C++ Should Be C++

David Sankel
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.”

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.
- StackOverflow annual survey
- 90,000 people
18.7% of polled C++ engineers said Rust is another tool they use
The Rust Programming Language
A place for all things related to the Rust programming language—an open-source systems language that emphasizes performance, reliability, and productivity.

289K Members 595 Online Top 1%

C++
Discussions, articles and news about the C++ programming language or programming in C++.

279K Members 162 Online Top 1%

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

Saying yes
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. \( T \ a = b; \ \text{assert}(a==b); \)
2. \( T \ a; \ a = b; \ \Leftrightarrow \ T \ a = b; \)
3. \( T \ a = c; \ T \ b = c; \ a = d; \ \text{assert}(b==c) \)
4. \( T \ a = c; \ T \ b = c; \ \text{zap}(a); \ \text{assert}(b==c \ \&\& \ a!=b) \) 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
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...

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

**Mandates:** The following are all true:

- No type is present more than once in the pack `remove_cvref_t<Args...&& 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]).

**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 ≤ j < i`.
- Let `FArgs` a pack of the types `Args...[j]` for each `j` such that `i ≤ j ≤ sizeof...(args)`.
- Let `fargs` a pack of the expressions `args...[j]` for each `j` such that `i ≤ j ≤ sizeof...(args)`.
Wording details...

```cpp
template<
    class F,
    class... Args
>

explicit thread(F&& f, Args&&... args);
```

**Mandates:** The following are all true:

- No type is present more than once in the pack `remove_cvref_t<Attr>`. *[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]).

**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 ≤ j < i`.
- Let `FArgs` a pack of the types `Args...[j]` for each `j` such that `i ≤ j ≤ sizeof...(args)`.
- Let `fargs` a pack of the expressions `args...[j]` for each `j` such that `i ≤ j ≤ sizeof...(args)`. 
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<Args>. [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 $F$Args a pack of the types Args...[j] for each $j$ such that $i \leq j \leq \text{sizeof}...(args)$.
• Let $f$args a pack of the expressions args...[j] for each $j$ such that $i \leq j \leq \text{sizeof}...(args)$.
Wording details...

```cpp
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<Args...>`. [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 ≤ j < i`.
- Let `FArgs` a pack of the types `Args...[j]` for each `j` such that `i ≤ j ≤ sizeof...(args)`.
- Let `fargs` a pack of the expressions `args...[j]` for each `j` such that `i ≤ j ≤ sizeof...(args)`. 
template<
class F, class... Args>
explicit thread(F&& f, Args&&... args);

Mandates: The following are all true:

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

Constraints:

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

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 ≤ j < i.
• Let FArgs a pack of the types Args...[j] for each j such that i ≤ j ≤ sizeof...(args).
• Let fargs a pack of the expressions args...[j] for each j such that i ≤ j ≤ sizeof...(args).
std::jthread thr({ .name = "worker", .stack = 16384 }, [] { std::puts("running"); });

- 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

```cpp
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

```cpp
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

```cpp
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

```cpp
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.
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

```cpp
template <class S>
struct _retry_sender {

    template <class Env>
    auto get_completion_signal(Env env) -> /*blah*/;

    template <class Env, class R>
    auto complete(Env env, 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.
## `std::optional` vs. `std::maybe_view`

<table>
<thead>
<tr>
<th><code>std::optional&lt;V&gt;</code></th>
<th><code>both</code></th>
<th><code>Proposed std::maybe_view&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Opt(nullopt)</code></td>
<td><code>Opt()</code></td>
<td></td>
</tr>
<tr>
<td><code>Opt(v)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Opt(in_place, v)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>o = nullopt;</code></td>
<td><code>o = v;</code></td>
<td></td>
</tr>
<tr>
<td><code>oemplace(v);</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>o.reset();</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>o.swap(o2);</code></td>
<td><code>o == o2 and o &lt;=&gt; o2</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>o == v and o &lt;=&gt; v</code></td>
<td></td>
</tr>
<tr>
<td><code>*o and o-&gt;m</code></td>
<td><code>o.transform(f)</code></td>
<td><code>o.begin() and o.end()</code></td>
</tr>
<tr>
<td><code>o.has_value() and bool(o)</code></td>
<td><code>o.and_then(f)</code></td>
<td><code>o.size()</code></td>
</tr>
<tr>
<td><code>o.value_or(v)</code></td>
<td><code>o.or_else(f)</code></td>
<td><code>o.data()</code></td>
</tr>
<tr>
<td><code>std::hash&lt;Opt&gt;{{}(o)</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## std::optional vs. std::maybe_view

<table>
<thead>
<tr>
<th>std::optional&lt;V&gt;</th>
<th>both</th>
<th>Proposed std::maybe_view&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opt(nullopt)</td>
<td>Opt()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Opt(v)</td>
<td></td>
</tr>
<tr>
<td>o = nullopt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>oemplace(v);</td>
<td>o = v;</td>
<td></td>
</tr>
<tr>
<td>o.reset();</td>
<td>o == o2 and o &lt;=&gt; o2</td>
<td></td>
</tr>
<tr>
<td>o.swap(o2);</td>
<td>o == v and o &lt;=&gt; v</td>
<td></td>
</tr>
<tr>
<td>*o and o-&gt;m</td>
<td>o.transform(f)</td>
<td>o.begin() and o.end()</td>
</tr>
<tr>
<td>o.has_value() and bool(o)</td>
<td>o.and_then(f)</td>
<td>o.size()</td>
</tr>
<tr>
<td>o.value_or(v)</td>
<td>o.or_else(f)</td>
<td>o.data()</td>
</tr>
<tr>
<td>std::hash&lt;Opt&gt;{{}(o)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
std::optional vs. std::maybe_view

<table>
<thead>
<tr>
<th>std::optional&lt;V&gt;</th>
<th>both</th>
<th>Proposed std::maybe_view&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opt(nullopt)</td>
<td>Opt()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Opt(v)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Opt(in_place, v)</td>
<td></td>
</tr>
<tr>
<td>o = nullopt;</td>
<td>o = v;</td>
<td></td>
</tr>
<tr>
<td>oemplace(v);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o.reset();</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o.swap(o2);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o == o2 and o &lt;=&gt; o2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o == v and o &lt;=&gt; v</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*o and o-&gt;m</td>
<td>o.transform(f)</td>
<td>o.begin() and o.end()</td>
</tr>
<tr>
<td>o.has_value() and bool(o)</td>
<td>o.and_then(f)</td>
<td>o.size()</td>
</tr>
<tr>
<td>o.value_or(v)</td>
<td>o.or_else(f)</td>
<td>o.data()</td>
</tr>
<tr>
<td>std::hash&lt;Opt&gt;{v}(o)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### std::optional vs. std::maybe_view

| std::optional<
> | both | Proposed std::maybe_view<|  |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Opt(nullopt)</td>
<td>Opt()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opt(v)</td>
<td>Opt(v)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opt(in_place, v)</td>
<td>Opt(in_place, v)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o = nullopt;</td>
<td>o = v;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o.emplace(v);</td>
<td>o.emplace(v);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o.reset();</td>
<td>o.reset();</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o.swap(o2);</td>
<td>o.swap(o2);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o == o2 and o == o2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o == v and o == v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*o and o-&gt;m</td>
<td>o.transform(f)</td>
<td>o.begin() and o.end()</td>
<td></td>
</tr>
<tr>
<td>o.has_value() and bool(o)</td>
<td>o.and_then(f)</td>
<td>o.size()</td>
<td></td>
</tr>
<tr>
<td>o.value_or(v)</td>
<td>o.or_else(f)</td>
<td>o.data()</td>
<td></td>
</tr>
<tr>
<td>std::hash&lt;Opt&gt;{(</td>
<td>o)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### std::optional vs. std::maybe_view

<table>
<thead>
<tr>
<th>std::optional&lt;V&gt;</th>
<th>both</th>
<th>Proposed std::maybe_view&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opt(nullptr)</td>
<td>Opt()</td>
<td></td>
</tr>
<tr>
<td>Opt(v)</td>
<td>Opt(v)</td>
<td></td>
</tr>
<tr>
<td>Opt(in_place, v)</td>
<td>Opt(in_place, v)</td>
<td></td>
</tr>
<tr>
<td>o = nullptr;</td>
<td>o = v;</td>
<td></td>
</tr>
<tr>
<td>o.emplace(v);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o.reset();</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o.swap(o2);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o == o2 and o == o2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o == v and o == v</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o and o-&gt;m</td>
<td>o.transform(f)</td>
<td>o.begin() and o.end()</td>
</tr>
<tr>
<td>o.has_value() and bool(o)</td>
<td>o.and_then(f)</td>
<td>o.size()</td>
</tr>
<tr>
<td>o.value_or(v)</td>
<td>o.or_else(f)</td>
<td>o.data()</td>
</tr>
<tr>
<td>std::hash&lt;Opt&gt;{}(o)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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!