be `Args...[i]`.
- Let `f` be `args...[i]`.
- Let `attrs` a pack of the expressions `args...[j]` for each j such that $0 \leq j < i$.
- Let `FArgs` a pack of the types `Args...[j]` for each j such that $i \leq j \leq$ `sizeof...(args)`.
- Let `fargs` a pack of the expressions `args...[j]` for each j such that $i \leq j \leq$ `sizeof...(args)`.

Wording details…

```
template<~~class F,~~ class... Args>
explicit thread(~~F&& f,~~ Args&&... args);
```

Constraints:

- si
- re

- Let i [Edito

  If no such i exists, the program is ill-formed.

```
void f(int);
int main() {
    std::jthread thread(std::thread_name_hint("Worker"), std::thread_stack_size_hint(512*1024)
    , f, 42);
    return 0;
}
```

- Let F be Args...[i].
- Let f be args...[i].
- Let attrs a pack of the expressions args...[j] for each j such that $0 \leq j < i$.
- Let FArgs a pack of the types Args...[j] for each j such that $i \leq j \leq$ sizeof...(args).
- Let fargs a pack of the expressions args...[j] for each j such that $i \leq j \leq$ sizeof...(args).

Mandates: The following are all true:

- No type is present more than once in the pack remove_cvref_t<Attrs>. [*Editor's note:* // is there a better way to say that ?]

attrs can be used to tailor the thread with additional implementation-defined behaviors. (see [thread.attributes]).

Doc. no.:	P3072R1
Date:	2024-2-15
Audience:	LEWG
Reply-to:	Zhihao Yuan <zy@miator.net>

```
std::jthread thr({.name = "worker", .stack
                  [] { std::puts("
```

Rejected!

- Make thread::attributes platform-independent
- Declare the name attribute as string const&
- Default a template parameter to thread::attributes
- Alias jthread::attributes to thread::attributes

- Implementation
- Wording
- References

Case study: Sender/Receiver

```
template <class S>
struct _retry_sender {

  template <class Env>
  friend auto tag_invoke(stdexec::get_completion_signatures_t,
                         const _retry_sender&, Env) -> /*blah*/;

  template <stdexec::receiver R>
  friend auto tag_invoke(stdexec::connect_t,
                         _retry_sender&& self,
                         R r) -> /*blah*/;

  friend auto tag_invoke(stdexec::get_env_t,
                         const _retry_sender& self) -> /*blah*/;
};
```

Case study: Sender/Receiver

```
template <class S>
struct _retry_sender {

  template <class Env>
  friend auto tag_invoke(stdexec::get_completion_signatures_t,
                         const _retry_sender&, Env) -> /*blah*/;

  template <stdexec::receiver R>
  friend auto tag_invoke(stdexec::connect_t,
                         _retry_sender&& self,
                         R r) -> /*blah*/;

  friend auto tag_invoke(stdexec::get_env_t,
                         const _retry_sender& self) -> /*blah*/;
};
```

Case study: Sender/Receiver

```
template <class S>
struct _retry_sender {

  template <class Env>
  friend auto tag_invoke(stdexec::get_completion_signatures_t,
                         const _retry_sender&, Env) -> /*blah*/;

  template <stdexec::receiver R>
  friend auto tag_invoke(stdexec::connect_t,
                         _retry_sender&& self,
                         R r) -> /*blah*/;

  friend auto tag_invoke(stdexec::get_env_t,
                         const _retry_sender& self) -> /*blah*/;
};
```

Case study: Sender/Receiver

```
template <class S>
struct _retry_sender {

  template <class Env>
  friend auto tag_invoke(stdexec::get_completion_signatures_t,
                         const _retry_sender&, Env) -> /*blah*/;

  template <stdexec::receiver R>
  friend auto tag_invoke(stdexec::connect_t,
                         _retry_sender&& self,
                         R r) -> /*blah*/;

  friend auto tag_invoke(stdexec::get_env_t,
                         const _retry_sender& self) -> /*blah*/;
};
```

Case study: Sender/Receiver

```
template <class S>
struct _retry_sender {

  template <class Env>
  friend auto tag_invoke(stdexec::get_completion_signatures_t,
                         const _retry_sender&, Env) -> /*blah*/;

  template <stdexec::receiver R>
  friend auto tag_invoke(stdexec::connect_t,
                         _retry_sender&& self,
                         R r) -> /*blah*/;

  friend auto tag_invoke(stdexec::get_env_t,
                         const _retry_sender& self) -> /*blah*/;
};
```

Case study: Sender/Receiver

Document number: P2855R1
Audience: LEWG

```
template <class S>
struct _retry_sender {

  template <class Env>
  auto get_completion_signatures(const _retry_sender&, Env) -> /*blah*/;

  template <stdexec::receiver R>
  auto connect(_retry_sender&& self, R r) -> /*blah*/;

  auto get_env(const _retry_sender& self) -> /*blah*/;
};
```

This is because non-queries don't need to forward calls to customization points, but it's useful for queries to be able to forward queries.

In order to be able to write perfect-forwarding function templates that work both for lvalues and rvalues, we use deduced this. When there is no need to write a single function for both lvalues and rvalues, a traditional non-static member function will do.

Case study: Sender/Receiver

Document number: P2855R1
Audience: LEWG

```
template <class S>
struct _retry_sender {

  template <class Env>
  auto get_completion_signatures(const _retry_sender&, Env) -> /*blah*/;

  template <stdexec::receiver R>
  auto connect(_retry_sender&& self, R r) -> /*blah*/;

  auto get_env(const _retry_sender& self) -> /*blah*/;
};
```

This is because non-queries don't need to forward calls to customization points, but it's useful for queries to be able to forward queries.

In order to be able to write perfect-forwarding function templates that work both for lvalues and rvalues, we use deduced this. When there is no need to write a single function for both lvalues and rvalues, a traditional non-static member function will do.

Case study: Sender/Receiver

Document number: P2855R1
Audience: LEWG

```
template <class S>
struct _retry_sender {

  template <class Env>
  auto get_completion_signatures(const _retry_sender&, Env) -> /*blah*/;

  template <stdexec::receiver R>
  auto connect(_retry_sender&& self, R r) -> /*blah*/;

  auto get_env(const _retry_sender& self) -> /*blah*/;
};
```

This is because non-queries don't need to forward calls to customization points, but it's useful for queries to be able to forward queries.

In order to be able to write perfect-forwarding function templates that work both for lvalues and rvalues, we use deduced this. When there is no need to write a single function for both lvalues and rvalues, a traditional non-static member function will do.

Case study: Sender/Receiver

Document number: P2855R1
Audience: LEWG

```
template <class S>
struct _retry_sender {

  template <class Env>
  auto get_completion_si[illegible] -> /*blah*/;

  template[illegible]:rec[illegible]
  auto con[illegible] self, R r) -> /*blah*/;

  auto get_[illegible]v(const _retry_sender& self) -> /*blah*/;
};
```

Accepted!

This is because non-queries don't need to forward calls to customization points, but it's useful for queries to be able to forward queries.

In order to be able to write perfect-forwarding function templates that work both for lvalues and rvalues, we use deduced this. When there is no need to write a single function for both lvalues and rvalues, a traditional non-static member function will do.

std::optional vs. std::maybe_view

`std::optional<V>`	both	Proposed `std::maybe_view<V>`
`Opt(nullopt)`	`Opt()`	
	`Opt(v)`	
	`Opt(in_place, v)`	
`o = nullopt;`	`o = v;`	
`o.emplace(v);`		
`o.reset();`		
`o.swap(o2);`		
	`o == o2` and `o <=> o2`	
	`o == v` and `o <=> v`	
`*o` and `o->m`	`o.transform(f)`	`o.begin()` and `o.end()`
`o.has_value()` and `bool(o)`	`o.and_then(f)`	`o.size()`
`o.value_or(v)`	`o.or_else(f)`	`o.data()`
`std::hash<Opt>{}(o)`		

std::optional vs. std::maybe_view

`std::optional<V>`	both	Proposed `std::maybe_view<V>`
`Opt(nullopt)`	`Opt()`	
	`Opt(v)`	
	`Opt(in_place, v)`	
`o = nullopt;`	`o = v;`	
`o.emplace(v);`		
`o.reset();`		
`o.swap(o2);`		
	`o == o2` and `o <=> o2`	
	`o == v` and `o <=> v`	
`*o` and `o->m`	`o.transform(f)`	`o.begin()` and `o.end()`
`o.has_value()` and `bool(o)`	`o.and_then(f)`	`o.size()`
`o.value_or(v)`	`o.or_else(f)`	`o.data()`
`std::hash<Opt>{}(o)`		

std::optional vs. std::maybe_view

`std::optional<V>`	both	Proposed `std::maybe_view<V>`
`Opt(nullopt)`	`Opt()`	
	`Opt(v)`	
	`Opt(in_place, v)`	
`o = nullopt;`	`o = v;`	
`o.emplace(v);`		
`o.reset();`		
`o.swap(o2);`		
	`o == o2` and `o <=> o2`	
	`o == v` and `o <=> v`	
`*o` and `o->m`	`o.transform(f)`	`o.begin()` and `o.end()`
`o.has_value()` and `bool(o)`	`o.and_then(f)`	`o.size()`
`o.value_or(v)`	`o.or_else(f)`	`o.data()`
`std::hash<Opt>{}(o)`		

std::optional vs. std::maybe_view

`std::optional<V>`	both	Proposed `std::maybe_view<V>`
`Opt(nullopt)`	`Opt()`	
	`Opt(v)`	
	`Opt(in_place, v)`	
`o = nullopt;`	`o = v;`	
`o.emplace(v);`		
`o.reset();`		
`o.swap(o2);`		
	`o == o2` and `o <=> o2`	
	`o == v` and `o <=> v`	
`*o` and `o->m`	`o.transform(f)`	`o.begin()` and `o.end()`
`o.has_value()` and `bool(o)`	`o.and_then(f)`	`o.size()`
`o.value_or(v)`	`o.or_else(f)`	`o.data()`
`std::hash<Opt>{}(o)`		

std::optional vs. std::maybe_view

`std::optional<V>`	both	Proposed `std::maybe_view<V>`
`Opt(nullopt)`	`Opt()`	
	`Opt(v)`	
	`Opt(in_place, v)`	
`o = nullopt;`	`o = v;`	
`o.emplace(v);`		
`o.reset();`		
`o.swap(o2);`		
	`o == o2` and `o <=> o2`	
	`o == v` and `o <=> v`	
`*o` and `o->m`	`o.transform(f)`	`o.begin()` and `o.end()`
`o.has_value()` and `bool(o)`	`o.and_then(f)`	`o.size()`
`o.value_or(v)`	`o.or_else(f)`	`o.data()`
`std::hash<Opt>{}(o)`		

I’ve stopped writing papers. Now I only write anti-papers.

operator=

```
myclass : type = {
    name: std::string = "Henry";
    addr: std::string = "123 Ford Dr.";

    operator=: (out this, that) = {
        std::cout << "general operator=";
    }

    operator=: (out this, x: std::string) = {
        name = x;
        std::cout << "conversion - from string";
    }
}
```

C++ incoherence, complexity, and lack of direction

- Increasing over time
- Increases training cost
- Increases maintenance cost

We shouldn’t worry about C++’s longevity…

We should worry about the longevity of our C++ code

Please do

- Write simple idiomatic C++ code
- Use simple libraries
- Use other tools when warranted
- Advocate for less churn in the committee

Please don't

- Make this language more complex
- Veer from core principles like generic programming
- Try to make C++ into other languages

I want to give a message of hope

The Beman Project

- Starting this week at C++Now
- Goal: *Improve the quality of standard library proposals via. community engagement*
- Community infrastructure and tools so people can provide feedback on proposal implementations

C++ Should be C++

Let’s talk!


Adobe
Artwork by Dan Zucco