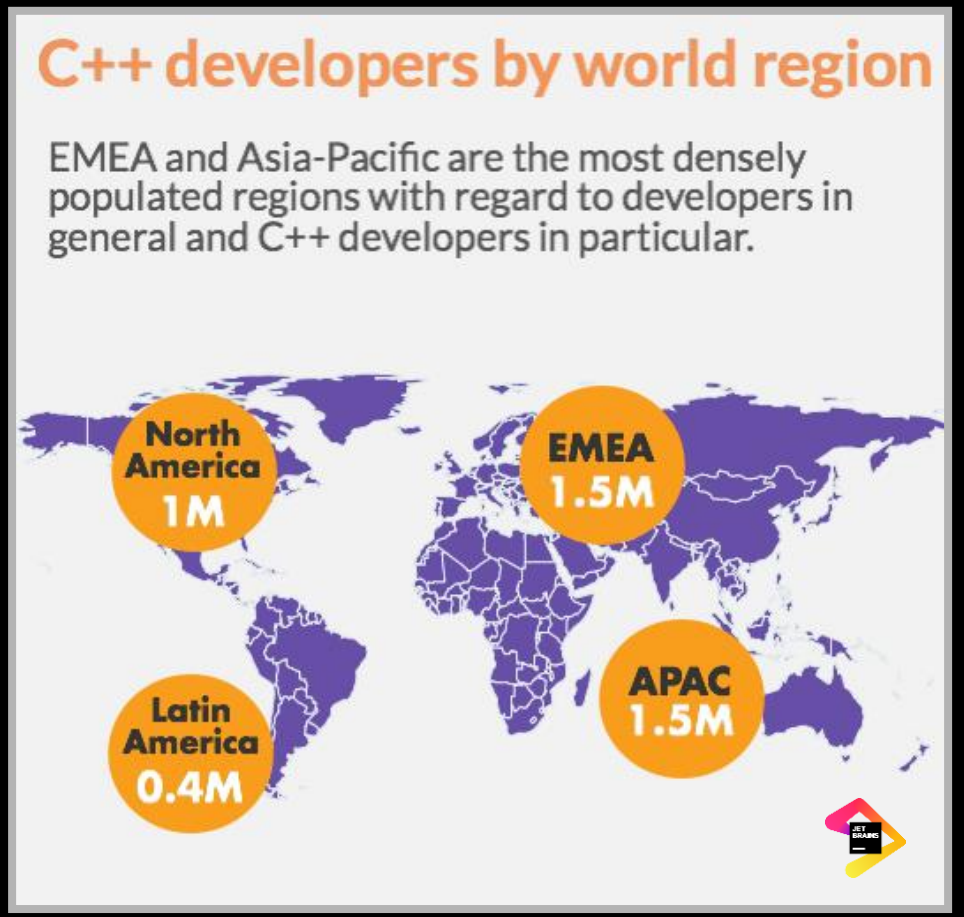
WHAT BELONGS IN THE C++ STANDARD LIBRARY?

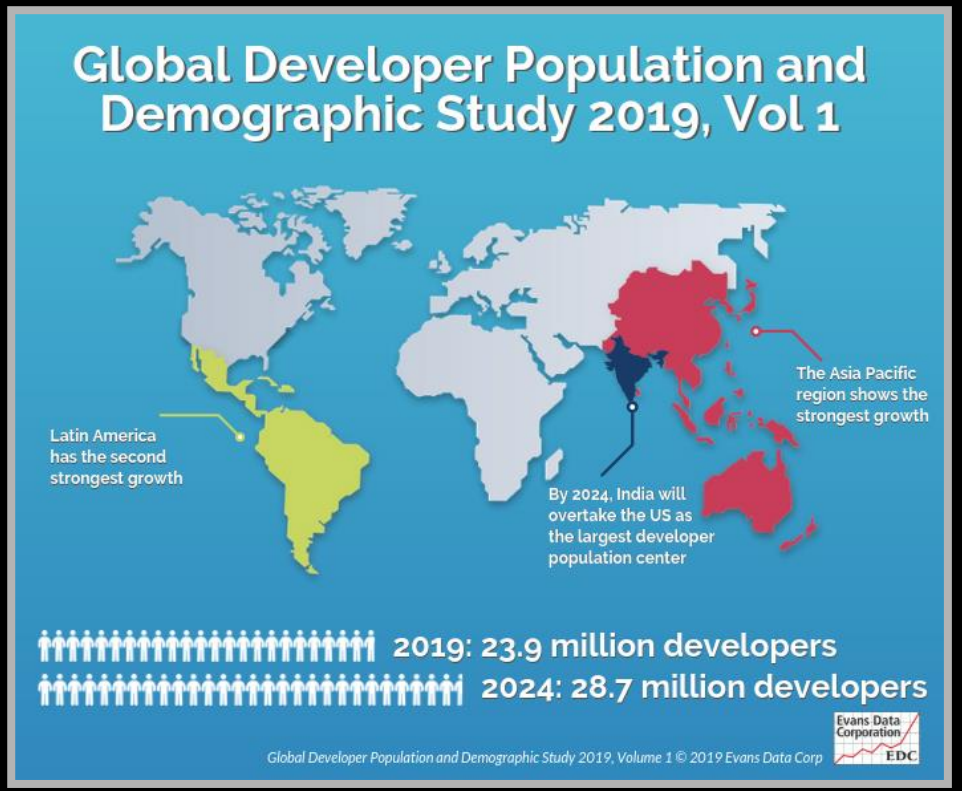
Bryce Adelstein Lelbach
@blelbach

NVIDIA. HPC Programming Models Architect
Standard C++ Library Evolution Chair, US Programming Languages Chair

{C++;} CppIndia

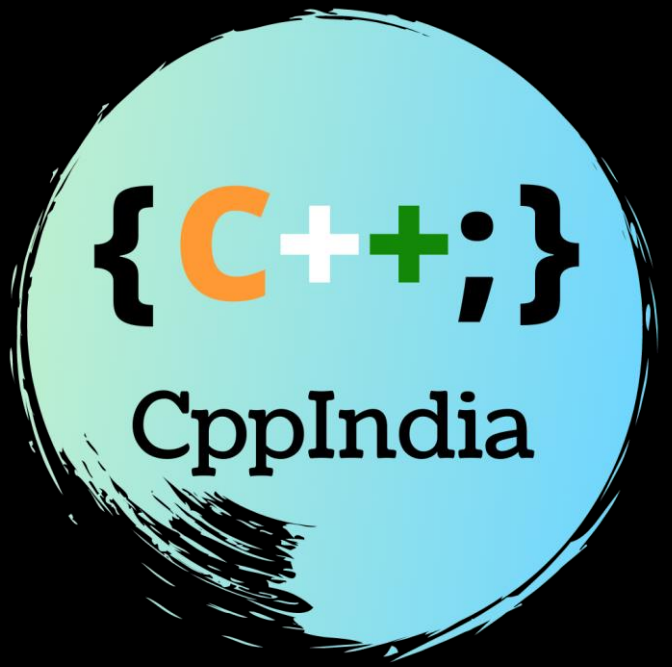
India’s Potential In The C++ Community



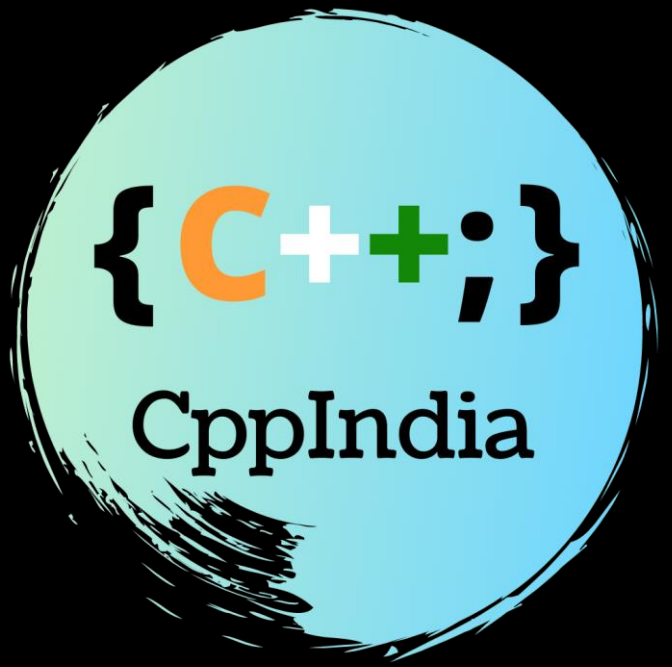


#include <C++>

India’s Potential In The C++ Community



India’s Potential In The C++ Community



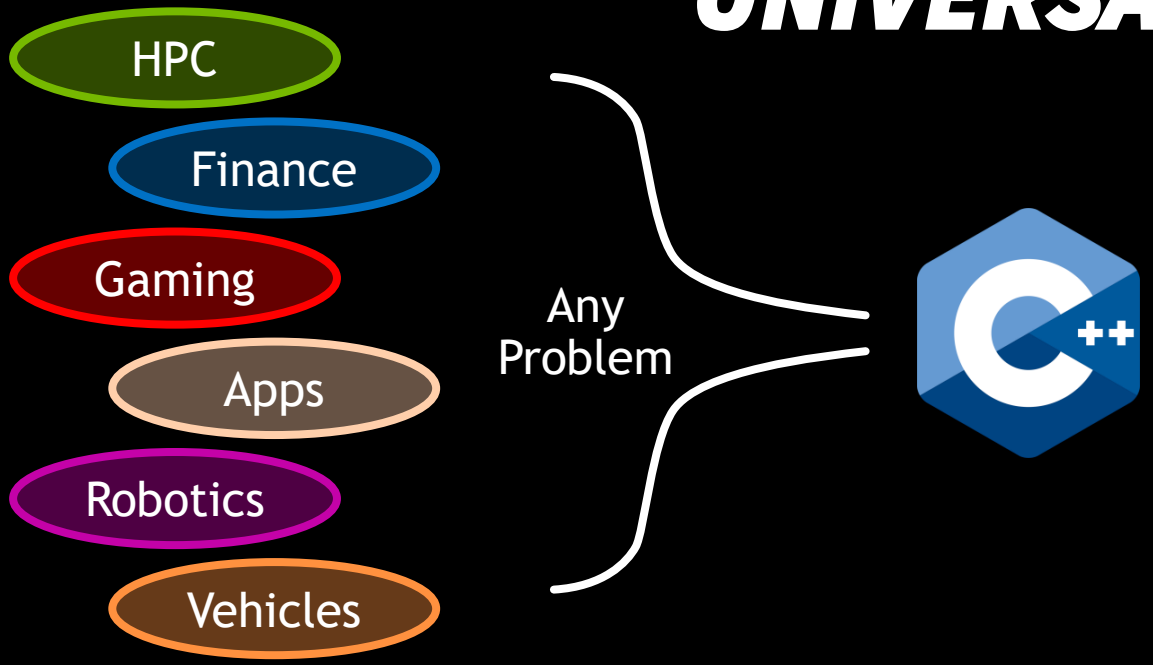
What Has Made C++ Successful?

What Has Made C++ Successful?

UNIVERSALITY

What Has Made C++ Successful?

UNIVERSALITY



What Has Made C++ Successful?

UNIVERSALITY

HPC

Finance

Gaming

Apps

Robotics

Vehicles

Any Problem

Any Paradigm

Imperative

Object Oriented

Functional

Generic

Parallel

Reactive

What Has Made C++ Successful?

UNIVERSALITY

HPC

Finance

Gaming

Apps

Robotics

Vehicles

Any Problem

Any Paradigm

Imperative

Object Oriented

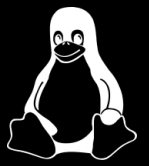
Functional

Generic

Parallel

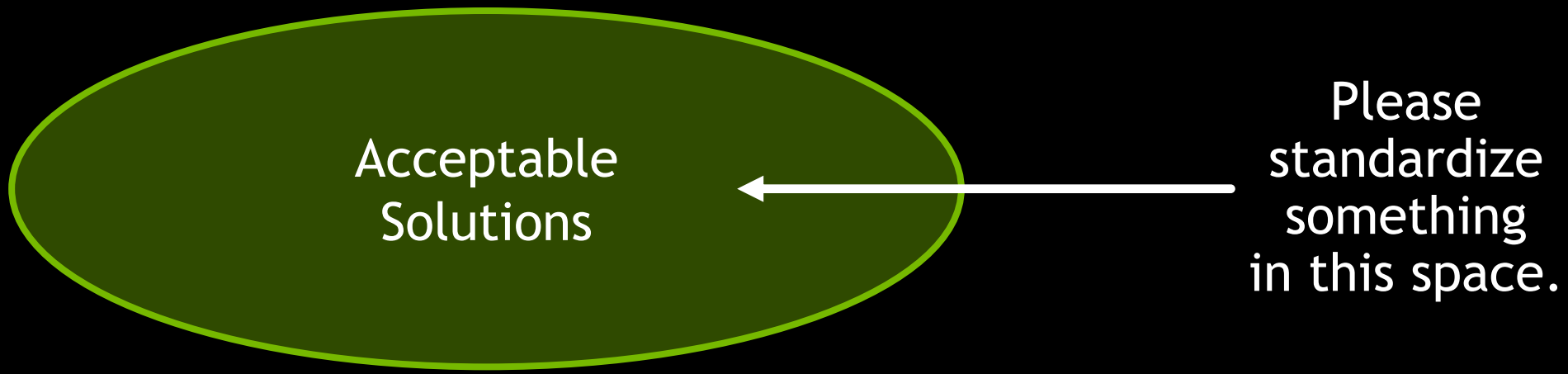
Reactive

Any Platform

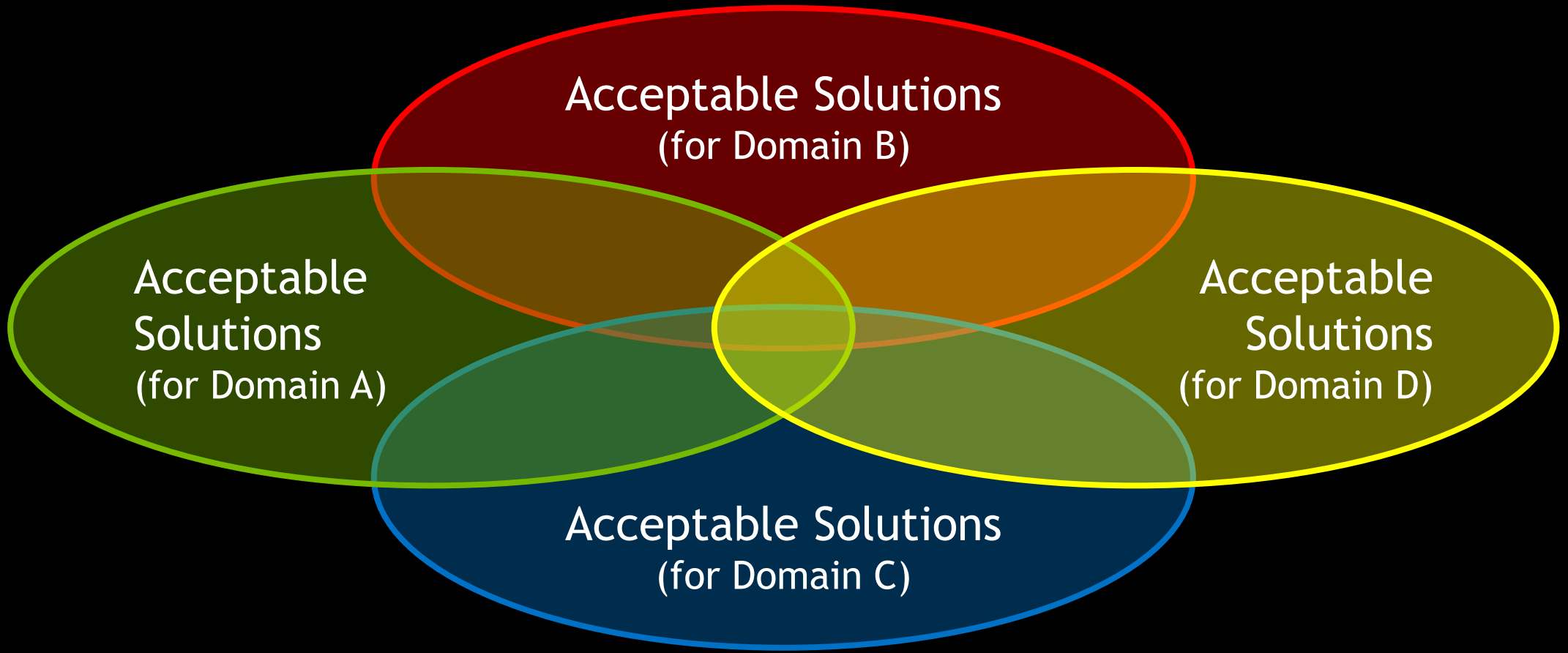


#include <C++>

Our Individual Perception



Reality

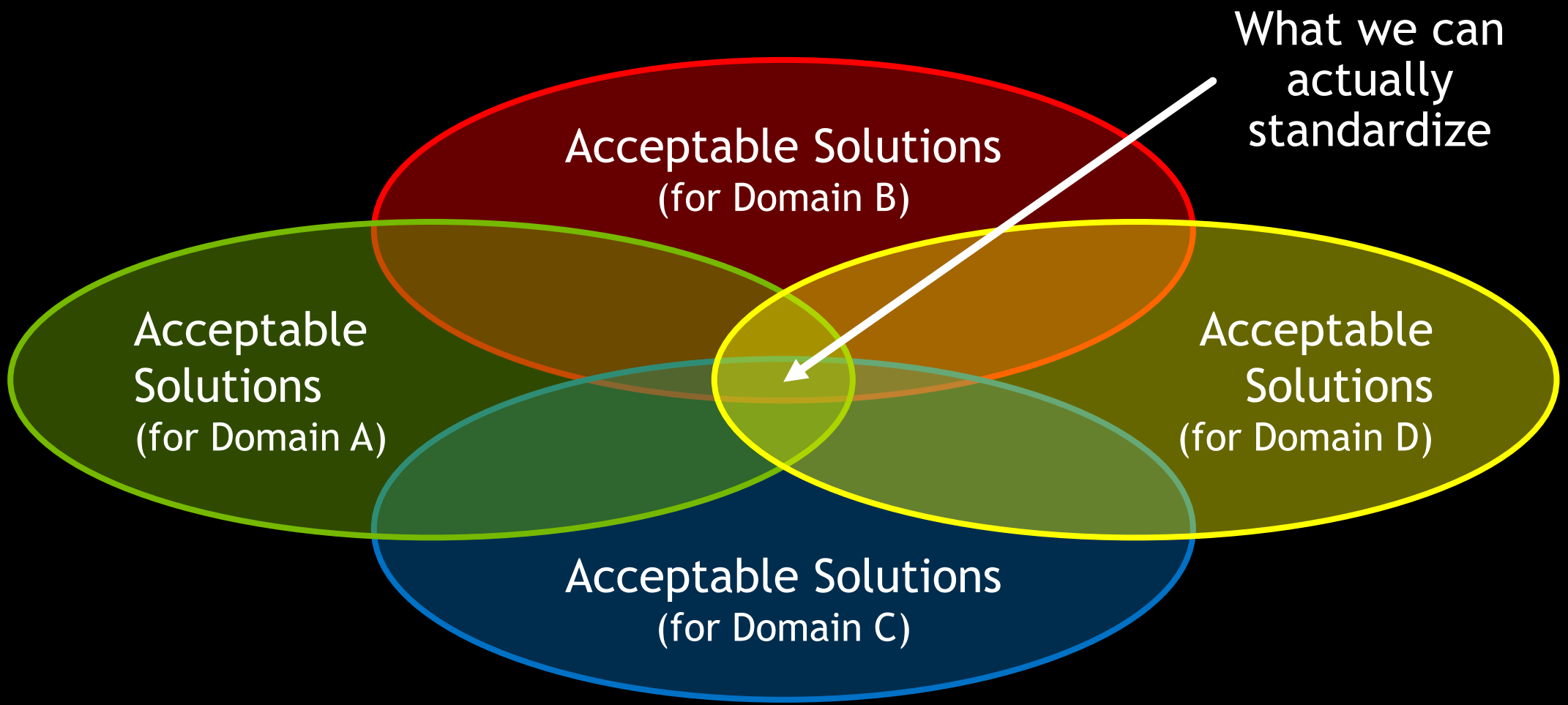


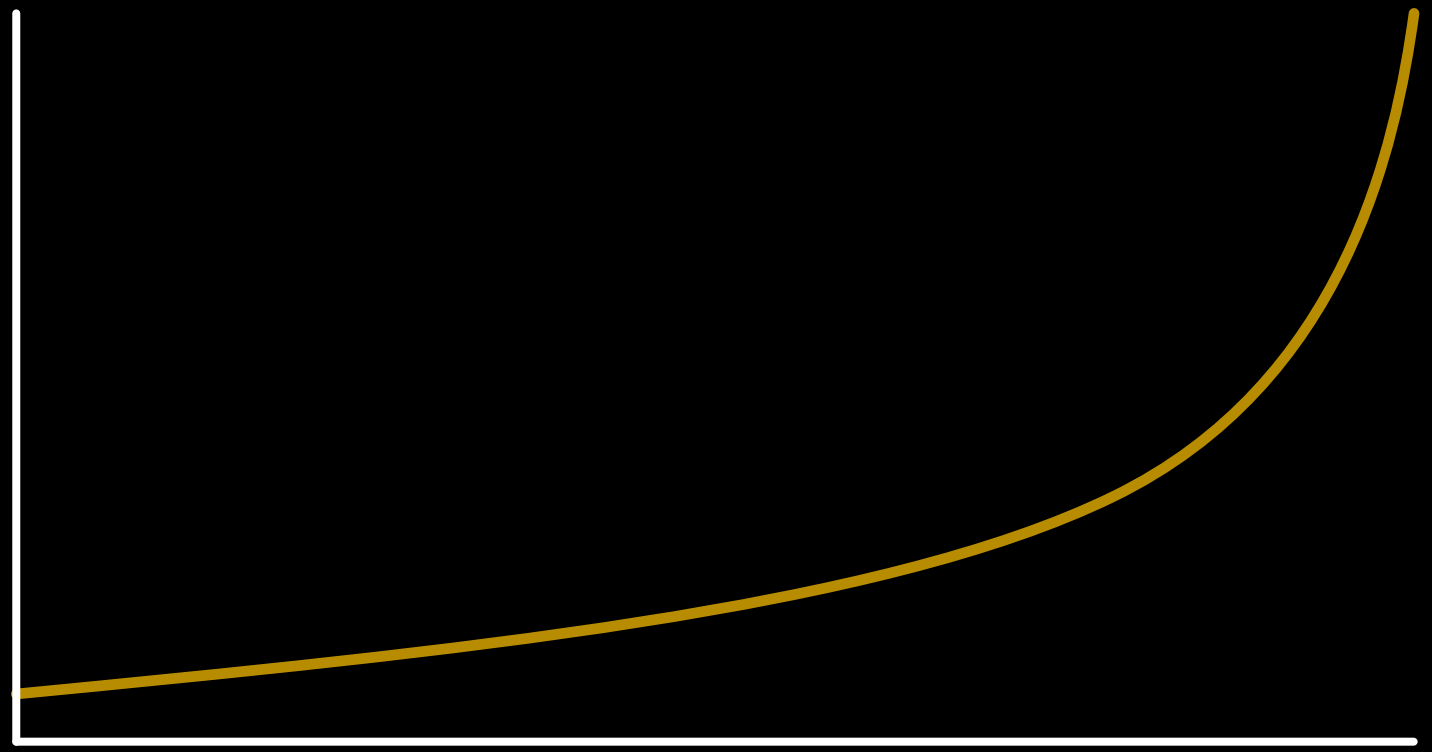
Use Case Sympathy (noun)

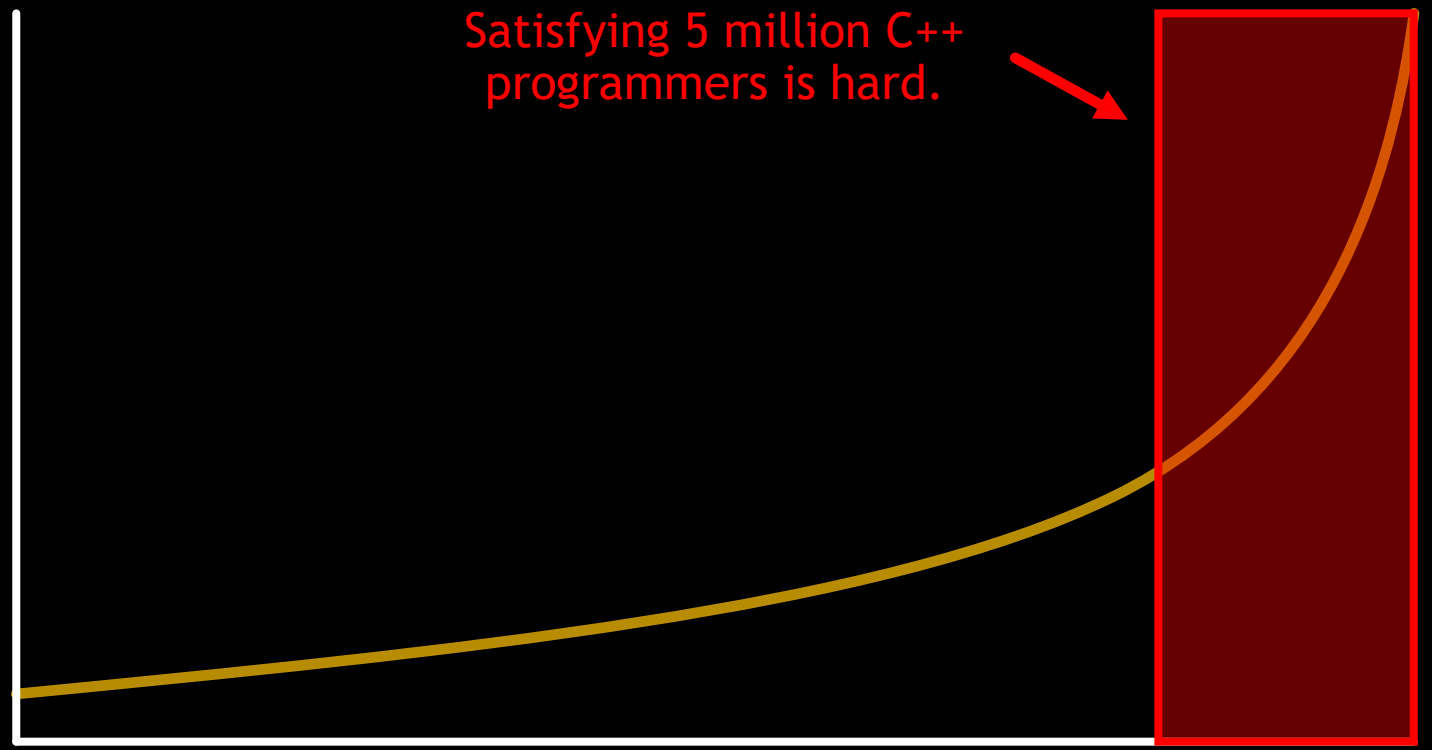
use case sym·pa·thy | yüz kās sim-pə-thē

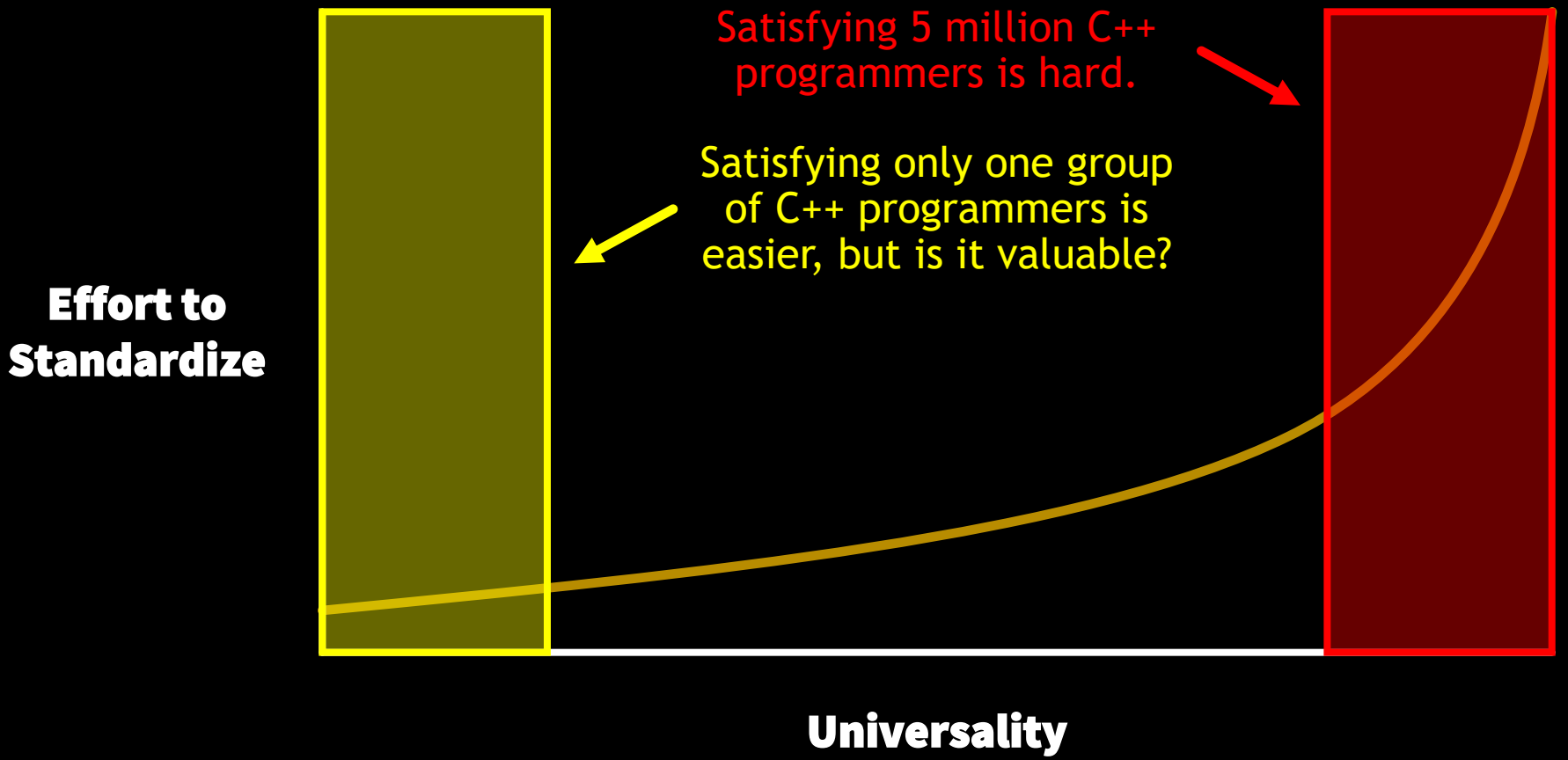
Accepting the importance and validity of use cases that you are not personally familiar with or believe in.

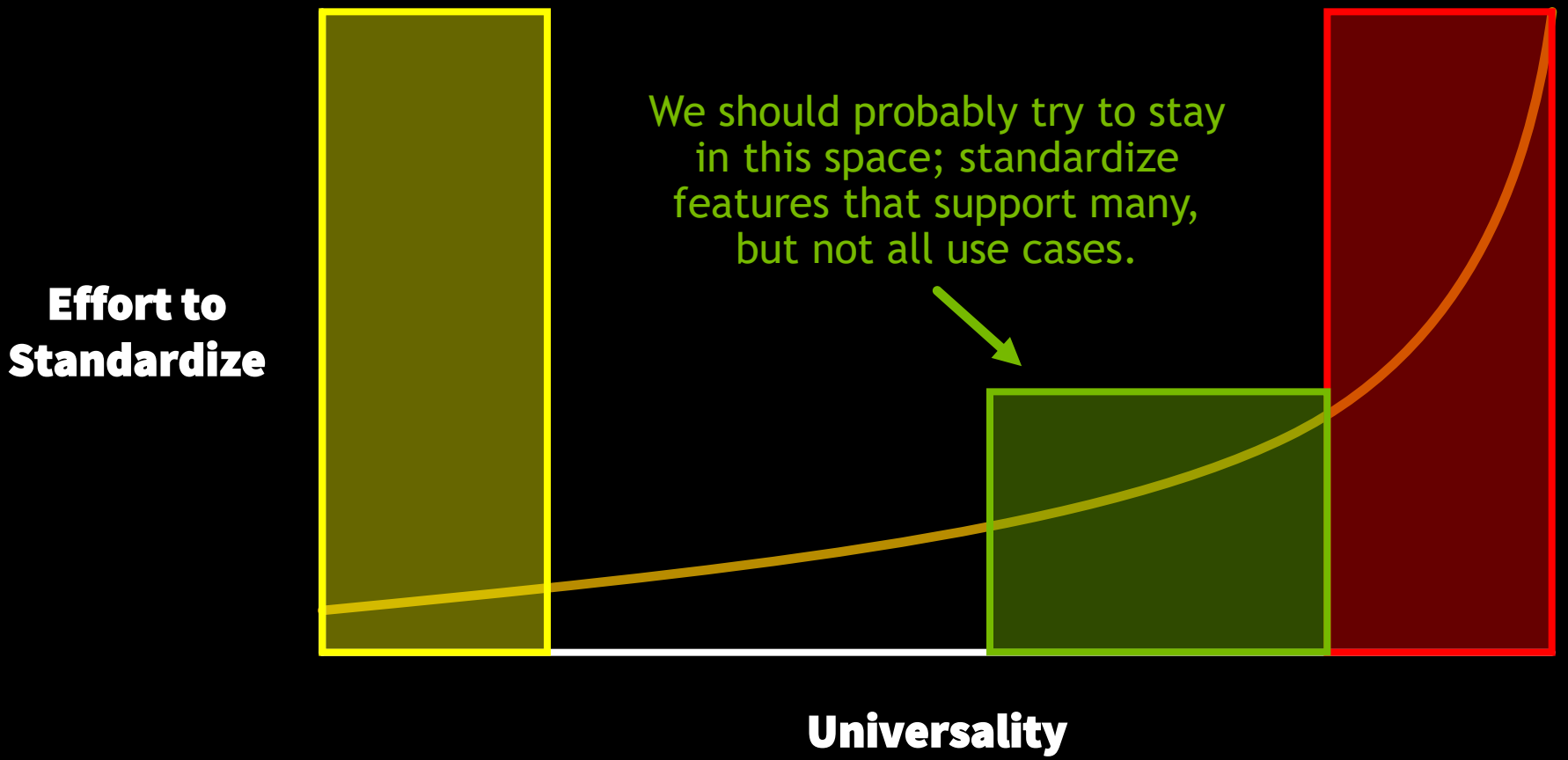
Reality



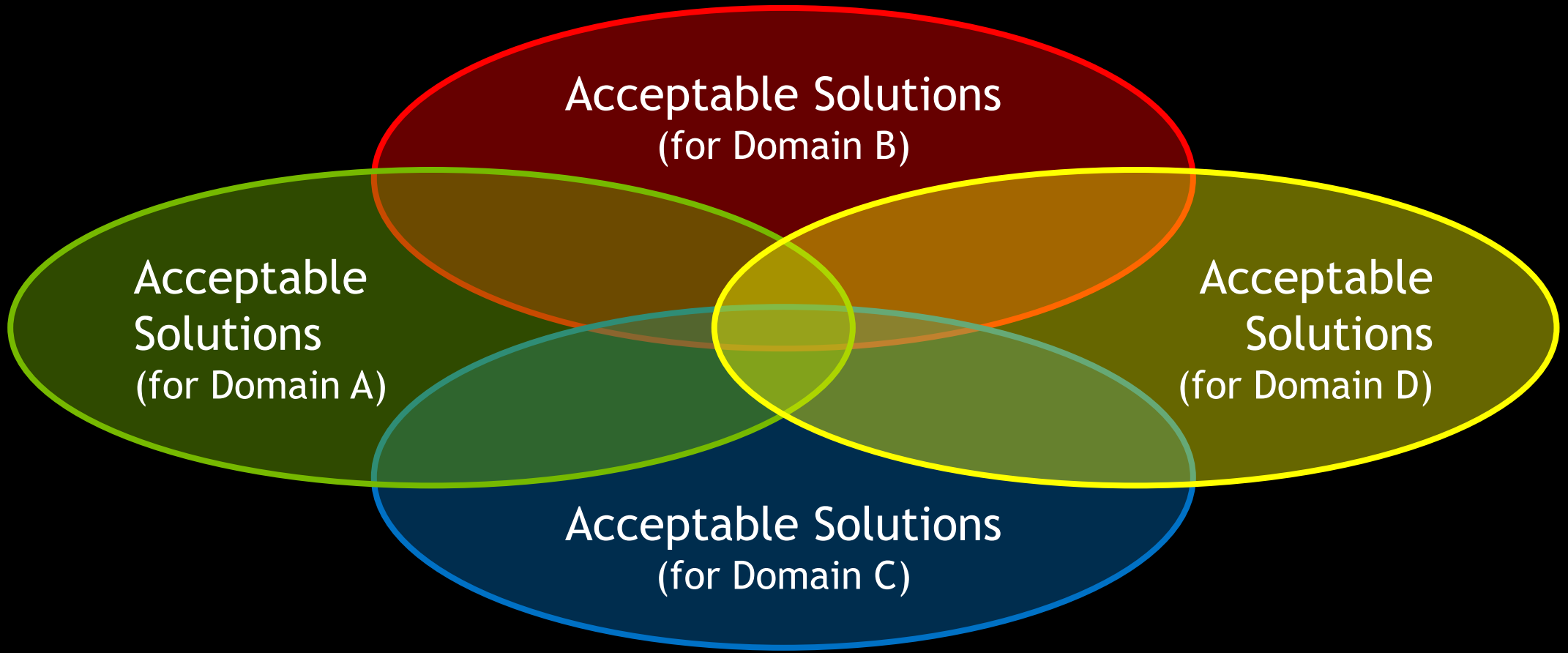








Universality Is A Double Edged Sword



Universality Is A Double Edged Sword

HPC

Finance

Gaming

Apps

Robotics

Vehicles

Any Problem

Any Paradigm

Imperative

Object Oriented

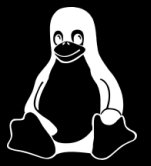
Functional

Generic

Parallel

Reactive

Any Platform

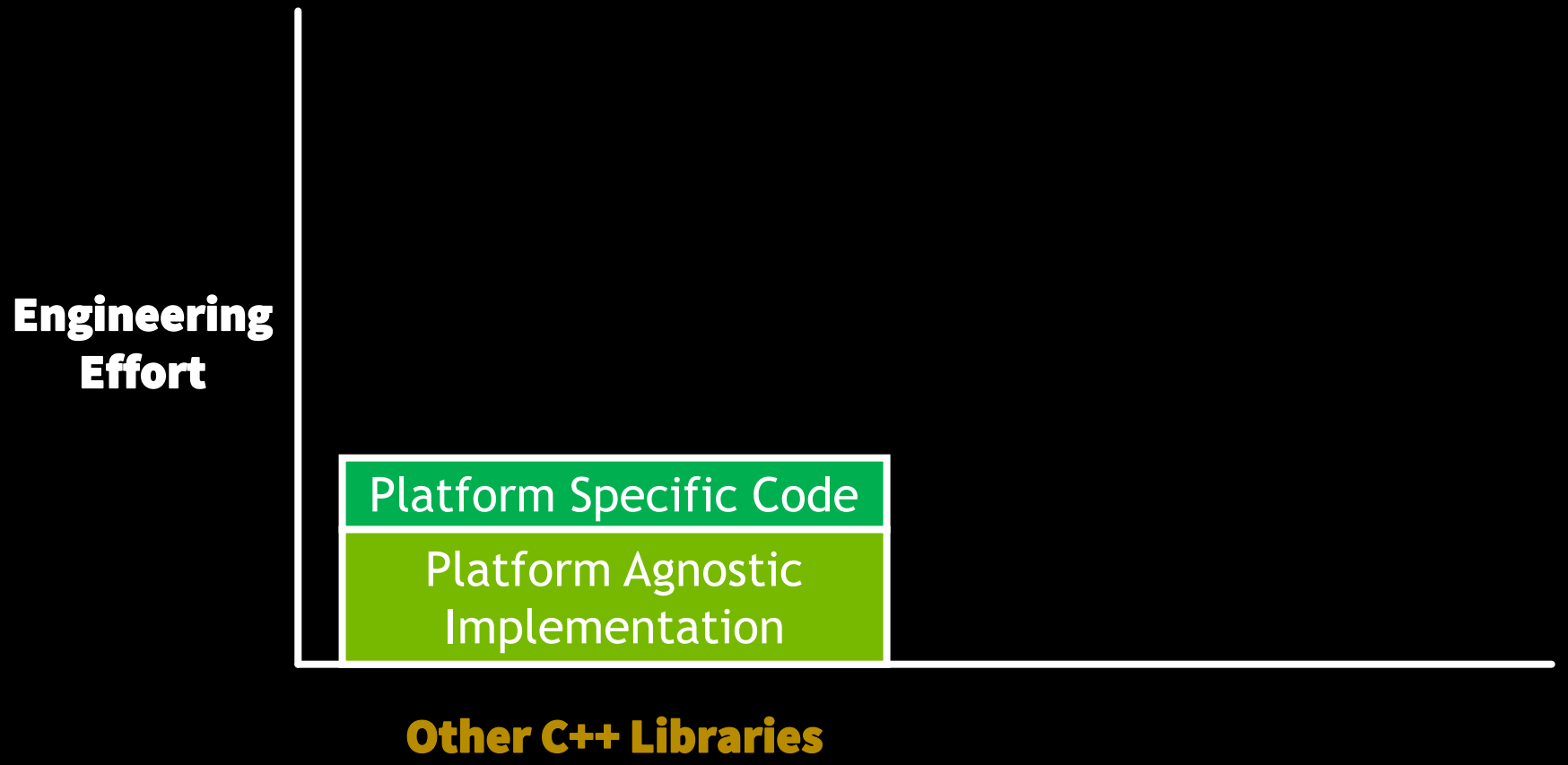


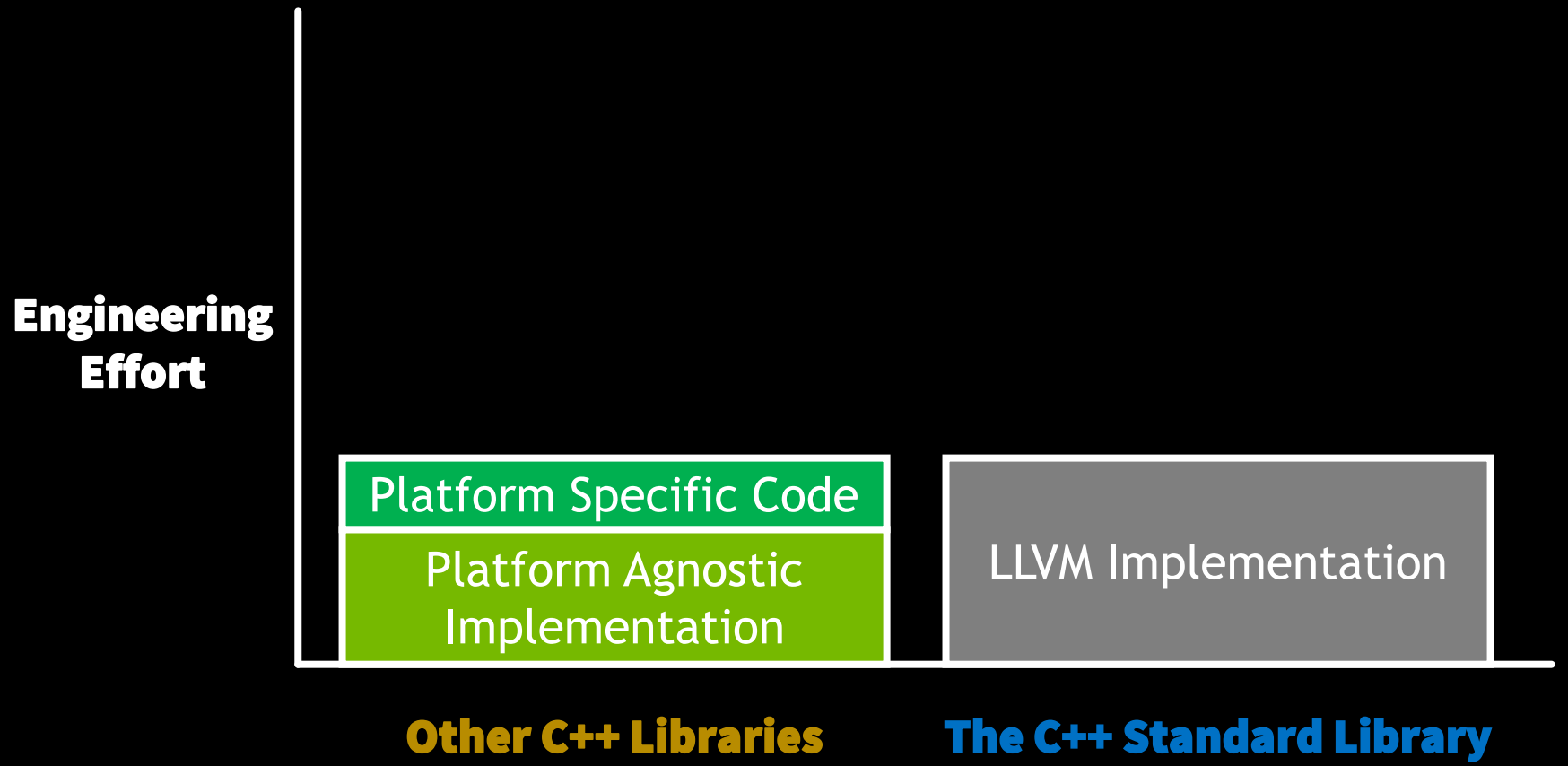
#include <C++>

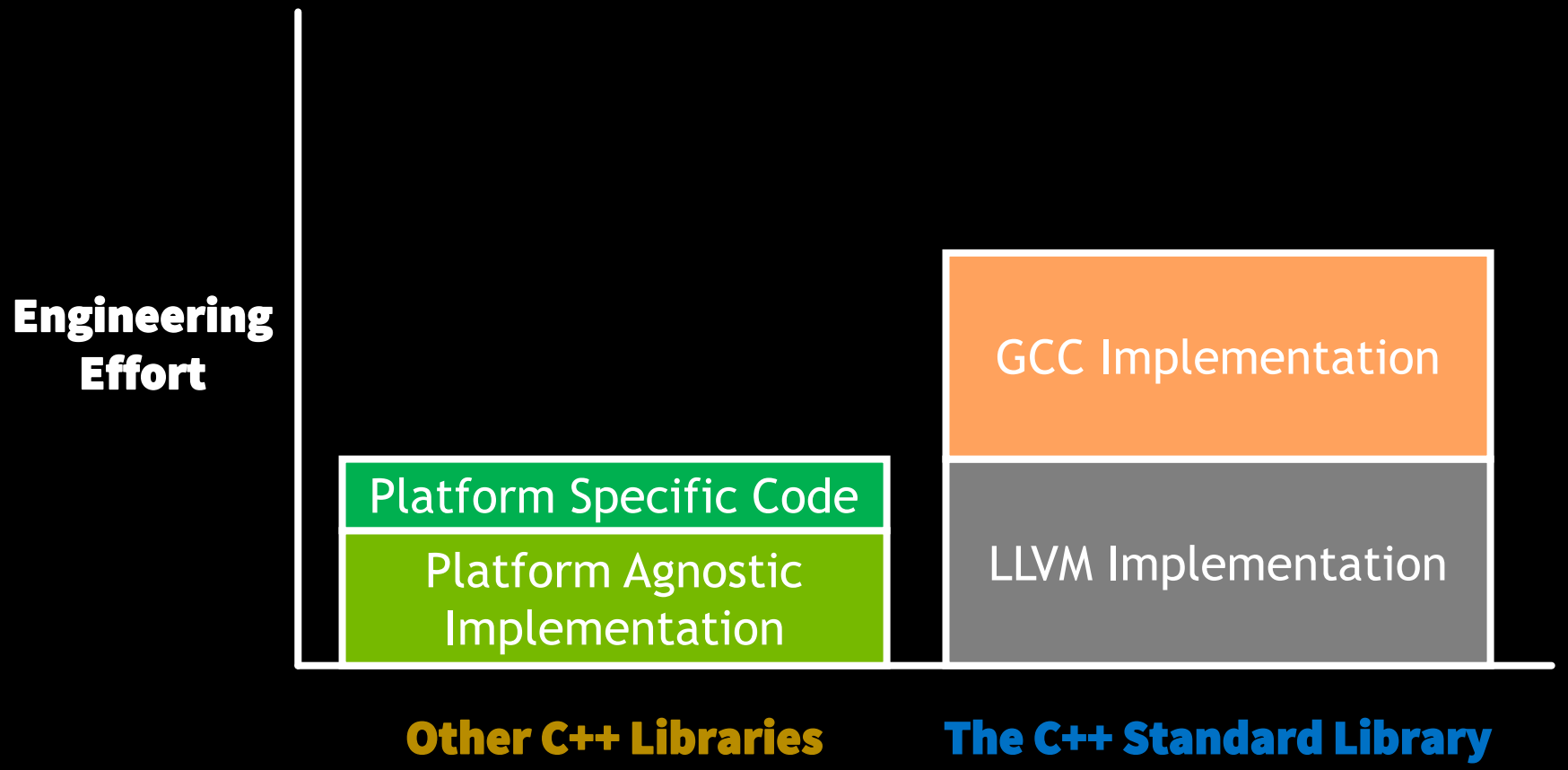
The C++ Standard Library is a *specification*.

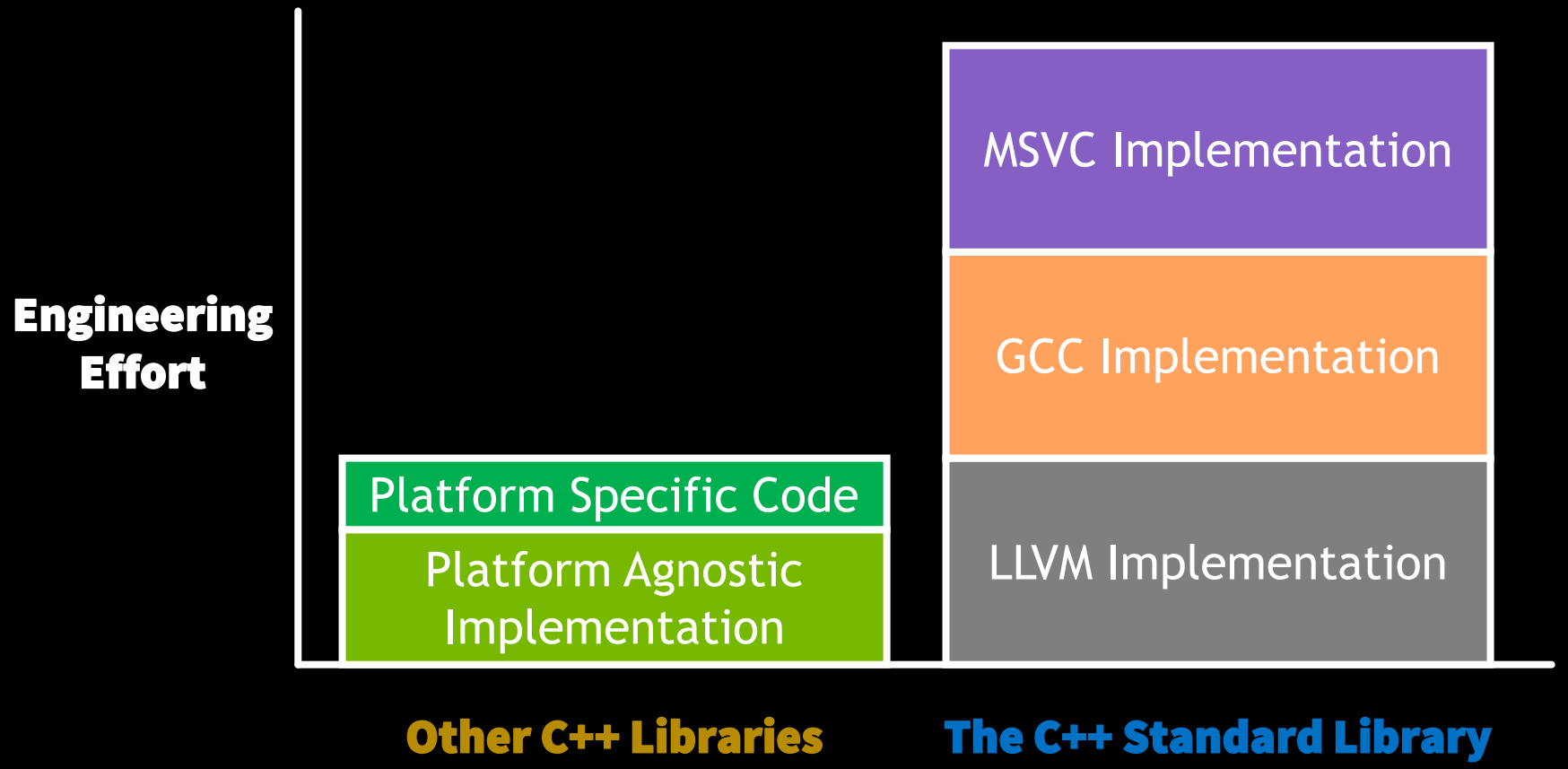
The C++ Standard Library is a *specification*.

That makes it an inefficient vehicle for delivering features.









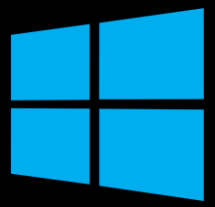
The C++ Standard is
descriptive, not ***prescriptive***.

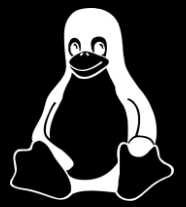
Domain A

Domain B

Domain A

The Standard specifies enough to be portable and consistent across platforms.





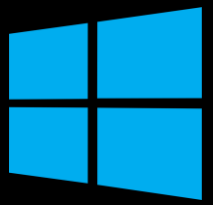
Domain A

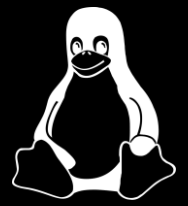
Domain B

Domain A

The Standard specifies enough to be portable and consistent across platforms.

The Standard grants enough freedom for each platform to choose the right design.





Implementation Freedom

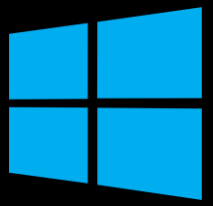
Domain A

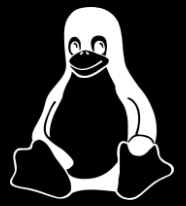
Domain B

Domain A

The Standard specifies enough to be portable and consistent across platforms.

The Standard grants enough freedom for each platform to choose the right design.



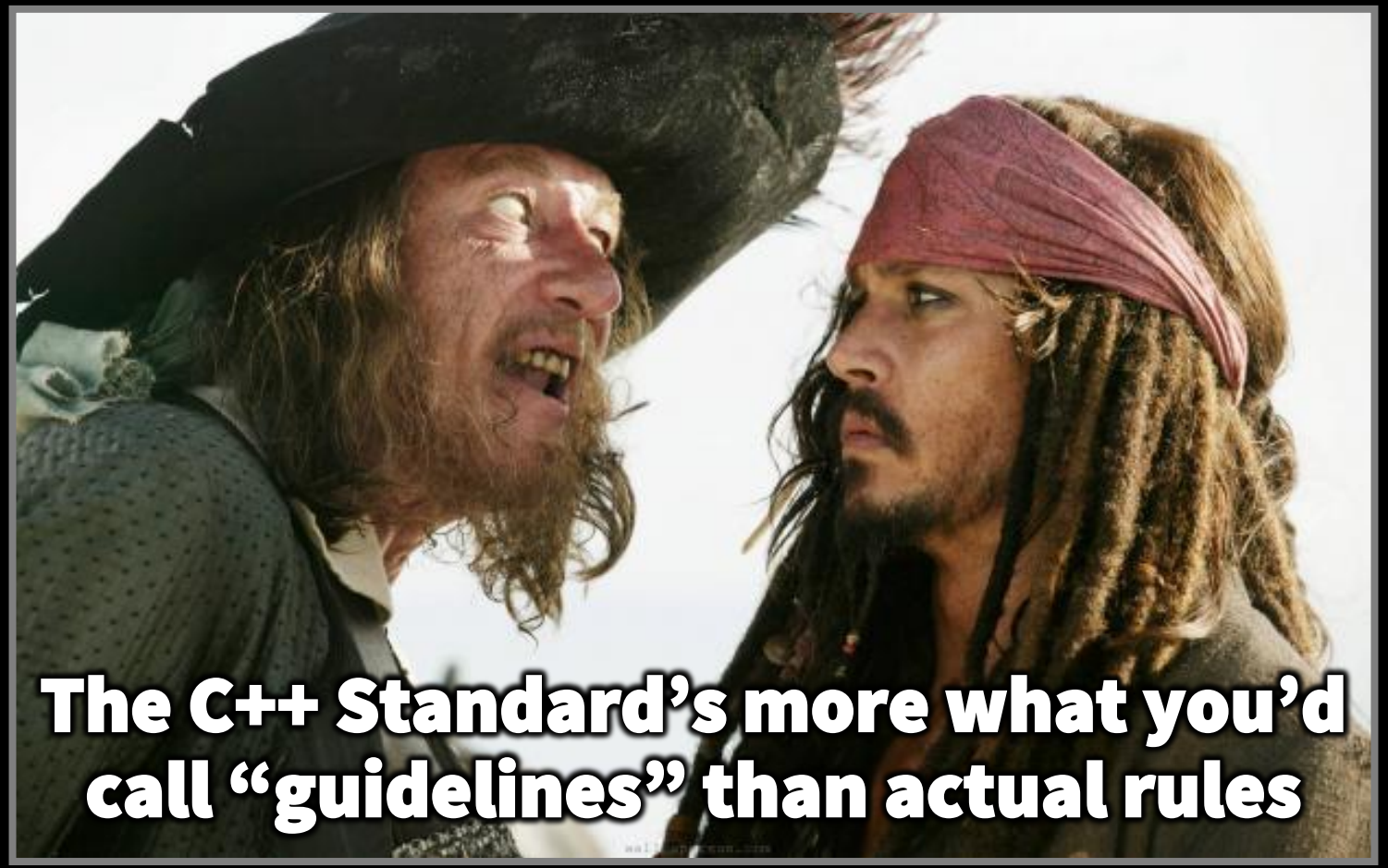


Implementation-defined and undefined behavior are often a feature, not a bug.

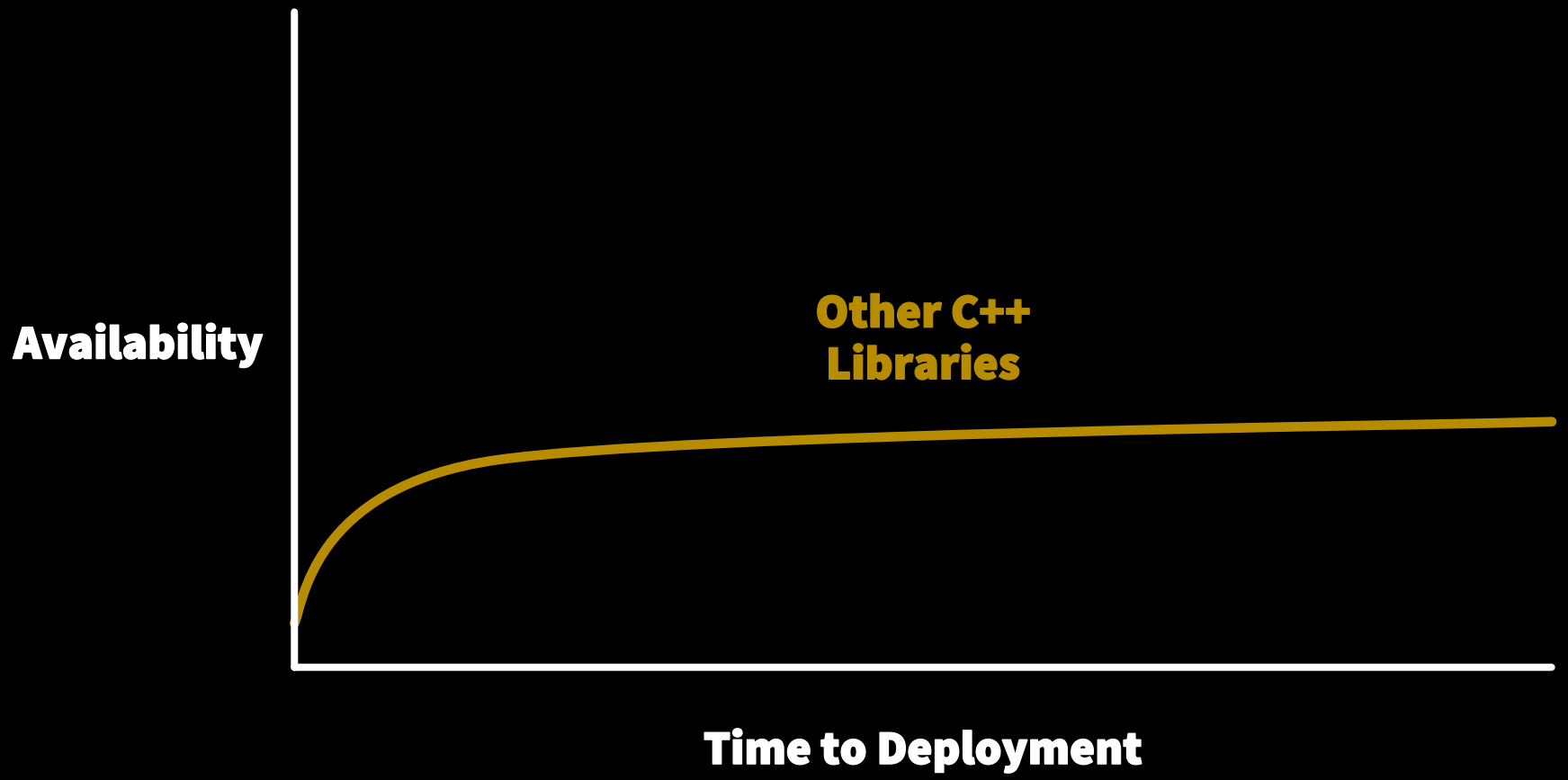
std::mutex

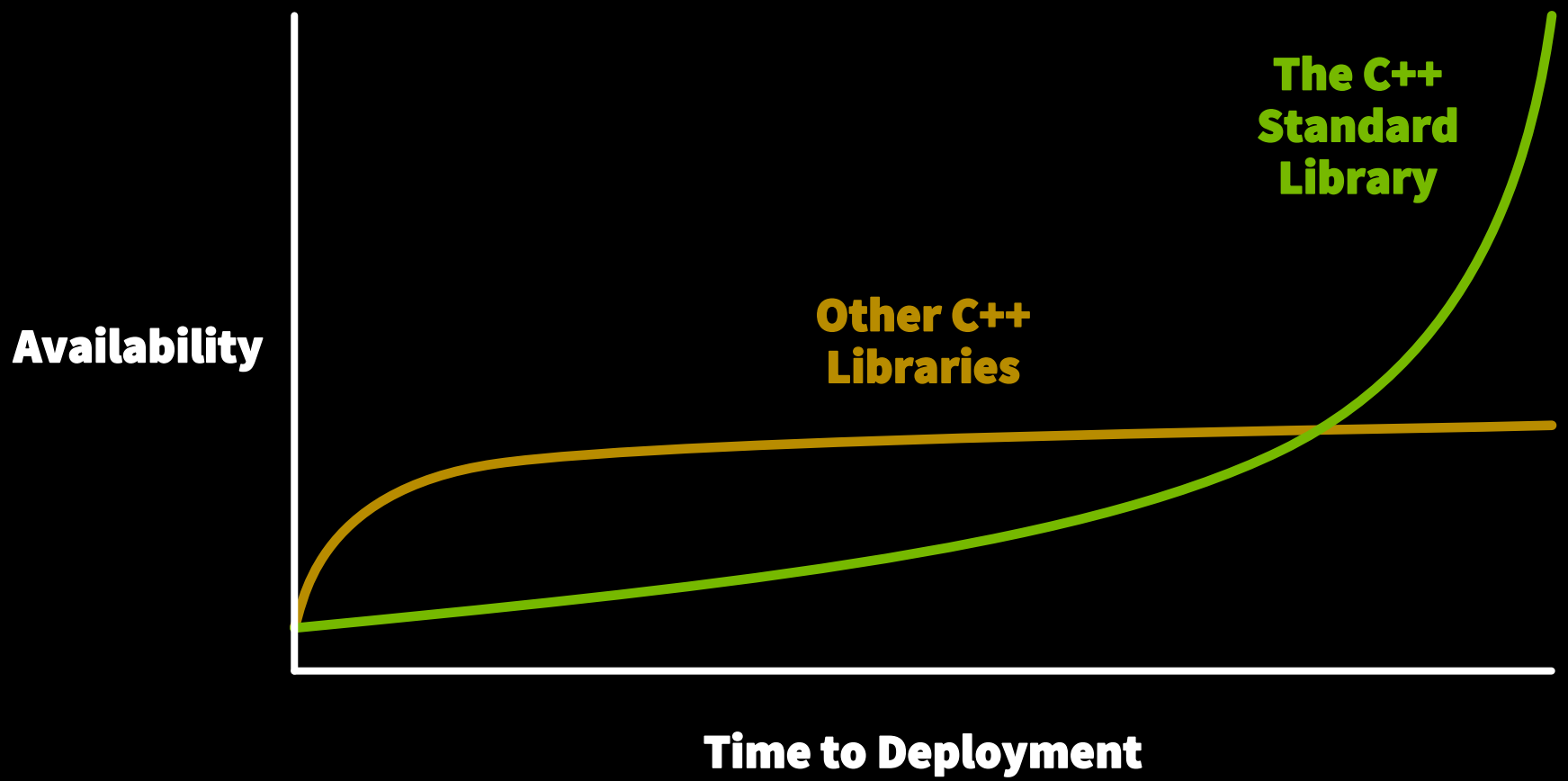
Implementation	Supported On	Pros	Cons
OS kernel mutexes	Older and newer operating systems.	Fair. Good perf under contention.	Higher latency.
Futexes	Newer operating systems.	Fair. Lower latency.	
Spinlocks	Bare metal.	Much lower latency. Never yields. Doesn't need an OS.	Unfair. Less energy efficient.
No synchronization	Single core platforms.	No overhead. Doesn't need an OS.	

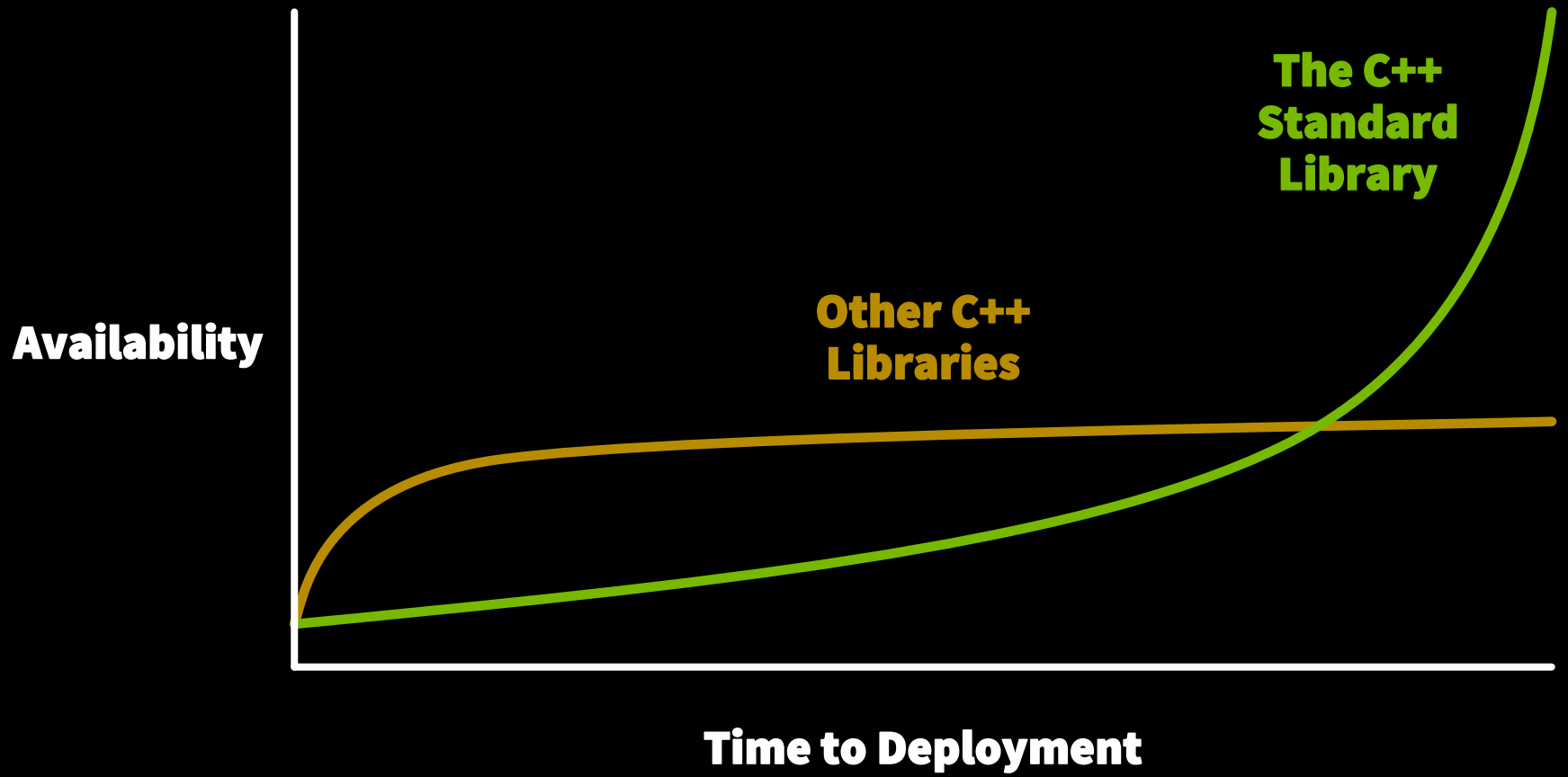
Implementation Freedom



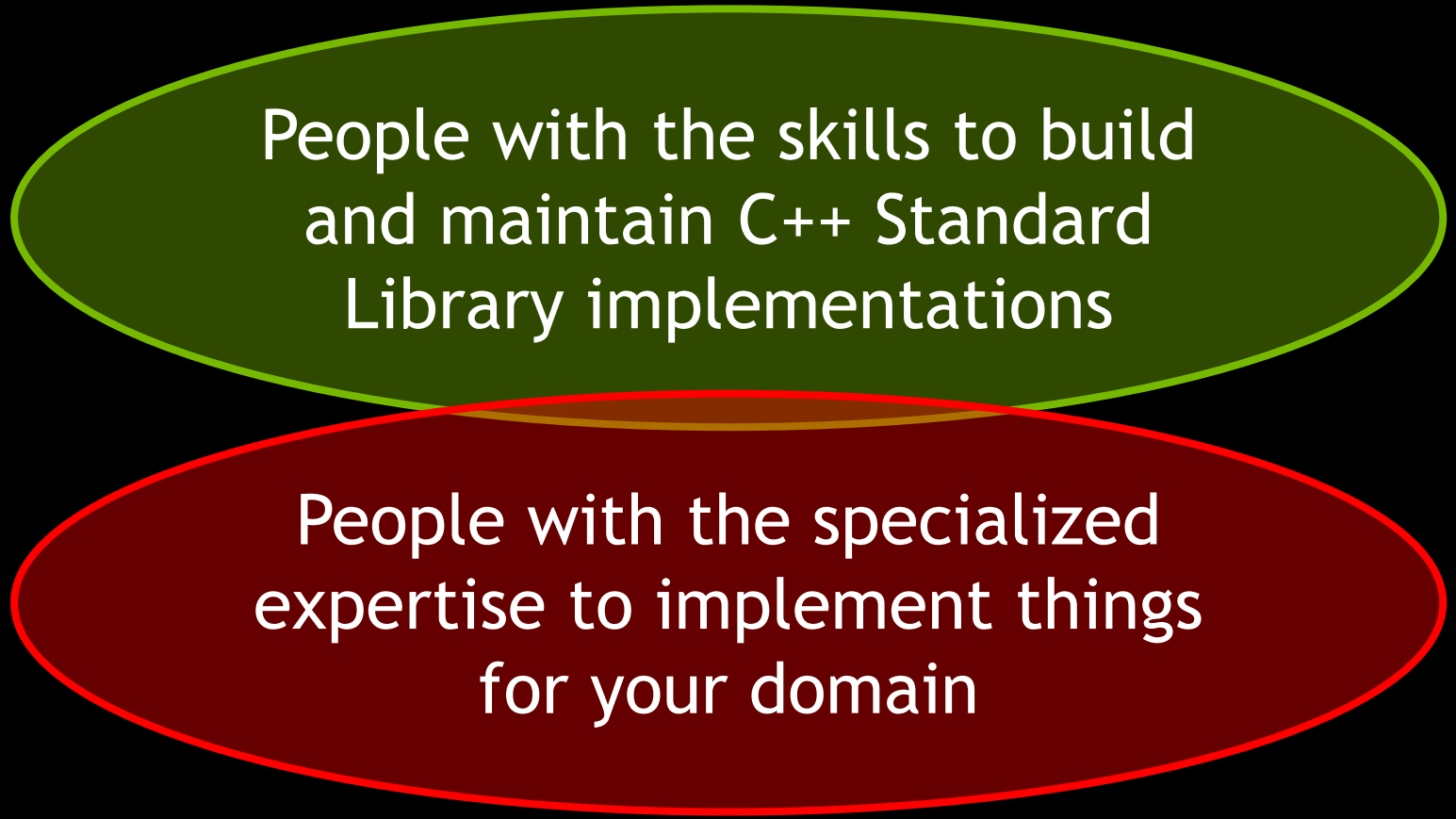
Standardization takes time.



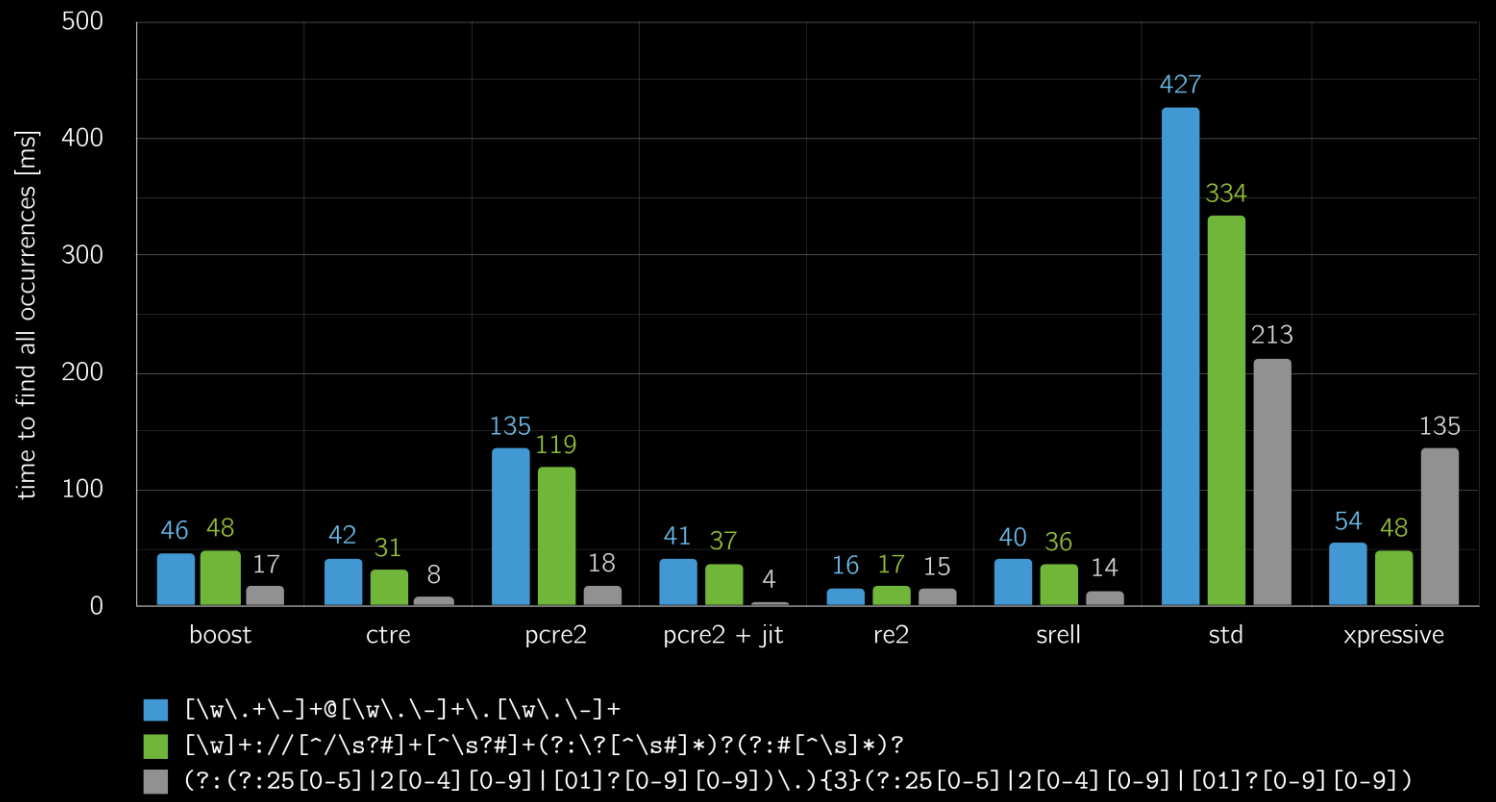




`std::` Implementors Aren't Domain Experts



std::regex Performance



Source: github.com/mariomka/regex-benchmark; Hana Dusíková

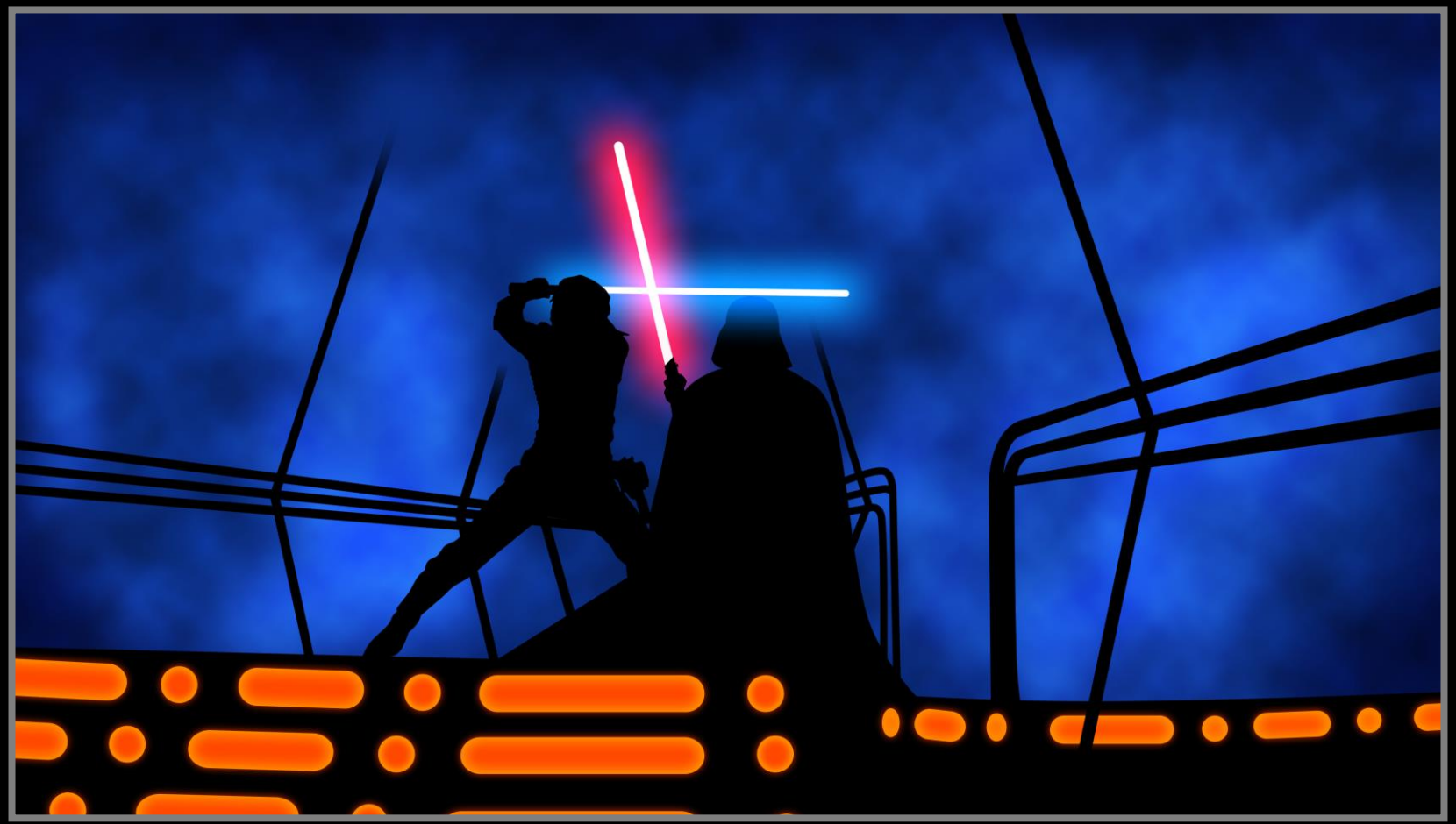
Implementers are experts at:

- Their specific platform.
- Balancing tradeoffs.
- Handling corner cases.

Implementers are less good at:

- Domain specific work:
  - Math special functions
  - `std::regex`
  - `<charconv>`
  - …

Stability vs Velocity



API: Syntax & semantics.

Source code. In the C++ Standard.

ABI: Binary representation & conventions.

Compiled code. Platform specific.

C++ Language ABI:

Binary representation & conventions for language facilities.

- Function calling conventions.
- Name mangling.
- Layout and size of types.
- Layout of virtual tables.
- Exception handling.
- Floating point mathematics.
- …

C++ Standard Library ABI:

Binary representation & conventions for C++ Standard Library facilities.

C++ Language ABI:

Binary representation & conventions for language facilities.

- Function calling conventions.
- Name mangling.
- Layout and size of types.
- Layout of virtual tables.
- Exception handling.
- Floating point mathematics.
- …

C++ Standard Library ABI:

Binary representation & conventions for C++ Standard Library facilities.

C++ Language ABI:

Binary representation & conventions for language facilities.

- Function calling conventions.
- Name mangling.
- Layout and size of types.
- Layout of virtual tables.
- Exception handling.
- Floating point mathematics.
- …

C++ Standard Library ABI:

Binary representation & conventions for C++ Standard Library facilities.

- Linkage of `std::` functions.
- `std::` name mangling.
- Layout and size of `std::` types.
- `std::` virtual tables.
- `std::` `constexpr` values and functions.
- `<type_traits>` and `std::` concepts.
- …

C++ Language ABI:

Binary representation & conventions for language facilities.

- Function calling conventions.
- Name mangling.
- Layout and size of types.
- Layout of virtual tables.
- Exception handling.
- Floating point mathematics.
- …

C++ Standard Library ABI:

Binary representation & conventions for C++ Standard Library facilities.

- Linkage of `std::` functions.
- `std::` name mangling.
- Layout and size of `std::` types.
- `std::` virtual tables.
- `std::` `constexpr` values and functions.
- `<type_traits>` and `std::` concepts.
- …

API Stability: Existing syntax and semantics should rarely change.

API Stability: Existing syntax and semantics should rarely change.

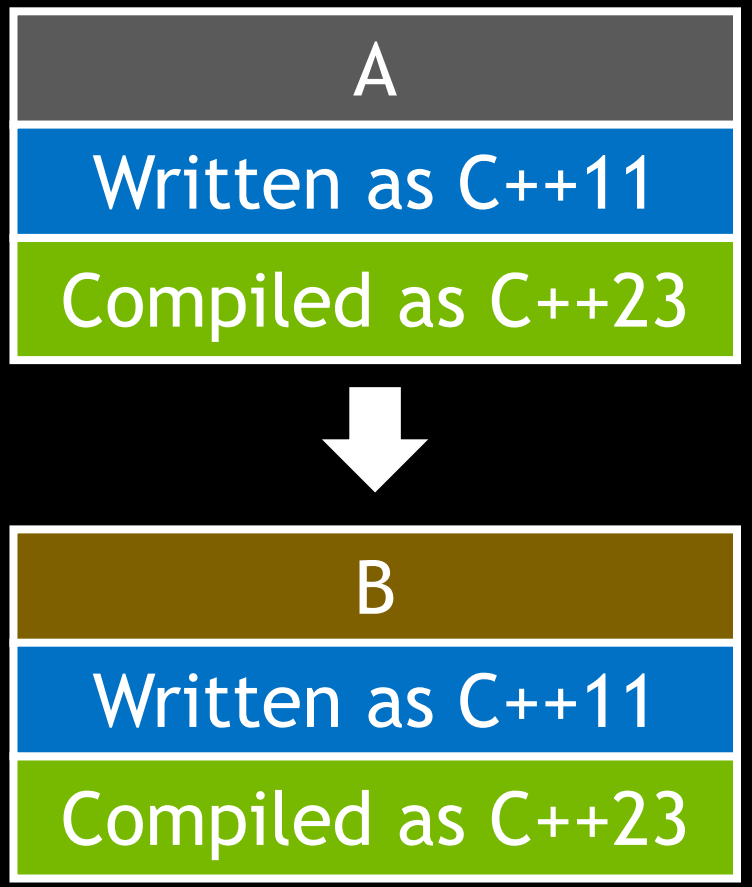
ABI Stability: Binary representations of existing facilities should rarely change.

Backward Compatibility:
Newer Builds, Older Code

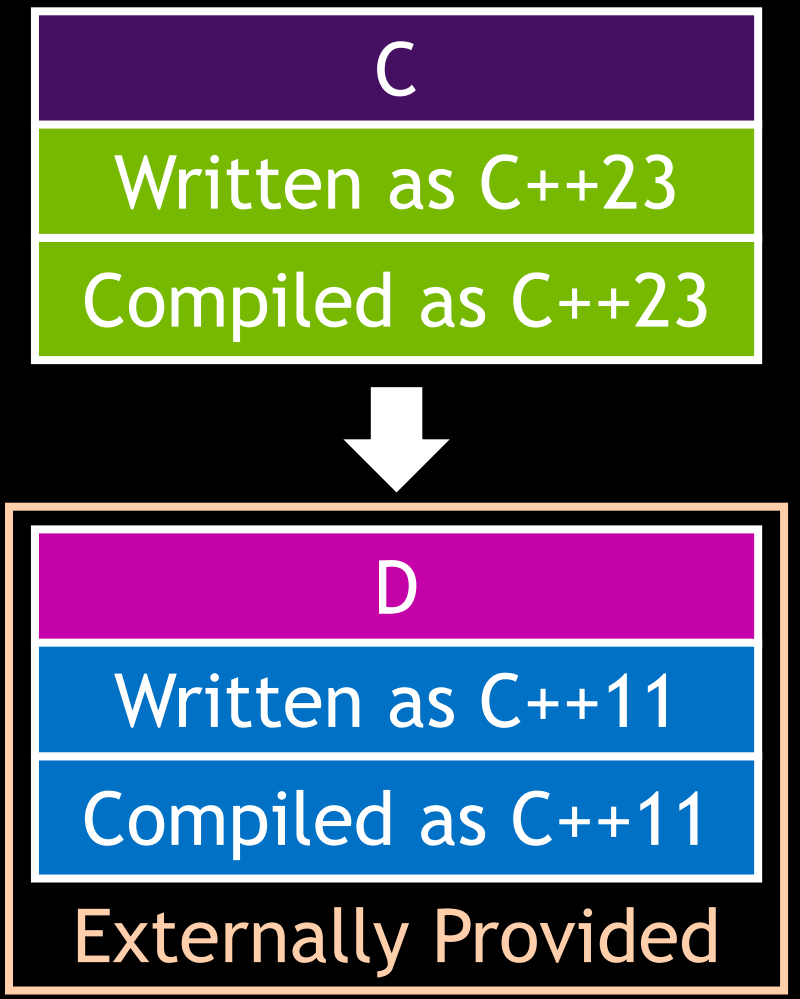
Backward Compatibility:
Newer Builds, Older Code

Forward Compatibility:
Older Builds, Newer Code

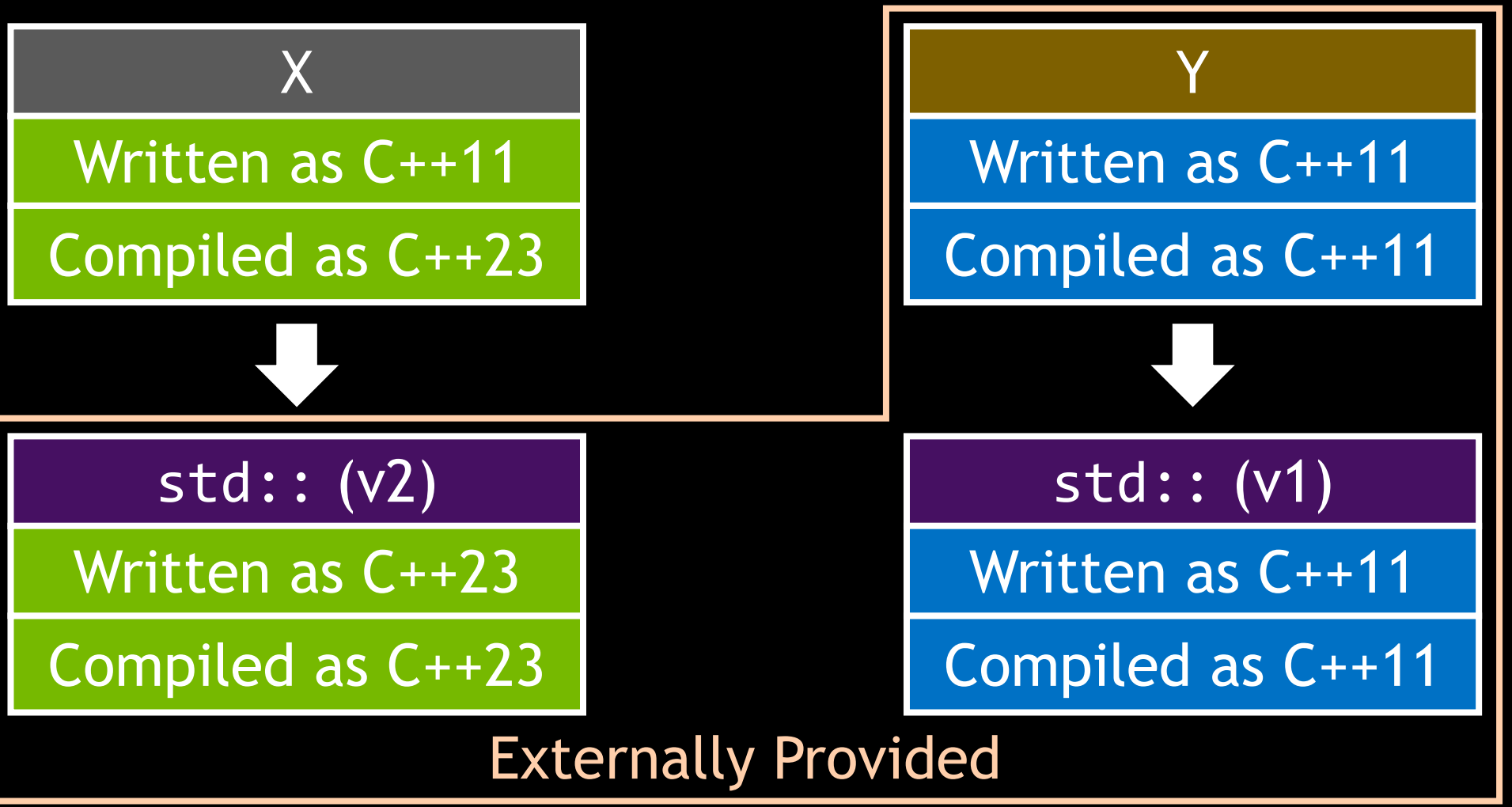
Backward Compatibility



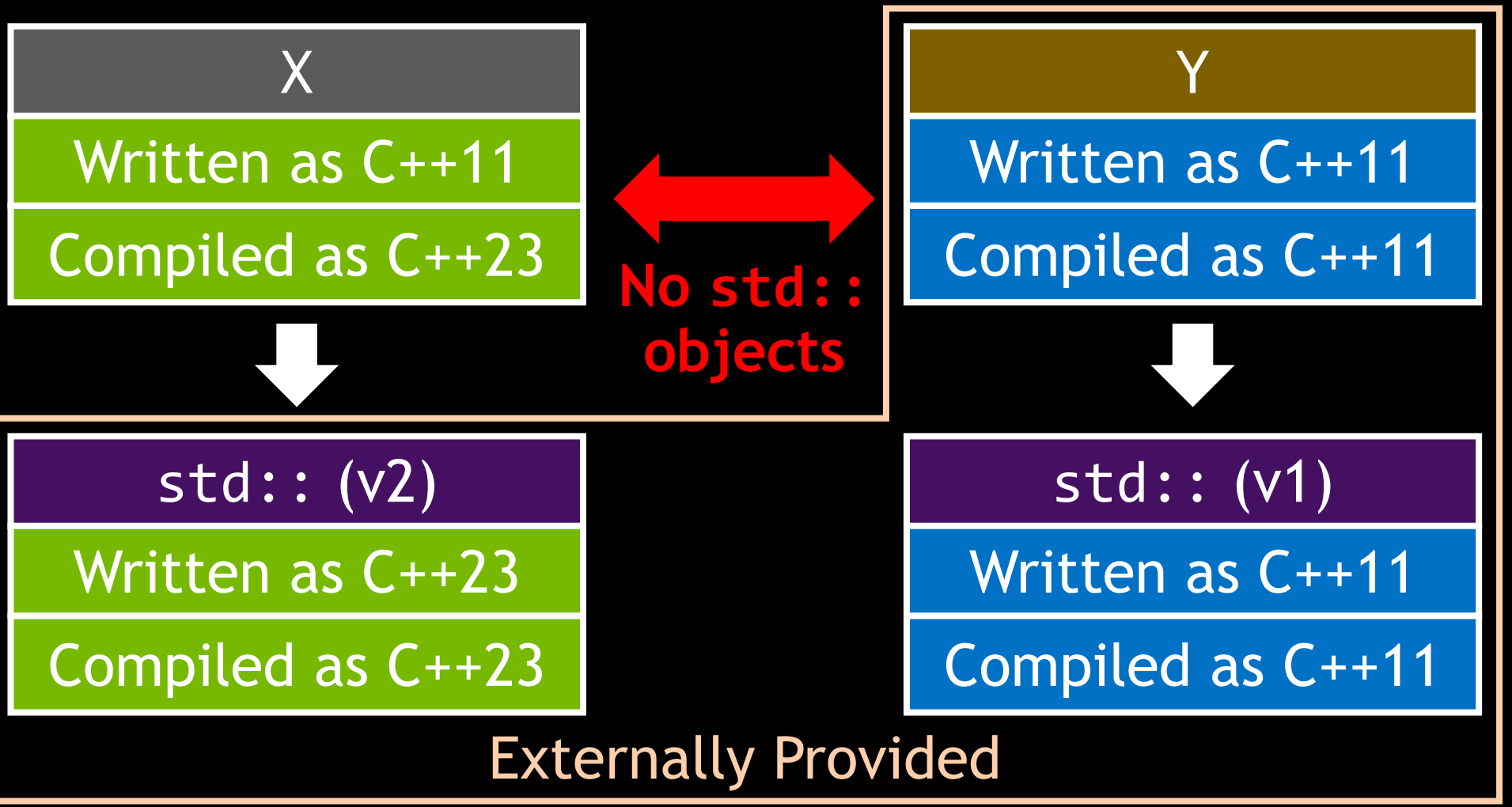
Forward Compatibility



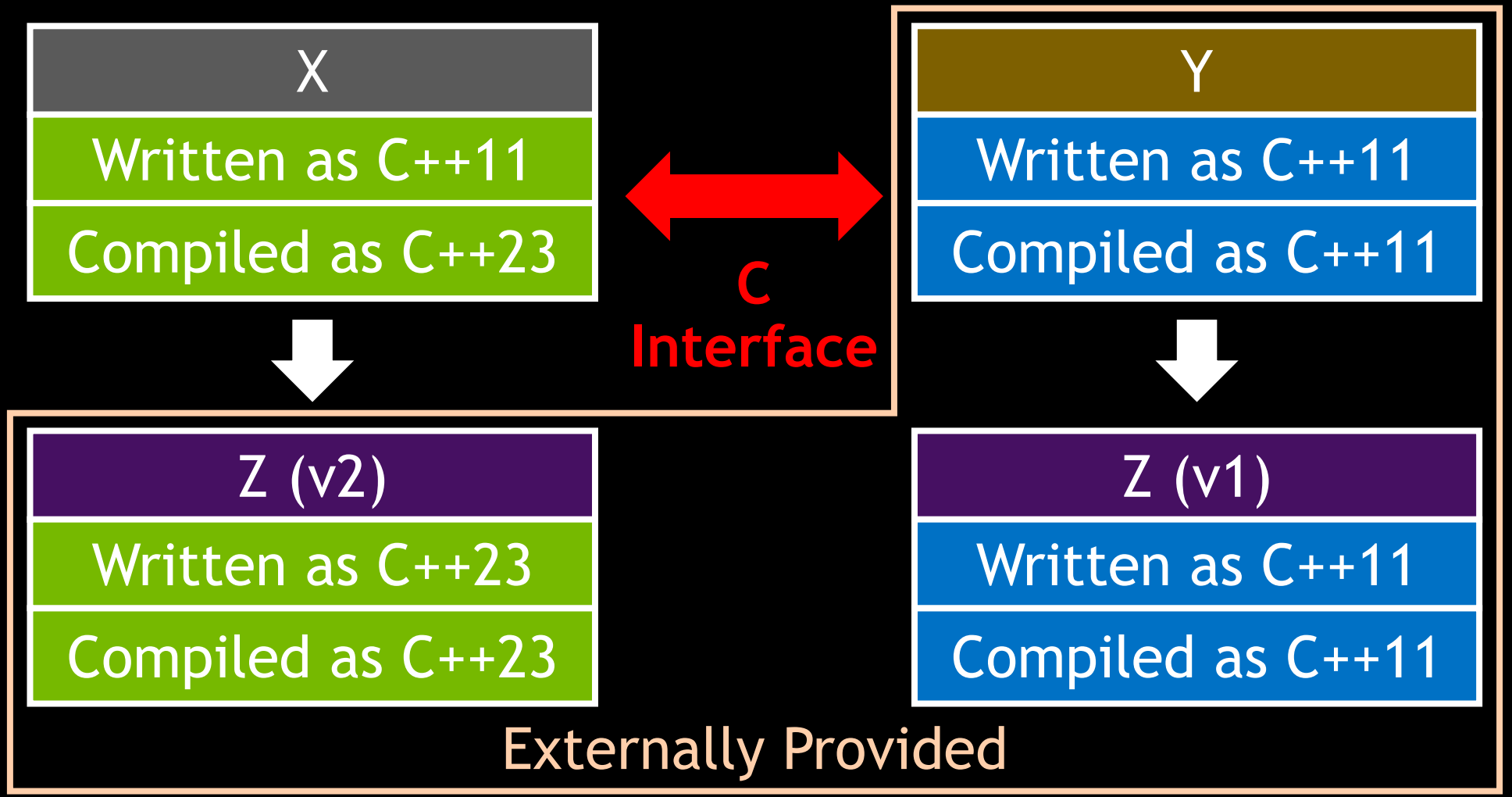
Dependencies Support All Versions



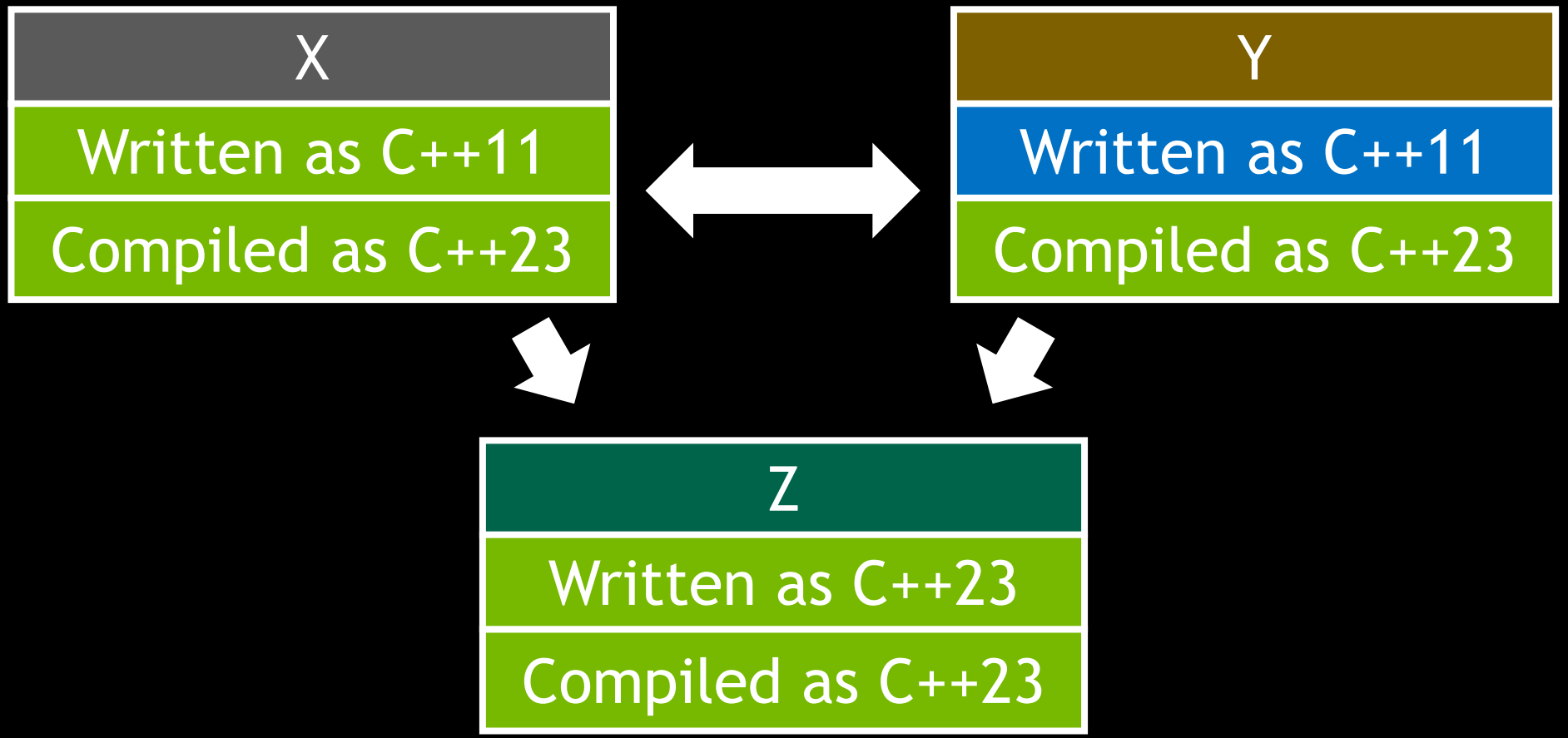
Dependencies Support All Versions



Write Your ABI Stable Interfaces In C



You Build All Dependencies



The Don’t Upgrade Solution:

If you can’t get newer dependency builds, don’t upgrade to a new C++ dialect.

Builds from different C++ dialects can’t be mixed.

Builds from different C++ dialects can't be mixed.

A project takes longer to move to new C++ dialects.

Builds from different C++ dialects can't be mixed.

A project takes longer to move to new C++ dialects.

Things that depend on it take longer to move to new C++ dialects

Builds from different C++ dialects can't be mixed.

A project takes longer to move to new C++ dialects.

Things that depend on it take longer to move to new C++ dialects

How does a breaking change manifest?

How does a breaking change manifest?

- Build Time?
  - Compile Time?
  - Link Time?
- Run Time?
  - Graceful?
  - Catastrophic?
- Undefined Behavior?

std:: Parameters Exposes You To ABI Breaks

A	Compiled as C++11

```
void f(std::string& s) { /* ... */ }
```

B	Compiled as C++23

```
void g() {
  f(std::string("BAL"));
}
```


std:: Parameters Exposes You To ABI Breaks

A — Compiled as C++11

```
void f(std::string& s) { /* ... */ }
```

B — Compiled as C++23

```
void g() {
  f(std::string("BAL"));
}
```


`std::` Return Types Exposes You To ABI Breaks

A — Compiled as C++11

```
std::string f() { /* ... */ }
```

B — Compiled as C++23

```
std::string g() {
  std::string s = f();
  return s + "bryce";
}
```

std:: Data Members Exposes You To ABI Breaks

A — Compiled as C++11

```
struct X {
  std::string s;
};
X make_x();
```

B — Compiled with C++23

```
void g() {
  X x = make_x();
  x.s = "hello world\n";
}
```

Inlining Of `std::` Exposes You To ABI Breaks

```
void f() {
  std::string s;
  s.reserve(42);
  s += "adelstein";
}
```

Inlining Of `std::` Exposes You To ABI Breaks

```
void f() {
  std::string s;
  s.reserve(42);    // What if this is inlined,
  s += "adelstein"; // but this isn’t?
}
```

std:: Default Params Expose You To ABI Breaks

```
namespace std {

template <class T, std::size_t E = std::dynamic_extent>
class span;

}
```

std:: Default Params Expose You To ABI Breaks

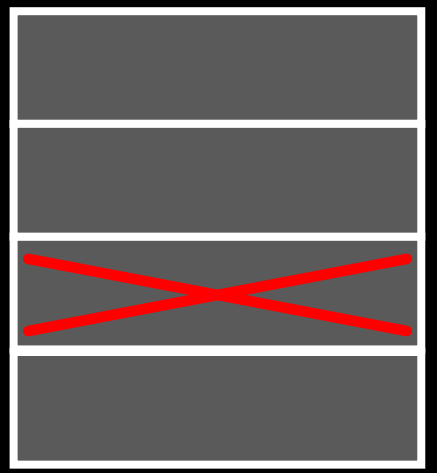
```
namespace std {

template <class T, std::size_t E = std::dynamic_extent>
class span;

}

void f(std::span<int>);
```

std:: Concepts Expose You To ABI Breaks

```
template <typename InputIt, typename OutputIt>
using intermediate_type = std::conditional_t<
  std::input_iterator<OutputIt>,
  typename std::iterator_traits<OutputIt>::value_type,
  typename std::iterator_traits<InputIt>::value_type
>;
```

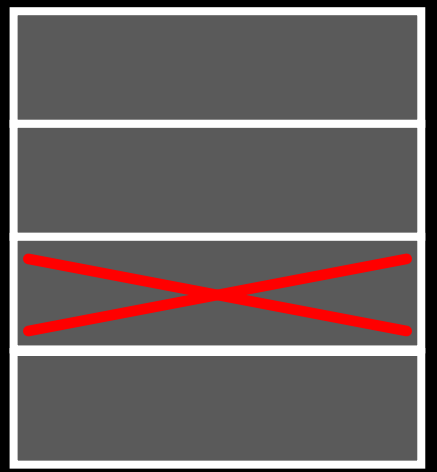
Polymorphism In `std::` Exposes You To ABI Breaks

Removing virtual functions causes layout changes.

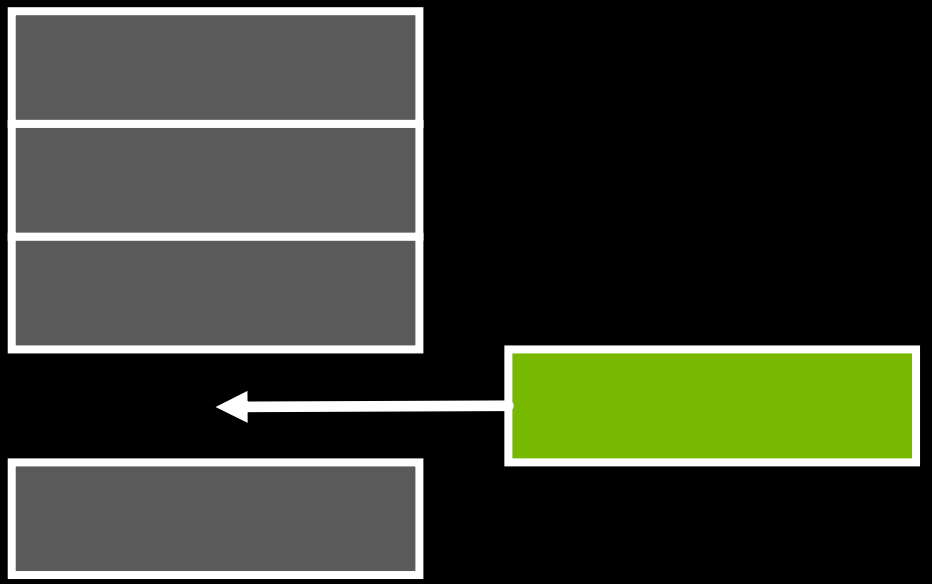


Polymorphism In `std::` Exposes You To ABI Breaks

Removing virtual functions causes layout changes.



Adding virtual functions causes layout changes.



In most cases, we can add functionality to facilities while preserving ABI.

For polymorphism, type erasure, and named concepts, adding functionality usually breaks ABI.

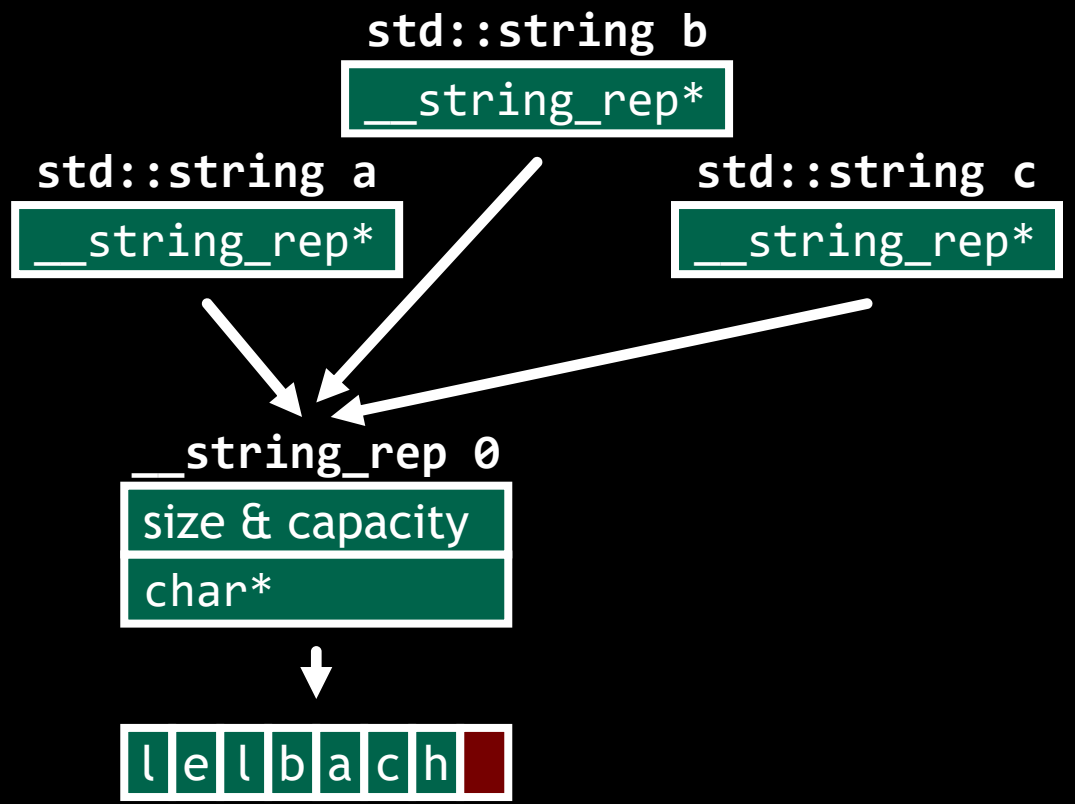
C++ Standard Library
polymorphism,
type erasure, and
named concepts
are fixed forever.

C++ Standard Library
polymorphism,
type erasure, and
named concepts
are fixed forever.

C++ Standard Library
polymorphism,
type erasure, and
named concepts
are fixed forever.

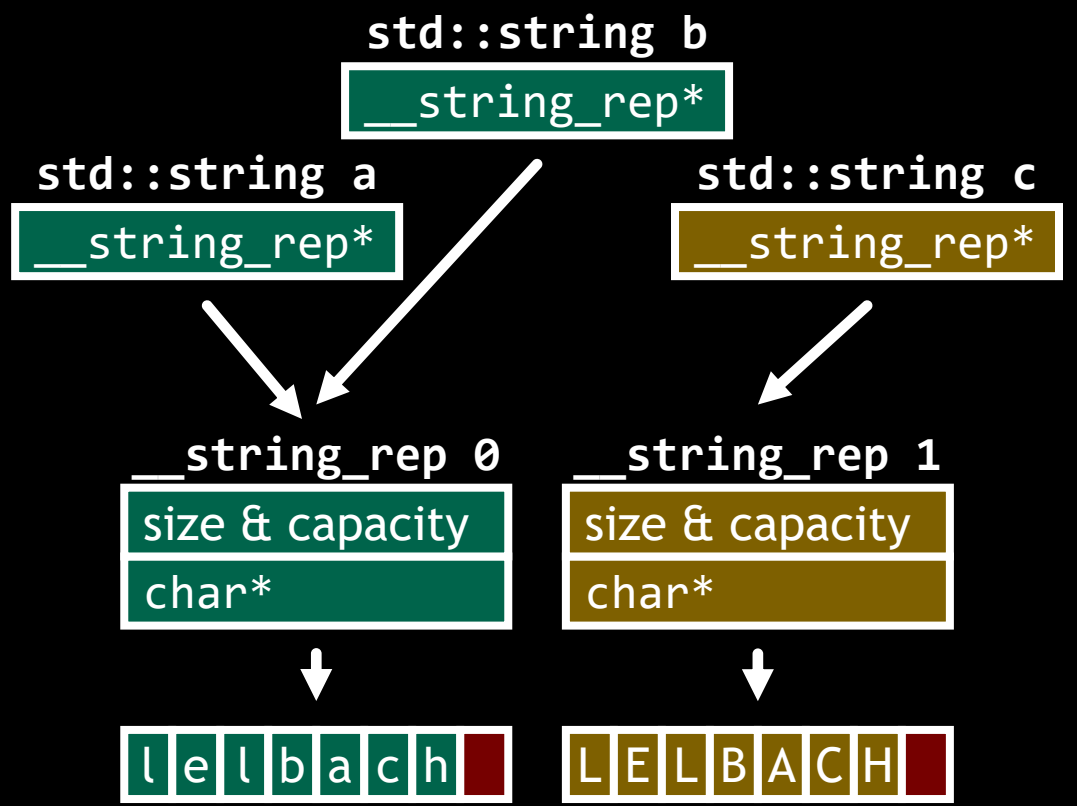
C++11 `std::string` COW Prohibition

In C++03 copy-on-write was allowed.



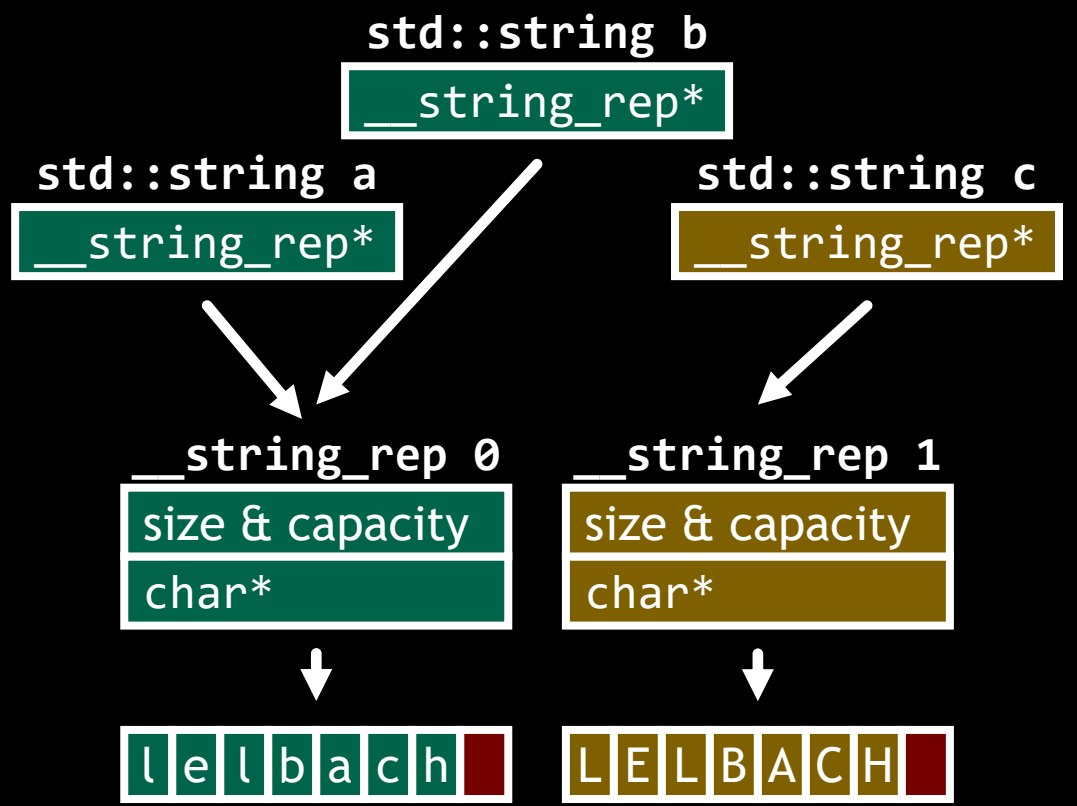
C++11 `std::string` COW Prohibition

In C++03 copy-on-write was allowed.



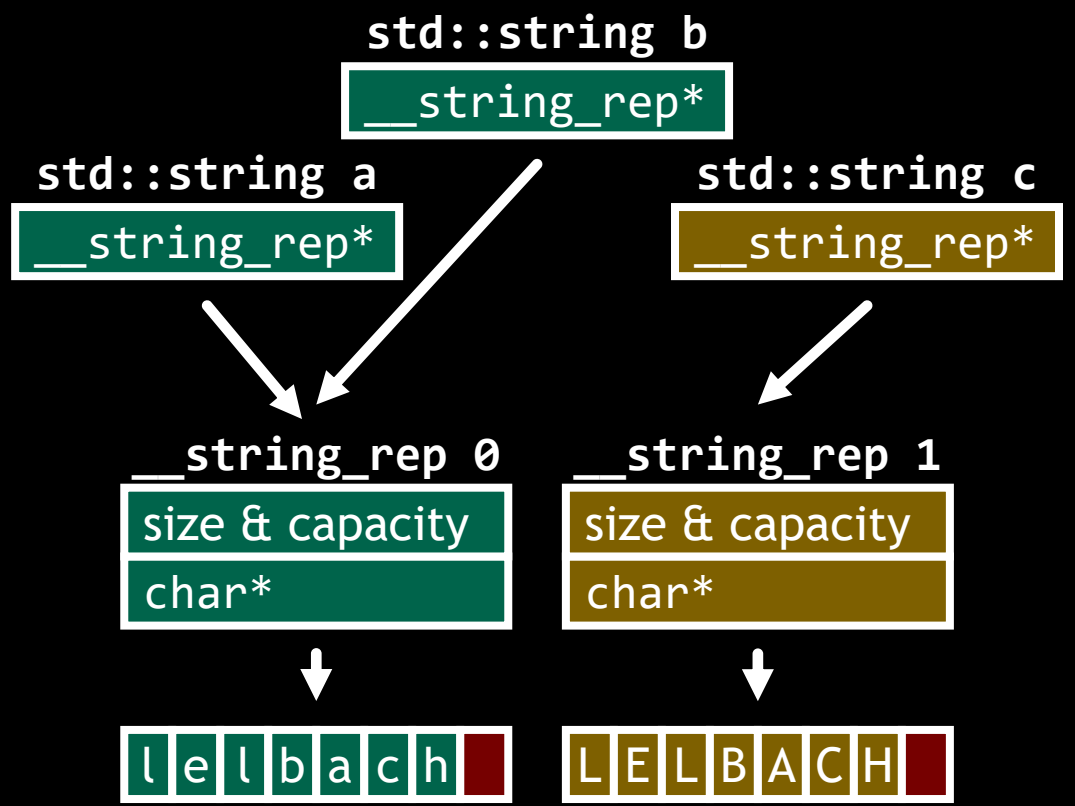
C++11 `std::string` COW Prohibition

In C++03 copy-on-write was allowed.

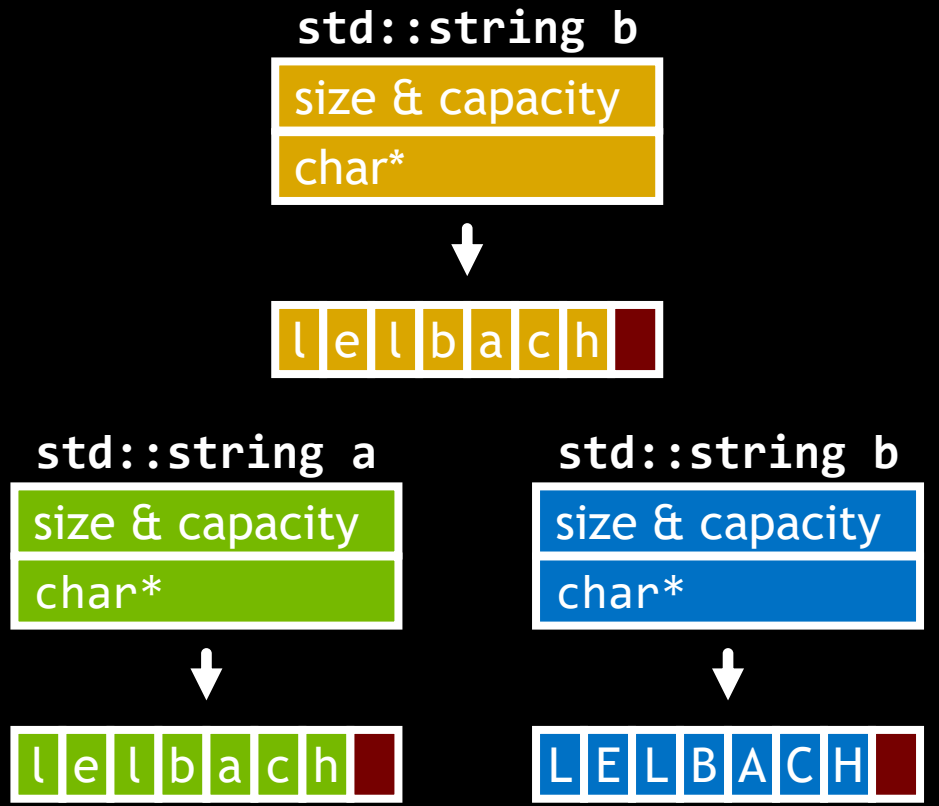


C++11 `std::string` COW Prohibition

In C++03 copy-on-write was allowed.



But in C++11, we prohibited it.



Was the C++11 `std::string` ABI break the right decision?

Was the C++11 `std::string` ABI break the right decision?

C++17 Non-Allocating `std::system_error::message`

```
virtual
~error_category();
```

```
virtual char const*
name() const noexcept = 0;
```

```
virtual error_condition
default_error_condition(int) const noexcept;
```

```
virtual bool
equivalent(int, error_condition const&) const noexcept;
```

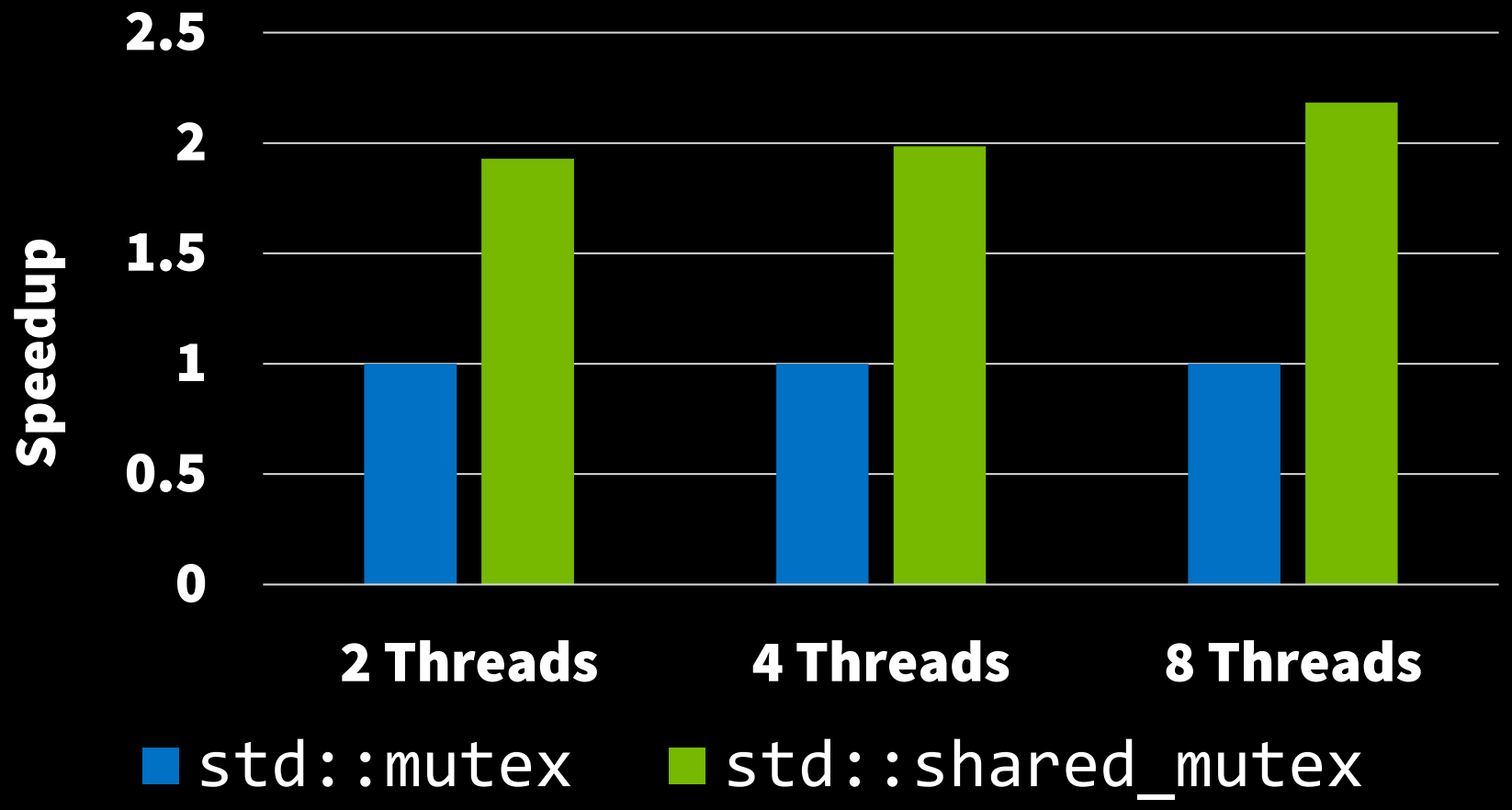
```
virtual bool
equivalent(error_code const&, int) const noexcept;
```

```
virtual char const*
message(int, char*, size_t) const noexcept;
```

```
virtual char const*
message(int, char*, size) const = 0;
```

Proposed addition was non-pure; existing derived classes would continue to compile.

std::mutex Performance with MSVC



Source: programming.vip/docs/performance-comparison-between-std-shared_mutex-and-std-mutex-banchmark.html

`std::mutex` Performance with MSVC

```
sizeof(std::mutex)        == 80

sizeof(std::shared_mutex) == 8
```

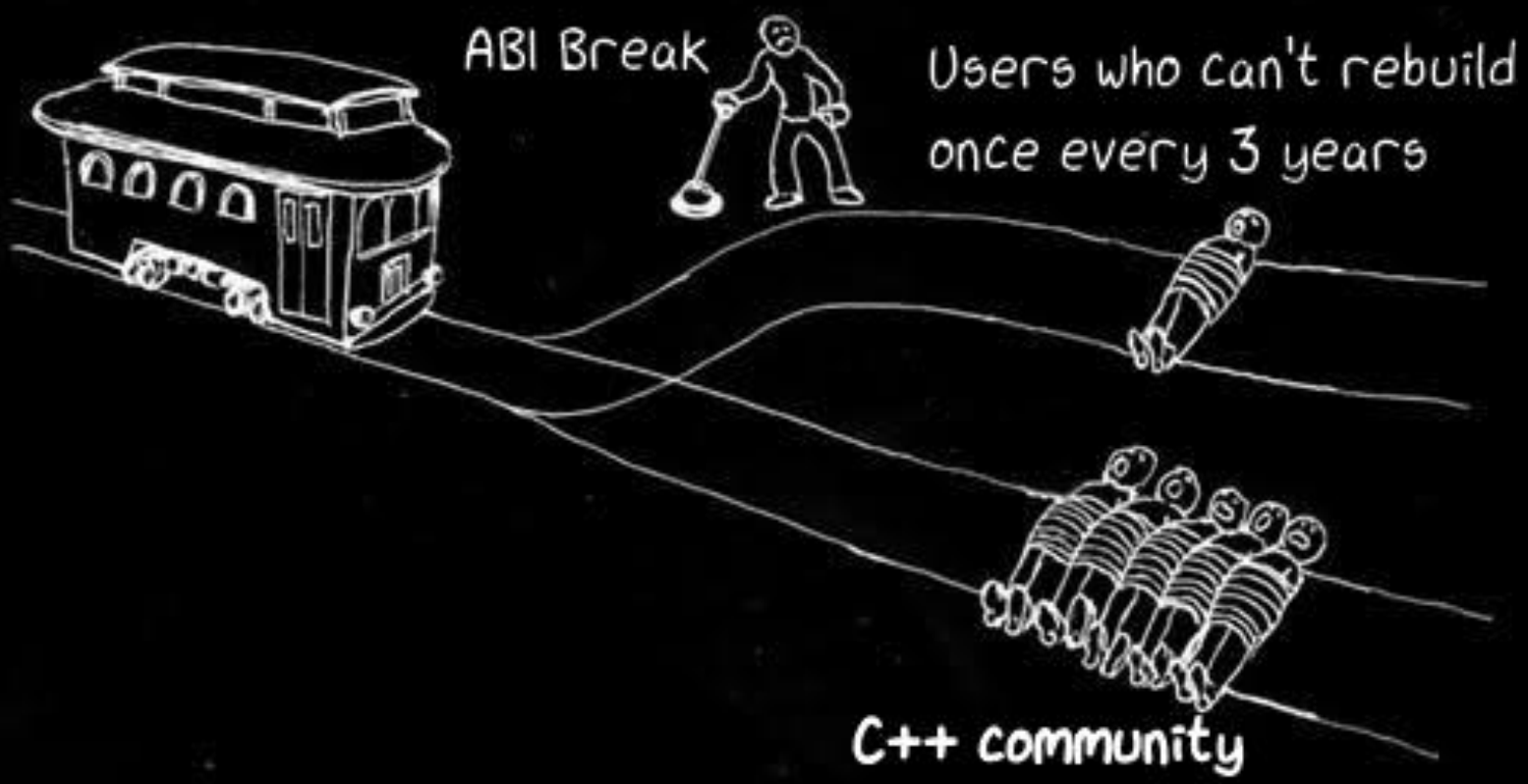
The C++ Standard Library is:

- Good at stability.

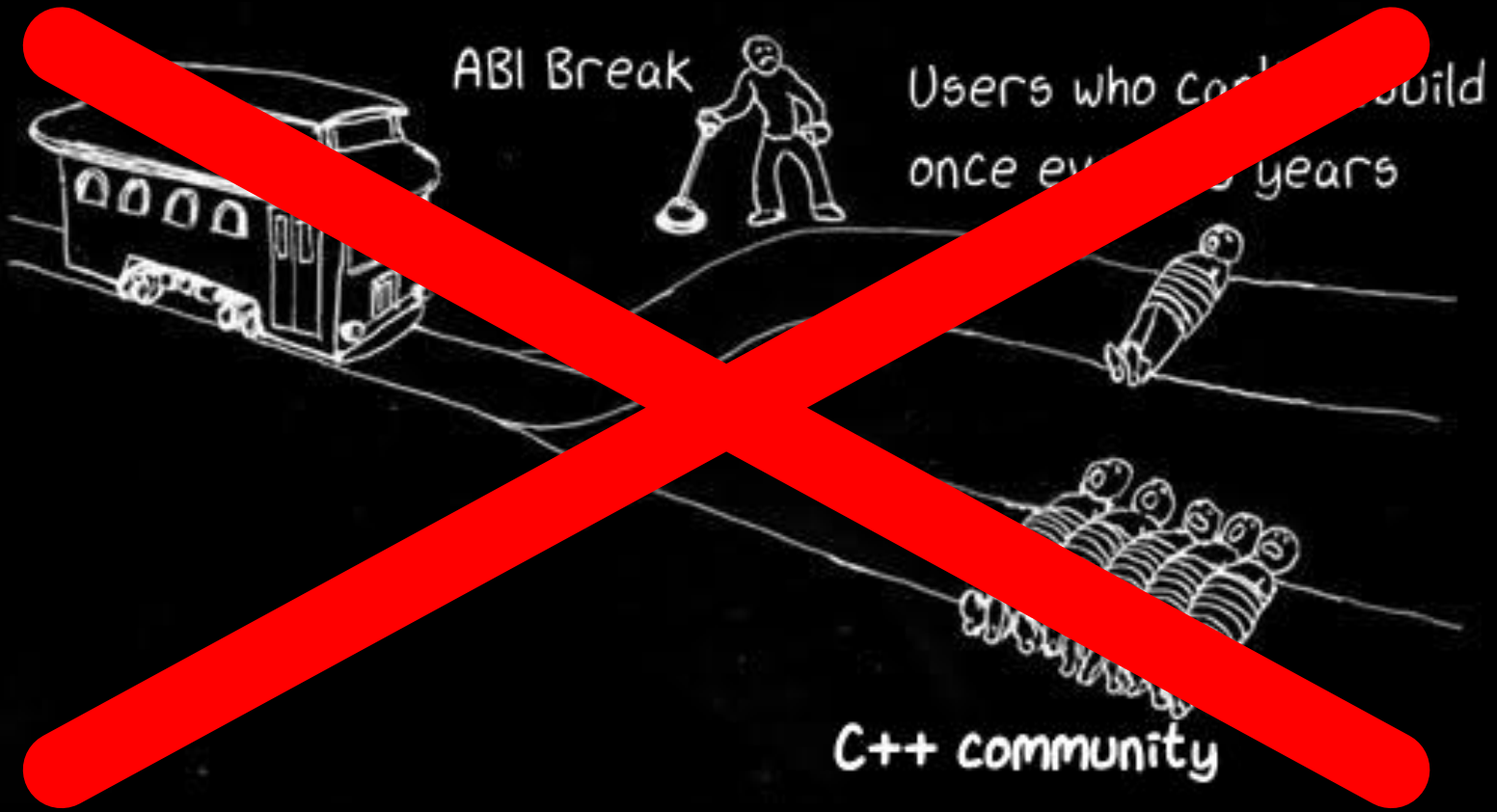
The C++ Standard Library is:

- Good at stability.
- Bad at fixing mistakes.

Stability vs Velocity



Stability vs Velocity



Stability vs Velocity is a myth.

We can’t make a binary choice here.

Today, all major implementations guarantee some degree of long term stability.

It is unrealistic to think we will stop caring about stability.

But stability inhibits velocity.

But stability inhibits velocity.

We can’t fix this via policy.

But stability inhibits velocity.

We can’t fix this via policy.

We need technical solutions.

Inline Namespaces

```
namespace std {
inline namespace __cxxNN {

template <
  class C,
  class T = std::char_traits<C>,
  class A = std::allocator<C>>
class basic_string;

using string = basic_string<char>;

}}
```


Inline Namespaces

```
namespace std {
inline namespace __cxxNN {

template <
  class C,
  class T = std::char_traits<C>,
  class A = std::allocator<C>>
class basic_string;

using string = basic_string<char>;

}}
```

A — Compiled as C++11

```
void f(std::string& s) {
  /* ... */
}
```

B — Compiled as C++23

```
void g() {
  std::string s("BAL");
  f(s);
}
```

Inline Namespaces

```
namespace std {
inline namespace __cxxNN {

template <
  class C,
  class T = std::char_traits<C>,
  class A = std::allocator<C>>
class basic_string;

using string = basic_string<char>;

}}
```

A — Compiled as C++11

```
void f(std::__cxx11::string& s) {
  /* ... */
}
```

B — Compiled as C++23

```
void g() {
  std::__cxx23::string s("BAL");
  f(s);
}
```

Inline Namespaces

```
namespace std {
inline namespace __cxxNN {

template <
  class C,
  class T = std::char_traits<C>,
  class A = std::allocator<C>>
class basic_string;

using string = basic_string<char>;

}}
```

A — Compiled as C++11

```
void f(std::string& s) {
  /* ... */
```

Mangling:

```
_Z1fRNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
```

B — Compiled as C++23

```
void g() {
  std::string s(/* ... */);
  f(s);
```

Expected Mangling:

```
_Z1fRNSt7__cxx2312basic_stringIcSt11char_traitsIcESaIcEEE
```

Inline Namespaces

```
namespace std {
inline namespace __cxx11 {

using string = basic_string<char>;

}

inline namespace __cxx23 {

using string = basic_string<char>;

}}
```

A — Compiled as C++11

```
void f(std::string& s) {
  /* ... */
}
```

B — Compiled as C++23

```
void g() {
  std::__cxx11::string s(/* ... */);
  f(s);
}
```

Inline Namespaces

```
namespace std {
inline namespace __cxxNN {

template <
  class C,
  class T = std::char_traits<C>,
  class A = std::allocator<C>>
class basic_string;

using string = basic_string<char>;

}}
```

A	Compiled as C++11

```
std::string f() {
  /* ... */
}
```

B	Compiled as C++23

```
std::string g() {
  std::string s = f();
  return s + "bryce";
}
```

Inline Namespaces

```
namespace std {
inline namespace __cxxNN {

template <
  class C,
  class T = std::char_traits<C>,
  class A = std::allocator<C>>
class basic_string;

using string = basic_string<char>;

}}
```

A — Compiled as C++11

```
std::__cxx11::string f() {
  /* ... */
}
```

Mangling: `_Z1fv`

B — Compiled as C++23

```
std::__cxx23::string g() {
  std::__cxx23::string s = f();
  return s + "bryce";
}
```

Expected Mangling: `_Z1fv`

Inline Namespaces

```
namespace std {
inline namespace __cxxNN {

template <
  class C,
  class T = std::char_traits<C>,
  class A = std::allocator<C>>
class basic_string;

using string = basic_string<char>;

}}
```

A — Compiled as C++11

```
std::__cxx11::string f() {
  /* ... */
}
```

B — Compiled as C++23

```
std::__cxx23::string g() {
  std::__cxx23::string s = f();
  return s + "bryce";
}
```

SILENT ODR VIOLATION

Inline Namespaces

```
namespace std {
inline namespace __cxxNN {

template <
  class C,
  class T = std::char_traits<C>,
  class A = std::allocator<C>>
class basic_string;

using string = basic_string<char>;

}}
```

A — Compiled as C++11

```
struct X {
  std::string s;
};
X make_x();
```

B — Compiled as C++23

```
struct X {
  std::string s;
};
void g() { X x = make_x(); }
```

Inline Namespaces

```
namespace std {
inline namespace __cxxNN {

template <
  class C,
  class T = std::char_traits<C>,
  class A = std::allocator<C>>
class basic_string;

using string = basic_string<char>;

}}
```

A — Compiled as C++11

```
struct X {
  std::__cxx11::string s;
};
X make_x();
```

B — Compiled as C++23

```
struct X {
  std::__cxx23::string s;
};
void g() { X x = make_x(); }
```

SILENT ODR VIOLATION

Inline Namespaces

```
namespace std {
inline namespace __cxxNN {

template <
  class C,
  class T = std::char_traits<C>,
  class A = std::allocator<C>>
class basic_string;

using string = basic_string<char>;

}}
```

ABI Problem	Diagnoses?	Solves?
Parameters	☑	☒
Return Types	☒	☒
Non-Local Variables	☑	☒
Data Members	☒	☒
Inlining	☒	☒
Constant Evaluation	☑	☒
Polymorphism	☑	☒

Interfaces

```
struct point {
  interface(std::cxx23) {
    int x, y, z;
    interface(std::cxx26) int w;

    int get_x() const { return x; }
    int get_y() const { return y; }
    int get_z() const { return z; }

    int get_w() const interface(std::cxx26) { return w; }
  }
};

sizeof(interface(std::cxx23) point) == 12
sizeof(interface(std::cxx26) point) == 16
```

Source: P2123: Extending The Type System To Provide API And ABI Flexibility; Hal Finkel & Tom Scogland

Interfaces

```
struct point {
  interface(std::cxx23) {
    int x, y, z;
    interface(std::cxx26) int w;

    int get_x() const { return x; }
    int get_y() const { return y; }
    int get_z() const { return z; }

    int get_w() const interface(std::cxx26) { return w; }
  }
};

sizeof(interface(std::cxx23) point) == 12
sizeof(interface(std::cxx26) point) == 16
```

Source: P2123: Extending The Type System To Provide API And ABI Flexibility; Hal Finkel & Tom Scogland

Interfaces

```
struct point {
  interface(std::cxx23) {
    int x, y, z;
    interface(std::cxx26) int w;

    int get_x() const { return x; }
    int get_y() const { return y; }
    int get_z() const { return z; }

    int get_w() const interface(std::cxx26) { return w; }
  }
};

sizeof(interface(std::cxx23) point) == 12
sizeof(interface(std::cxx26) point) == 16
```

Source: P2123: Extending The Type System To Provide API And ABI Flexibility; Hal Finkel & Tom Scogland

Interfaces

```
struct point {
  interface(std::cxx23) {
    int x, y, z;
    interface(std::cxx26) int w;

    int get_x() const { return x; }
    int get_y() const { return y; }
    int get_z() const { return z; }

    int get_w() const interface(std::cxx26) { return w; }
  }
};

sizeof(interface(std::cxx23) point) == 12
sizeof(interface(std::cxx26) point) == 16
```

Source: P2123: Extending The Type System To Provide API And ABI Flexibility; Hal Finkel & Tom Scogland

Interfaces

```
struct point {
  interface(std::cxx23) {
    int x, y, z;
    interface(std::cxx26) int w;

    int get_x() const { return x; }
    int get_y() const { return y; }
    int get_z() const { return z; }

    int get_w() const interface(std::cxx26) { return w; }
  }
};

sizeof(interface(std::cxx23) point) == 12
sizeof(interface(std::cxx26) point) == 16
```

Source: P2123: Extending The Type System To Provide API And ABI Flexibility; Hal Finkel & Tom Scogland

Interfaces

```
struct point {
  interface(std::cxx23) {
    int x, y, z;
    interface(std::cxx26) int w;

    int get_x() const { return x; }
    int get_y() const { return y; }
    int get_z() const { return z; }

    int get_w() const interface(std::cxx26) { return w; }
  }
};

sizeof(interface(std::cxx23) point) == 12
sizeof(interface(std::cxx26) point) == 16
```

Source: P2123: Extending The Type System To Provide API And ABI Flexibility; Hal Finkel & Tom Scogland

Interfaces

```
struct point {
  interface(std::cxx23) {
    int x, y, z;
    interface(std::cxx26) int w;

    int get_x() const { return x; }
    int get_y() const { return y; }
    int get_z() const { return z; }

    int get_w() const interface(std::cxx26) { return w; }
  }
};

sizeof(interface(std::cxx23) point) == 12
sizeof(interface(std::cxx26) point) == 16
```

Source: P2123: Extending The Type System To Provide API And ABI Flexibility; Hal Finkel & Tom Scogland

Interfaces

```
struct point {
  interface(std::cxx23) {
    int x, y, z;
    interface(std::cxx26) int w;

    int get_x() const { return x; }
    int get_y() const { return y; }
    int get_z() const { return z; }

    int get_w() const interface(std::cxx26) { return w; }
  }
};

sizeof(interface(std::cxx23) point) == 12
sizeof(interface(std::cxx26) point) == 16
```

Source: P2123: Extending The Type System To Provide API And ABI Flexibility; Hal Finkel & Tom Scogland

Interfaces

```
void f(interface(std::cxx23) std::string& s);
```

Source: P2123: Extending The Type System To Provide API And ABI Flexibility; Hal Finkel & Tom Scogland

Interfaces

```
void f(interface(std::cxx23) std::string& s);

void f(interface(std::cxx23+) std::string& s);
```

Source: P2123: Extending The Type System To Provide API And ABI Flexibility; Hal Finkel & Tom Scogland

Interfaces

```
void f(interface(std::cxx23) std::string& s);

void f(interface(std::cxx23+) std::string& s);
```

Source: P2123: Extending The Type System To Provide API And ABI Flexibility; Hal Finkel & Tom Scogland

Interfaces

ABI Problem	Diagnoses?	Solves?
Parameters	☑	☑
Return Types	☑	☑
Non-Local Variables	☑	☑
Data Members	☑	☑
Inlining	☑	☑
Constant Evaluation	☑	☑
Polymorphism	☑	☑

Source: P2123: Extending The Type System To Provide API And ABI Flexibility; Hal Finkel & Tom Scogland

The Stability Thesis:

Until we learn to change things after we ship them,

the C++ Standard Library should only contain things that are unlikely to need many changes.

Today, the C++ Standard Library is:

- Good at stability.
- Bad at fixing mistakes.

The C++ Standard Library
shouldn't innovate.

The C++ community
should innovate.

The C++ Standard Library ***should*** standardize existing practice.

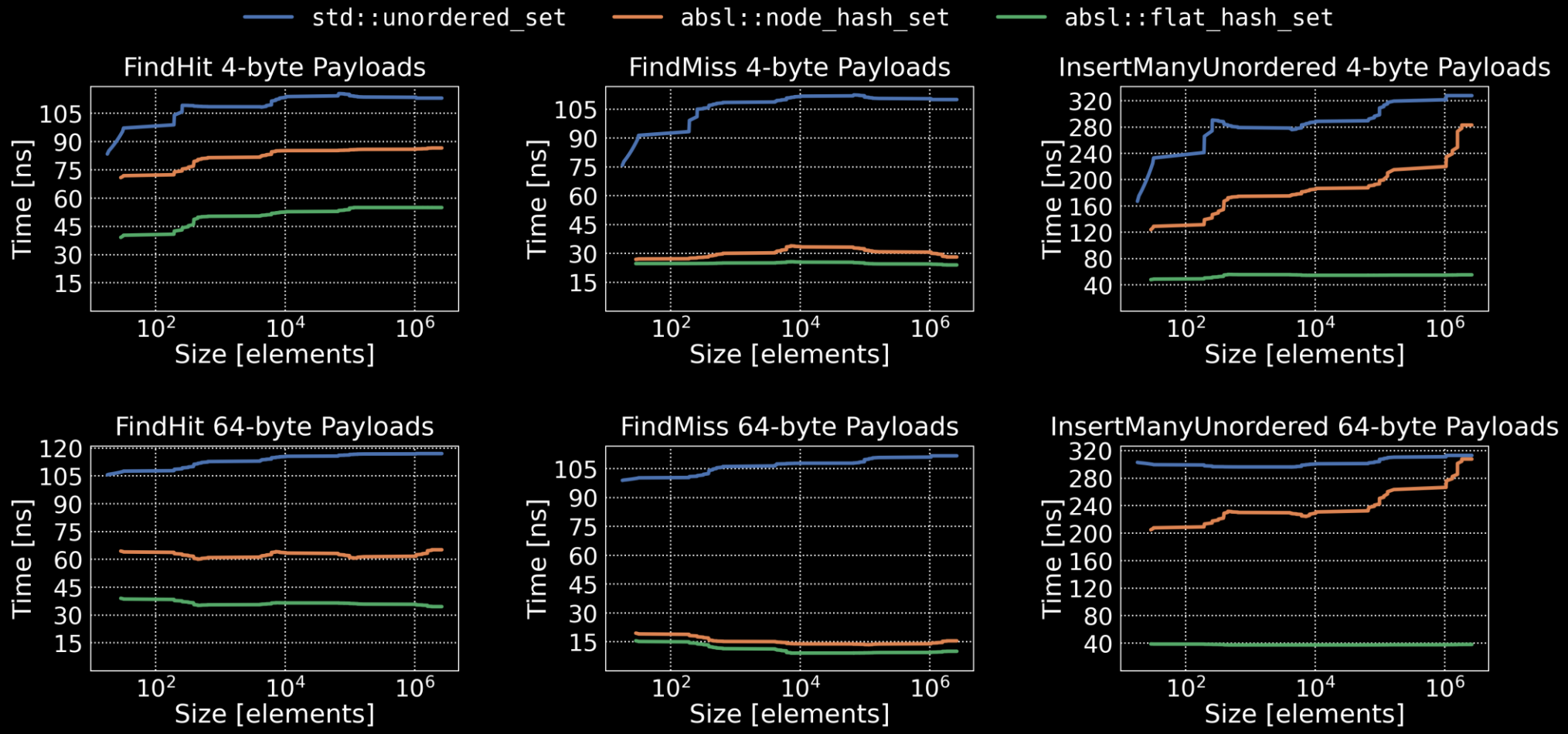
Anything that goes into the C++ Standard Library must stand the test of time.

Will we be happy with it in 10 to 20 years?

Avoid premature standardization
in evolving fields.

If there's substantial active research,
or the best practices change every
few years, it's not ready yet.

std:: Unordered Containers Performance



Source: Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step; Matt Kulukundis; CppCon 2017

We need to see ***field experience*** before we can standardize.

Implementation Experience:

Experience implementing
a proposed design.

Usage Experience:

Experience using implementations of the proposed design.

Deployment Experience:

Experience evolving and maintaining the proposed design over time.

Field Experience:

- Implementation Experience.
- Usage Experience.
- Deployment Experience.

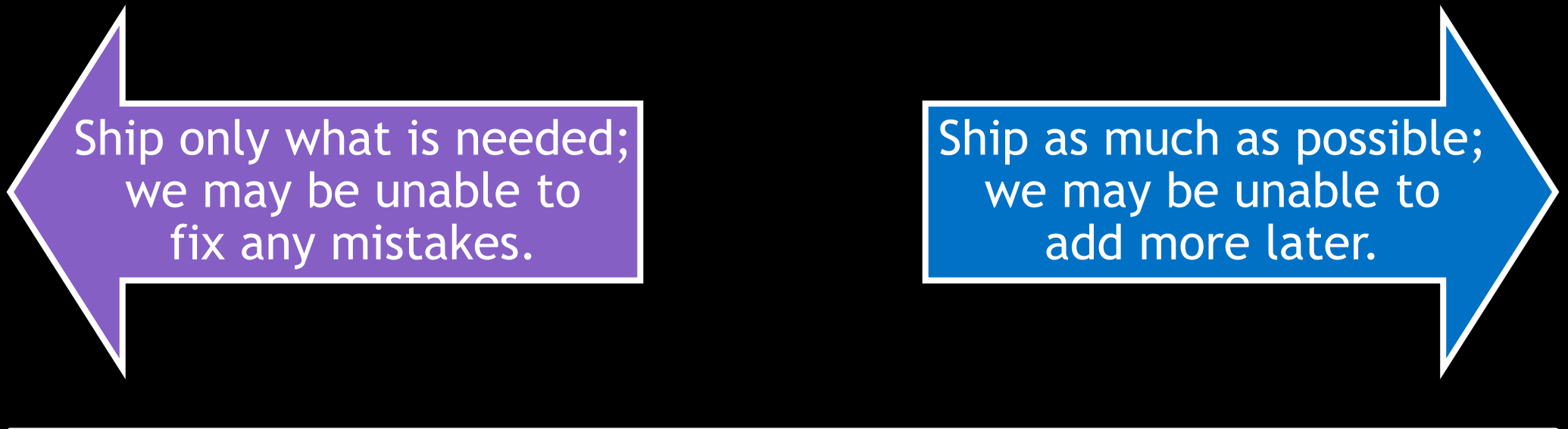
If we can’t change a feature after it ships, then we should…

Ship only what is needed; we may be unable to fix any mistakes.

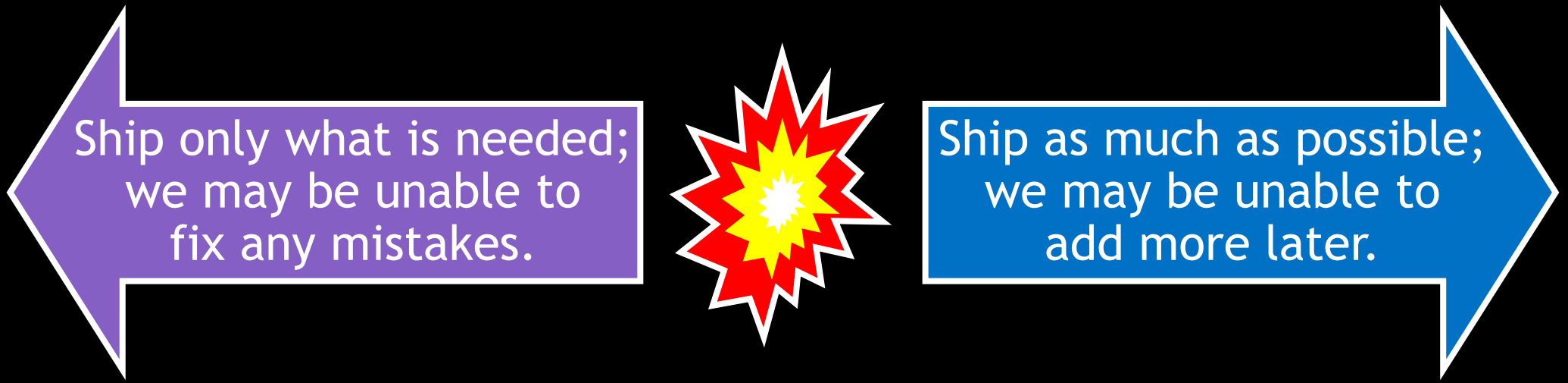
Smaller Scope

Larger Scope

If we can’t change a feature after it ships, then we should…



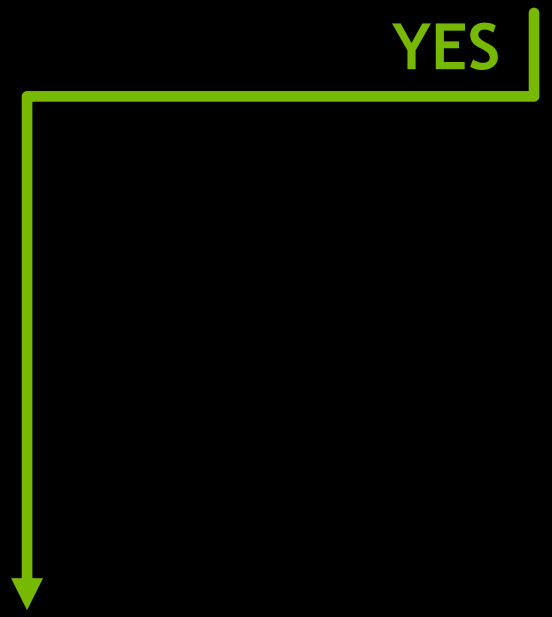
If we can't change a feature after it ships, then we should…



If we don’t include this in the initial release of the feature, is it a breaking change to add it later?

If we don’t include this in the initial release of the feature, is it a breaking change to add it later?

YES



It must be included in the initial release.

If we don't include this in the initial release of the feature, is it a breaking change to add it later?

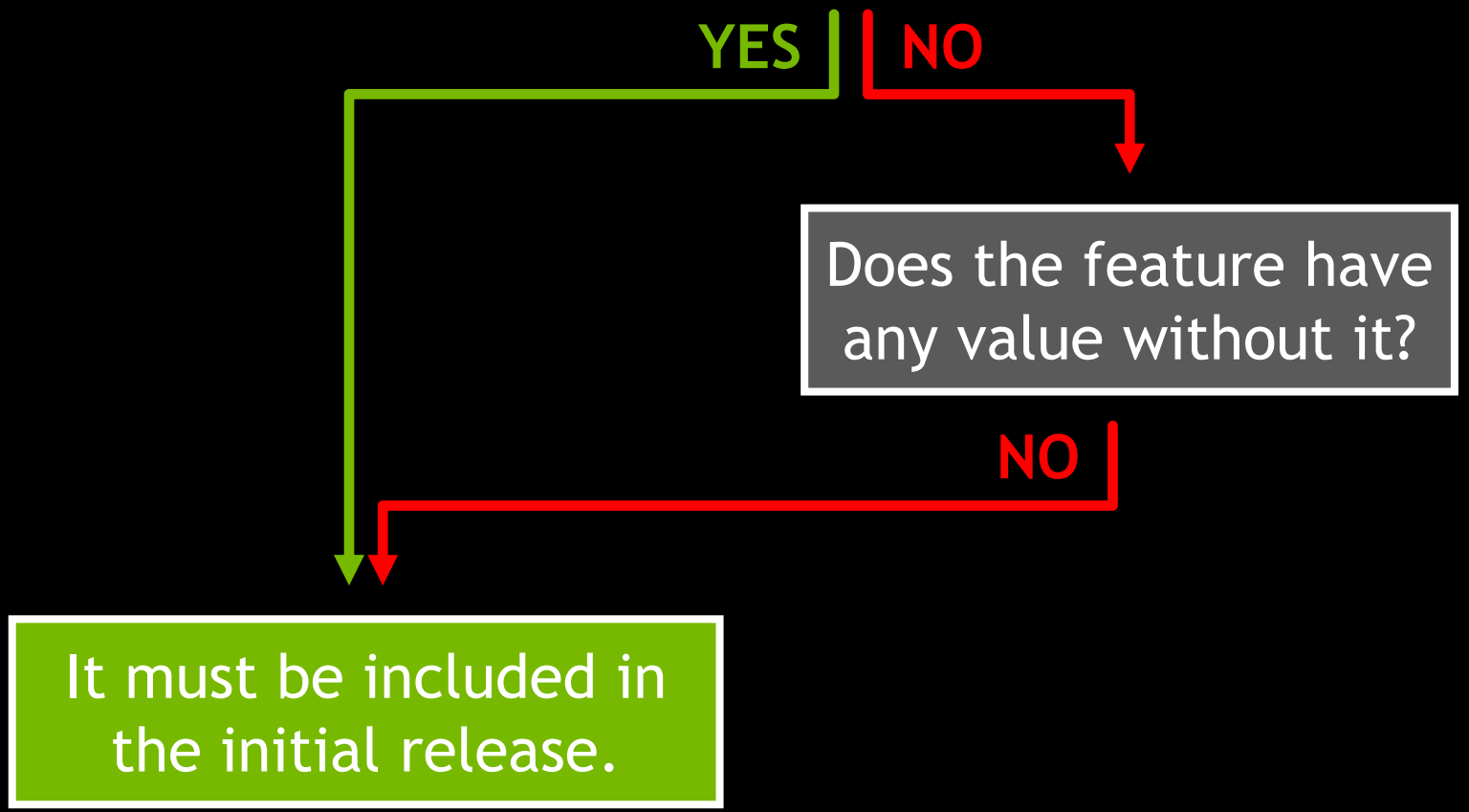
YES

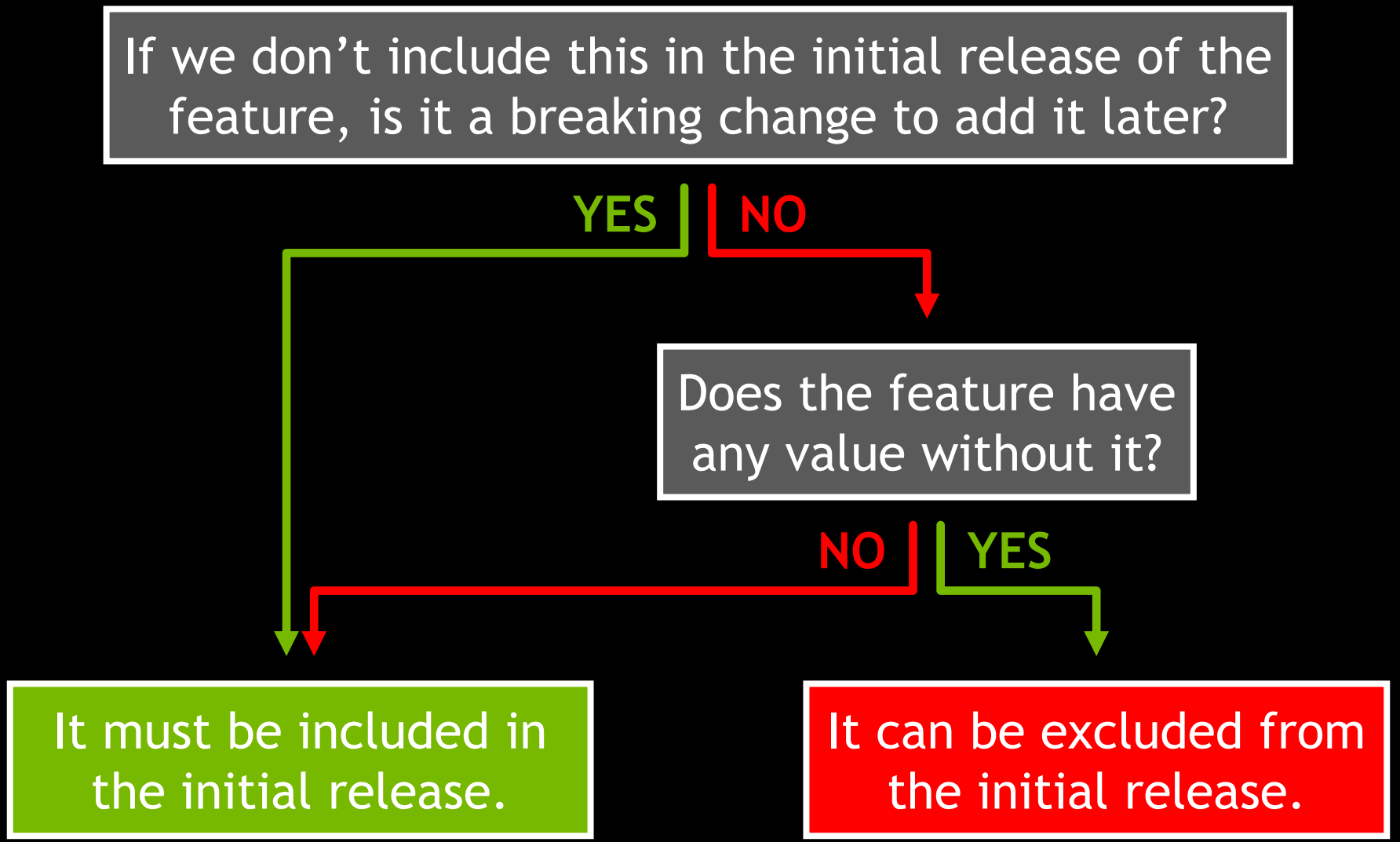
NO

Does the feature have any value without it?

It must be included in the initial release.

If we don't include this in the initial release of the feature, is it a breaking change to add it later?





The Stability Thesis:

Until we learn to change things after we ship them,

the C++ Standard Library should only contain things that are unlikely to need many changes.

The Necessity Thesis:

The C++ Standard Library should only contain facilities that can’t live elsewhere.

Language Support:

Facilities that require language support for correct or optimal implementation.

Language Support:

- <type_traits>

Language Support:

- `<type_traits>`
- `std::stacktrace`

Language Support:

- <type_traits>
- std::stacktrace
- std::tuple_element

Language Support:

- <type_traits>
- std::stacktrace
- std::tuple_element
- std::memcpy

Portability:

Facilities that provide portable abstractions of platform-specific behavior and interfaces.

Portability:

- `std::chrono`

Portability:

- `std::chrono`
- `std::atomic`

Portability:

- `std::chrono`
- `std::atomic`
- `std::sort`

Portability:

- std::chrono
- std::atomic
- std::sort
- std::numeric_limits

Vocabulary:

Facilities that need a common definition for interoperability across the C++ ecosystem.

Vocabulary:

```
void my_f(std::ranges::input_range&& r);
void your_f(std::ranges::input_range&& r);
```

Vocabulary:

```
void my_f(std::ranges::input_range&& r);
void your_f(std::ranges::input_range&& r);

void my_g(std::string_view s);
void your_g(std::string_view s);
```

Vocabulary:

```
void my_f(std::ranges::input_range&& r);
void your_f(std::ranges::input_range&& r);

void my_g(std::string_view s);
void your_g(std::string_view s);
```

Vocabulary:

- Concepts

Vocabulary:

- Concepts
- Containers and Views

Vocabulary:

- Concepts
- Containers and Views
- `<algorithm>`

Vocabulary:

- Concepts
- Containers and Views
- `<algorithm>`
- `std::format`

The Necessity Thesis:

The C++ Standard Library should only contain facilities that can't live elsewhere.

- Language Support
- Portability
- Vocabulary

The Usefulness Thesis:

The C++ Standard Library should expand in scope to contain anything that is useful to C++ programmers.

I hate to be the person that says we can’t have nice things,

but nice is not a sufficient motivation for standardization.

Default availability.

Is there a popular C++ library that has the facility you want to add to the C++ Standard Library?

Is there a popular C++ library that has the facility you want to add to the C++ Standard Library?

NO

There's no existing practice; how could we standardize it?

Is there a popular C++ library that has the facility you want to add to the C++ Standard Library?

NO

There's no existing practice; how could we standardize it?

YES

Why isn't that good enough?

Is there a popular C++ library that has the facility you want to add to the C++ Standard Library?

NO

There's no existing practice; how could we standardize it?

YES

Why isn't that good enough?

My company or project only lets me use things in the C++ Standard Library.

Is there a popular C++ library that has the facility you want to add to the C++ Standard Library?

NO

There's no existing practice; how could we standardize it?

YES

Why isn't that good enough?

My company or project only lets me use things in the C++ Standard Library.

Standardization is not a substitute for culture change.

Is there a popular C++ library that has the facility you want to add to the C++ Standard Library?

NO

There's no existing practice; how could we standardize it?

YES

Why isn't that good enough?

There's a specific reason it would be better in the C++ Standard Library.

My company or project only lets me use things in the C++ Standard Library.

Standardization is not a substitute for culture change.

Is there a popular C++ library that has the facility you want to add to the C++ Standard Library?

NO

There's no existing practice; how could we standardize it?

YES

Why isn't that good enough?

There's a specific reason it would be better in the C++ Standard Library.

Can you live with making no breaking changes to it for the next 20 years?

My company or project only lets me use things in the C++ Standard Library.

Standardization is not a substitute for culture change.

Is there a popular C++ library that has the facility you want to add to the C++ Standard Library?

NO

There's no existing practice; how could we standardize it?

YES

Why isn't that good enough?

There's a specific reason it would be better in the C++ Standard Library.

Can you live with making no breaking changes to it for the next 20 years?

YES

Is it worth 5x the effort and time that it would take just to put it on GitHub?

My company or project only lets me use things in the C++ Standard Library.

Standardization is not a substitute for culture change.

Is there a popular C++ library that has the facility you want to add to the C++ Standard Library?

NO

There's no existing practice; how could we standardize it?

YES

Why isn't that good enough?

There's a specific reason it would be better in the C++ Standard Library.

Can you live with making no breaking changes to it for the next 20 years?

YES

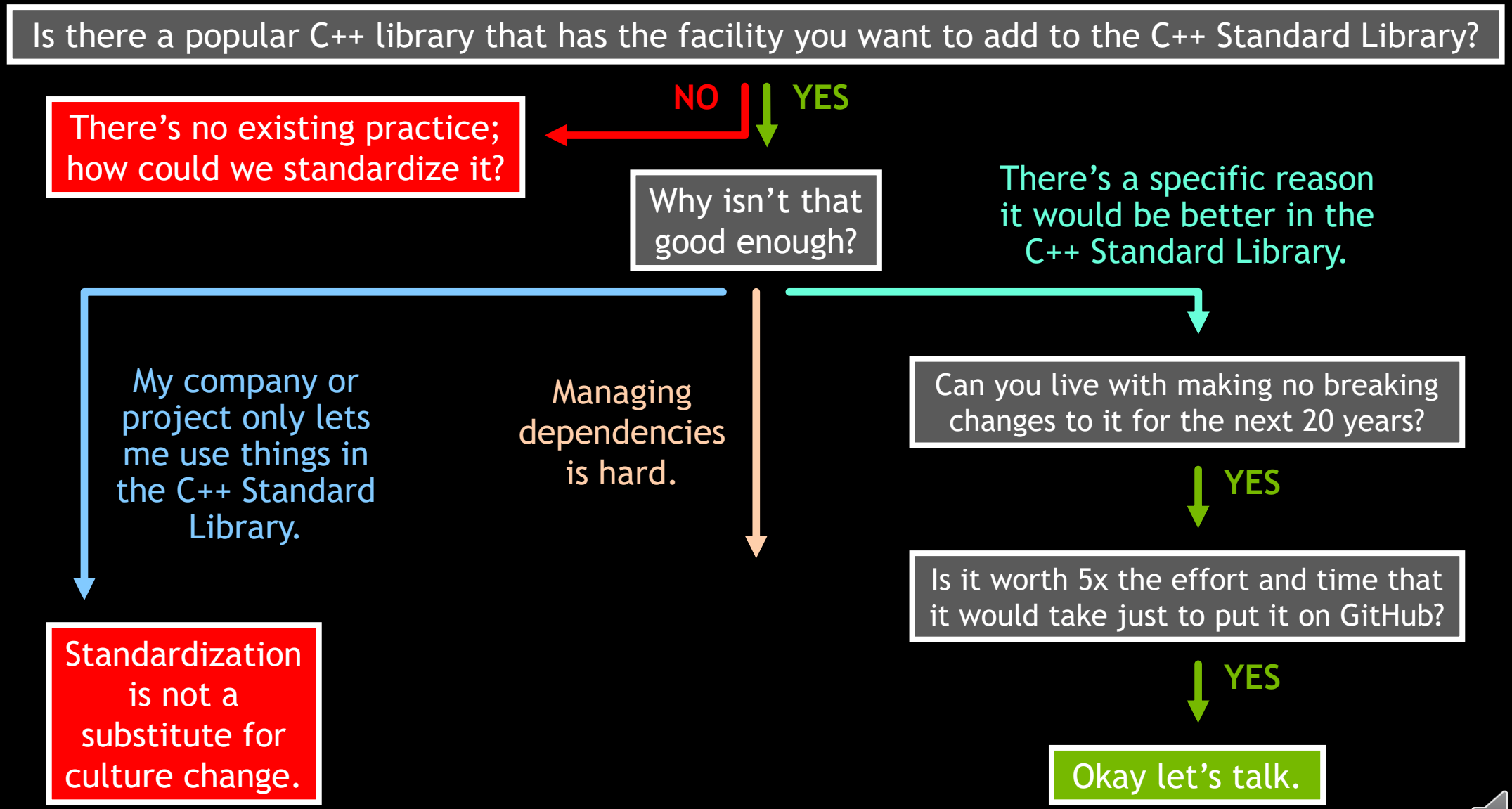
Is it worth 5x the effort and time that it would take just to put it on GitHub?

YES

Okay let's talk.

My company or project only lets me use things in the C++ Standard Library.

Standardization is not a substitute for culture change.



Is there a popular C++ library that has the facility you want to add to the C++ Standard Library?

NO

There's no existing practice; how could we standardize it?

YES

Why isn't that good enough?

My company or project only lets me use things in the C++ Standard Library.

Standardization is not a substitute for culture change.

Managing dependencies is hard.

Have you tried a C++ package manager?

There's a specific reason it would be better in the C++ Standard Library.

Can you live with making no breaking changes to it for the next 20 years?

NO

YES

Is it worth 5x the effort and time that it would take just to put it on GitHub?

NO

YES

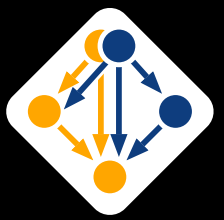
Okay let's talk.

The C++ Standard Library
is not a package manager.

The problem isn't that C++ has no package manager.

The problem is that there are too many.






Can C++ standardize package management?

As C++ committee members, package management may be out of our scope.

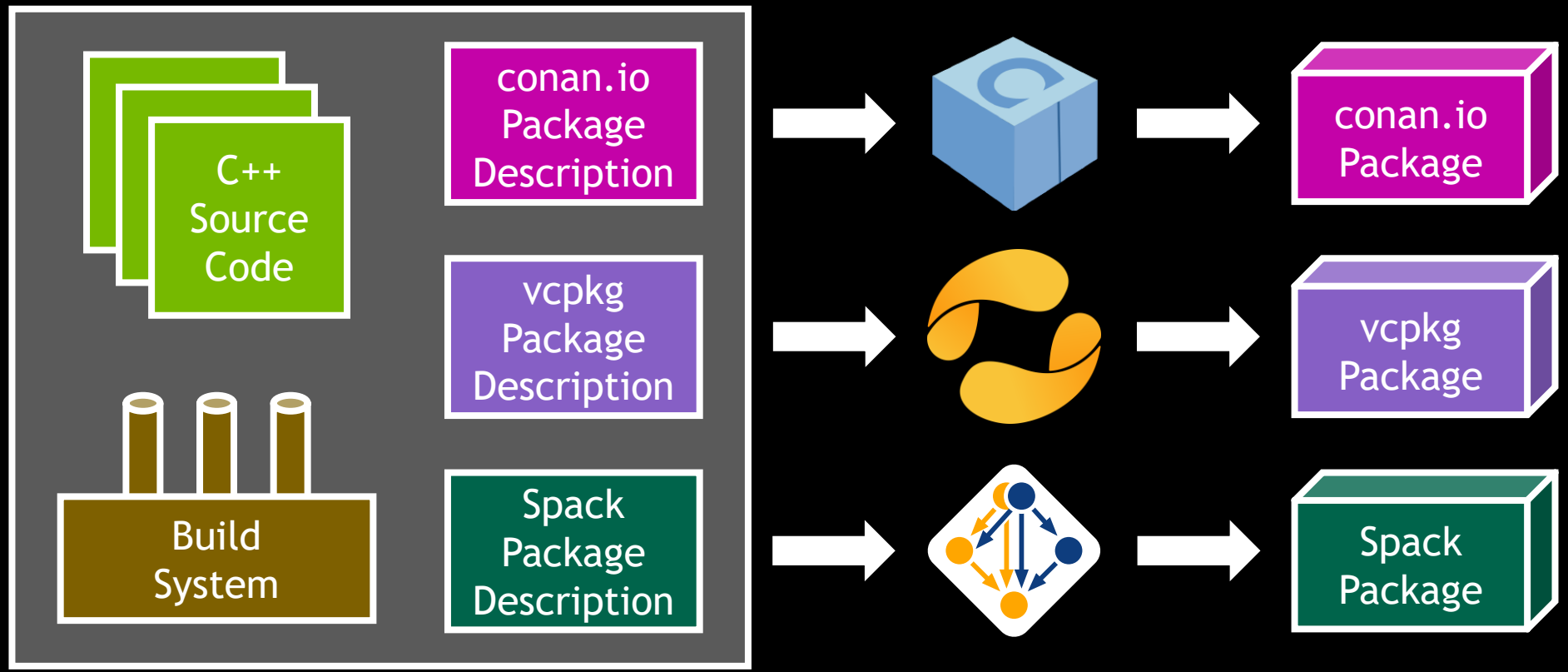
As C++ committee members, package management may be out of our scope.

As leaders of the C++ community, it is our duty and responsibility to act.

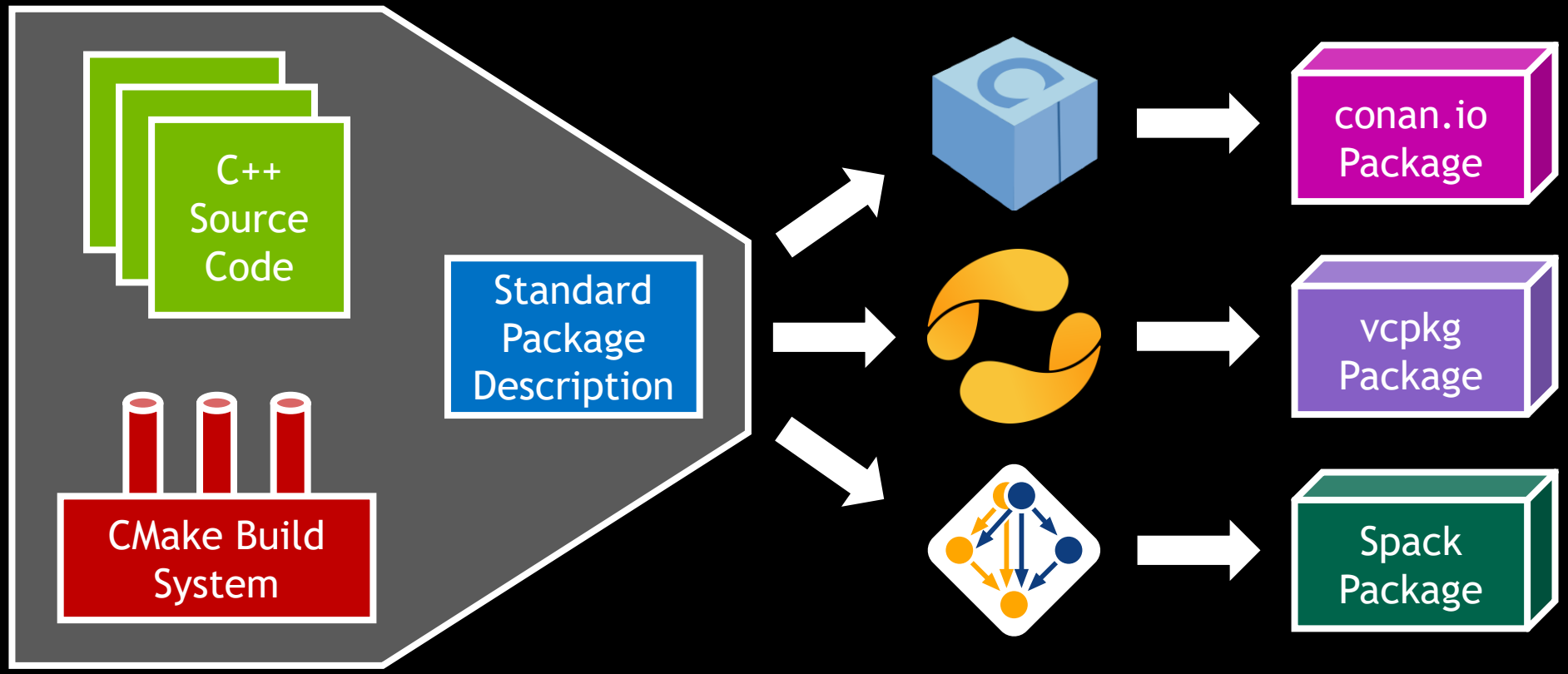
Using external libraries in C++ should be an order of magnitude easier than it is today.

This should be the primary goal for C++ in the next decade.

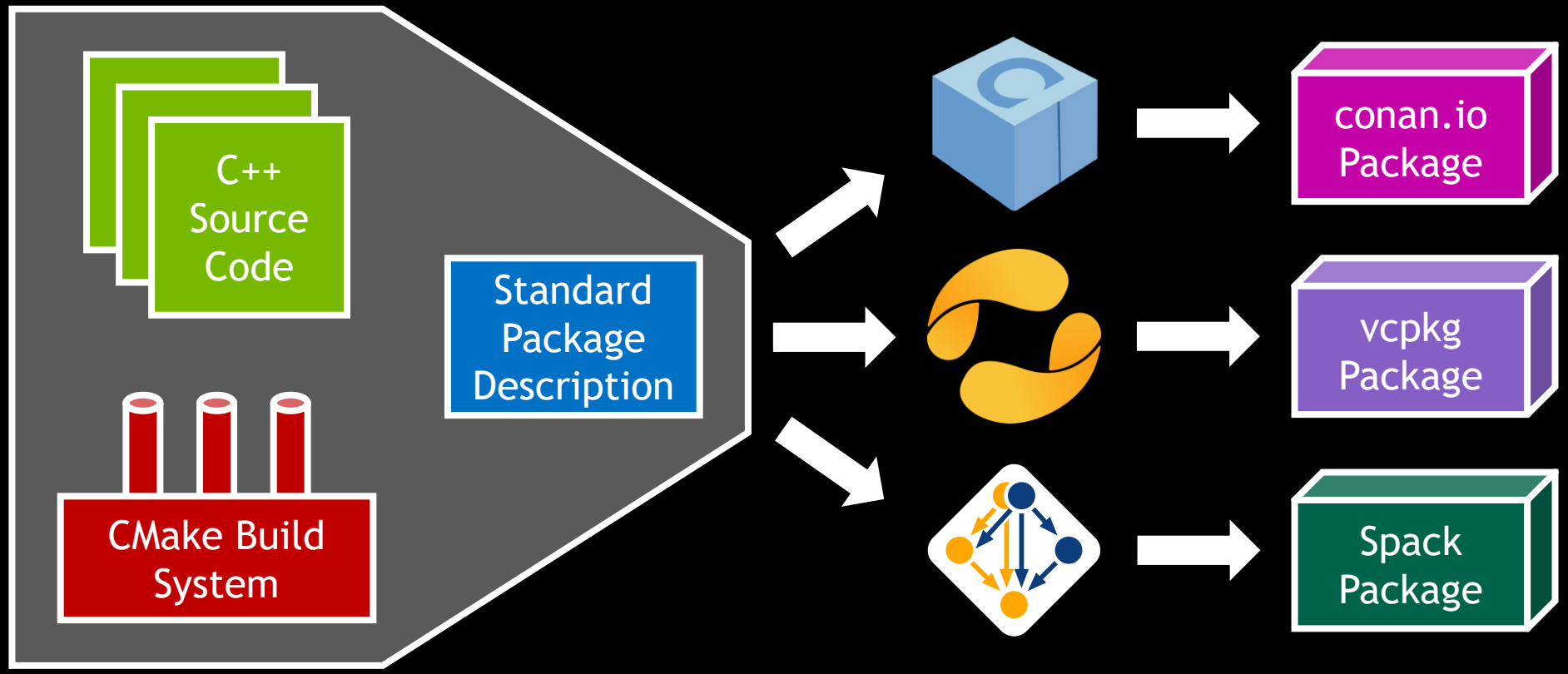
Standardizing C++ Package Management



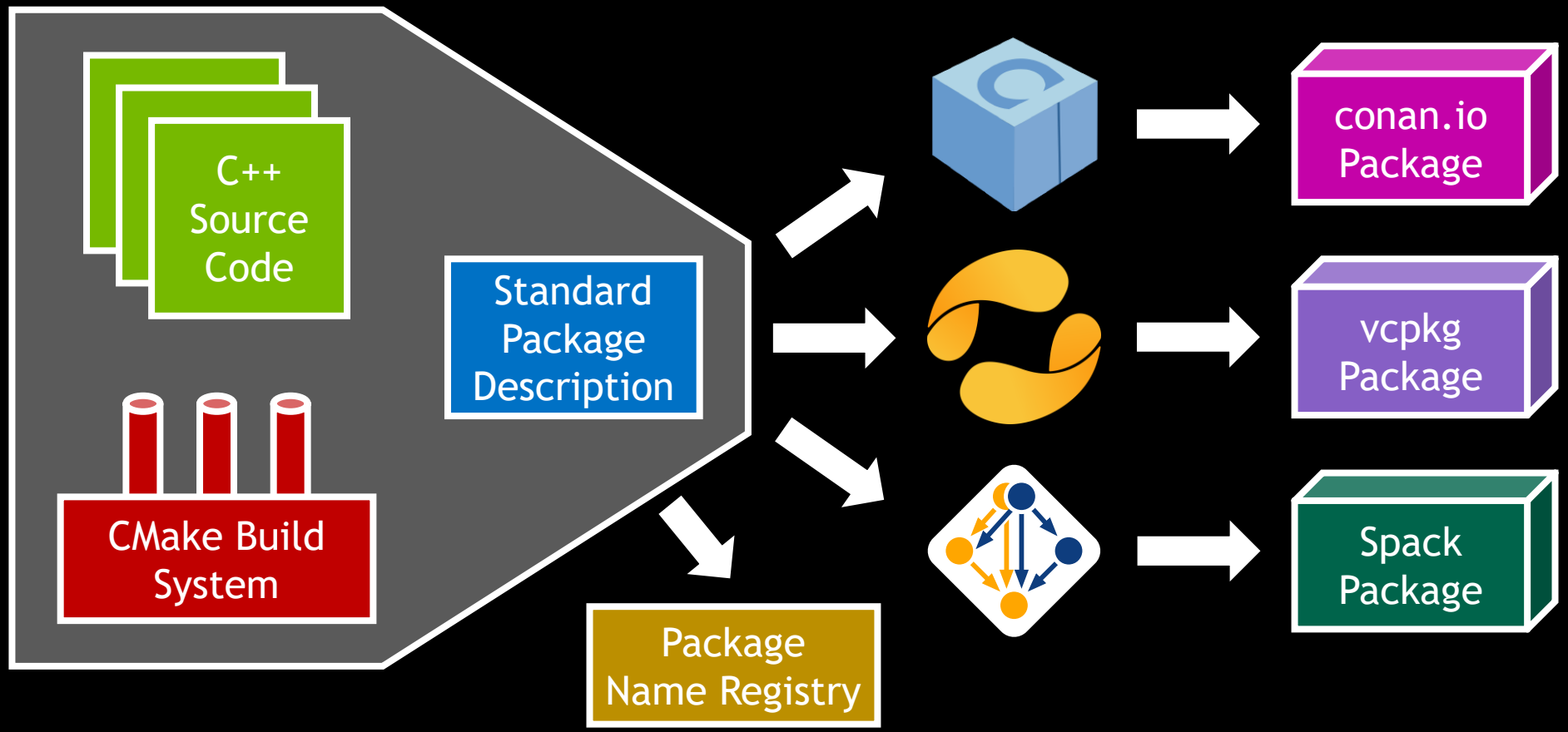
Standardizing C++ Package Management



Standardizing C++ Package Management



Standardizing C++ Package Management



The Necessity Thesis:

The C++ Standard Library should only contain facilities that can't live elsewhere.

- Language Support
- Portability
- Vocabulary

We must find a balance
between Stability and Velocity.

We need new tools to
make that happen.

The Stability Thesis:

Until we learn to change things after we ship them,

the C++ Standard Library should only contain things that are unlikely to need major changes.

Thanks

- Hana Dusíková
- Conor Hoekstra
- Billy O'Neal
- Olivier Giroux
- Michał Dominiak
- Tom Honermann
- Corentin Jabot
- Matt Kulukundis
- Casey Carter
- Matt Calabrese

WHAT BELONGS IN THE C++ STANDARD LIBRARY?

Bryce Adelstein Lelbach
@blelbach

NVIDIA. HPC Programming Models Architect
Standard C++ Library Evolution Chair, US Programming Languages Chair

{C++;} CppIndia