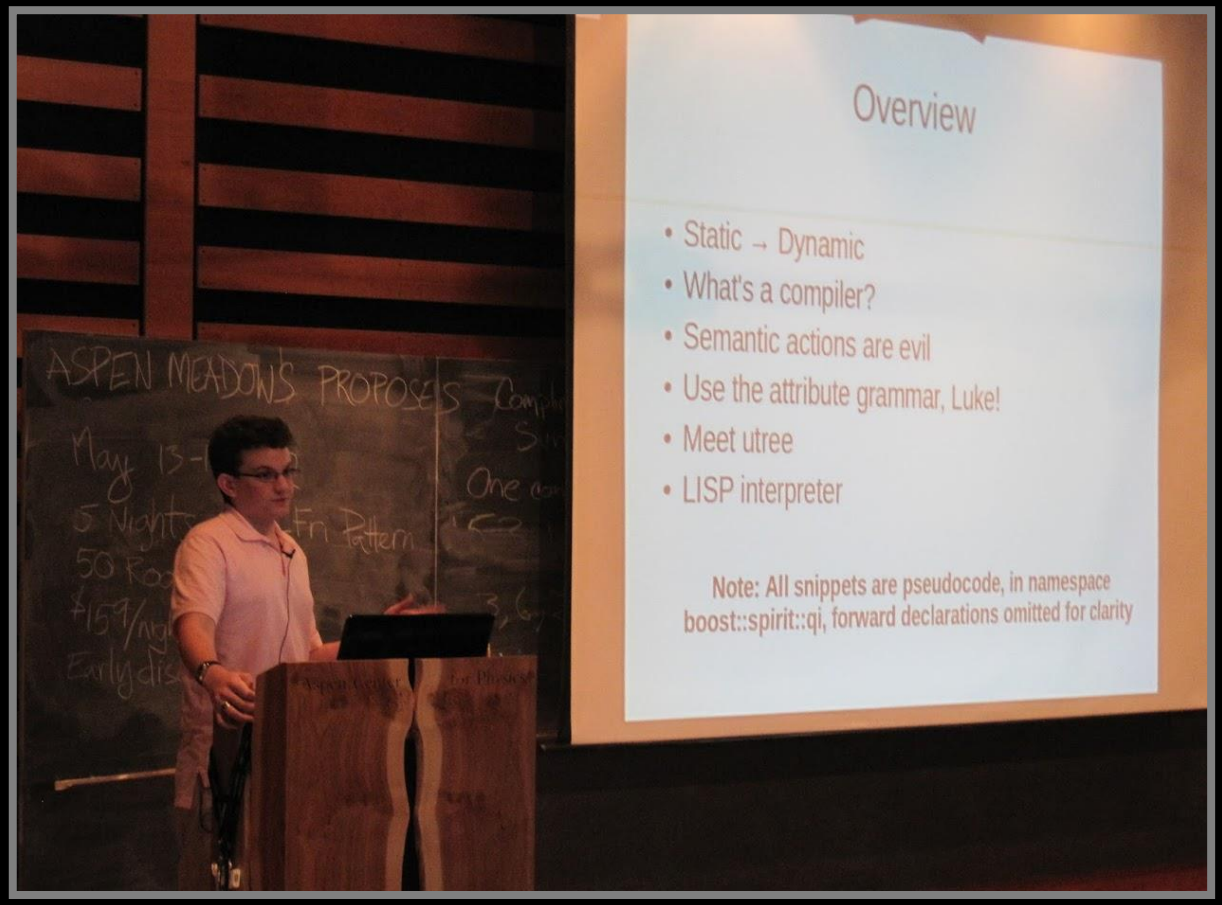
WHAT BELONGS IN THE C++ STANDARD LIBRARY?

Bryce Adelstein Lelbach
@blelbach

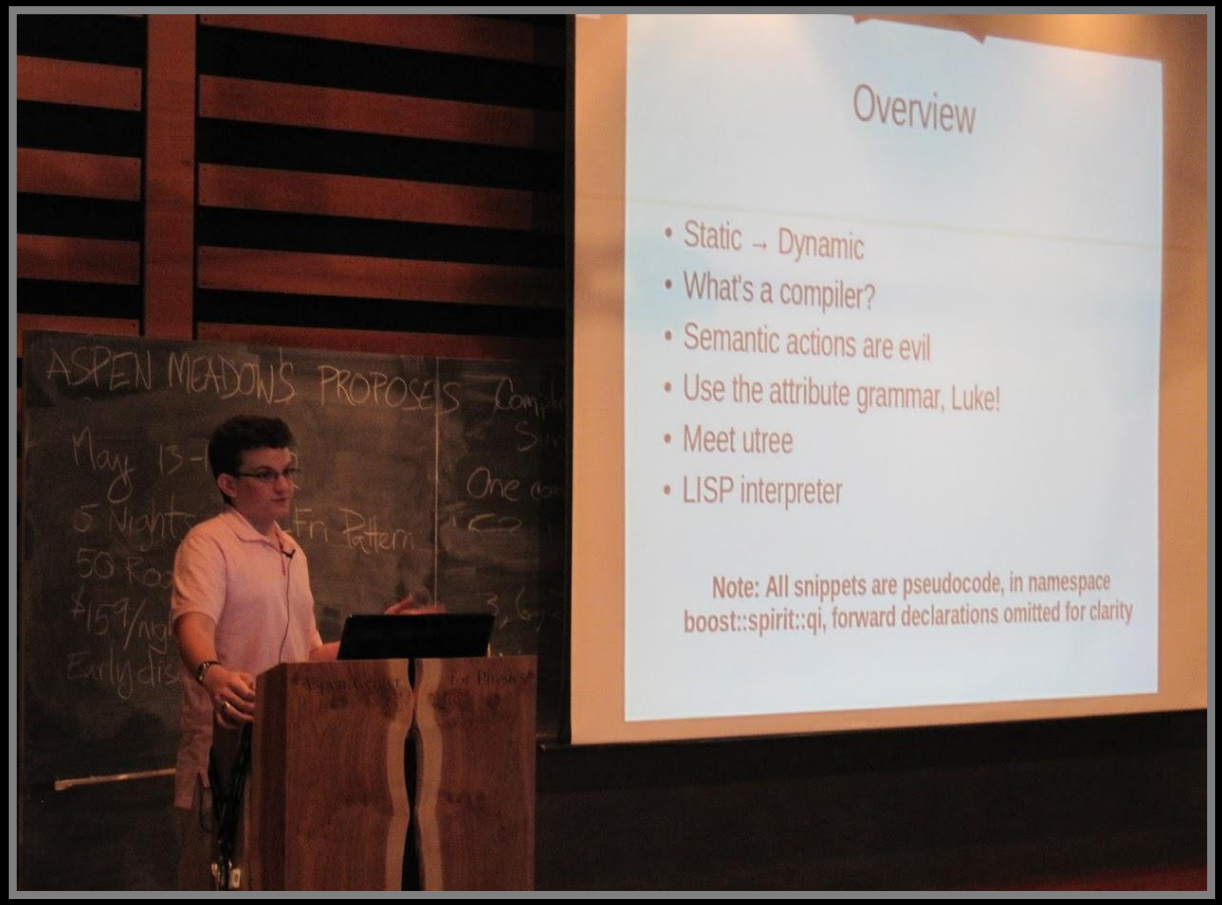
NVIDIA. HPC Programming Models Architect
Standard C++ Library Evolution Chair, US Programming Languages Chair

C++ now

BoostCon 2011



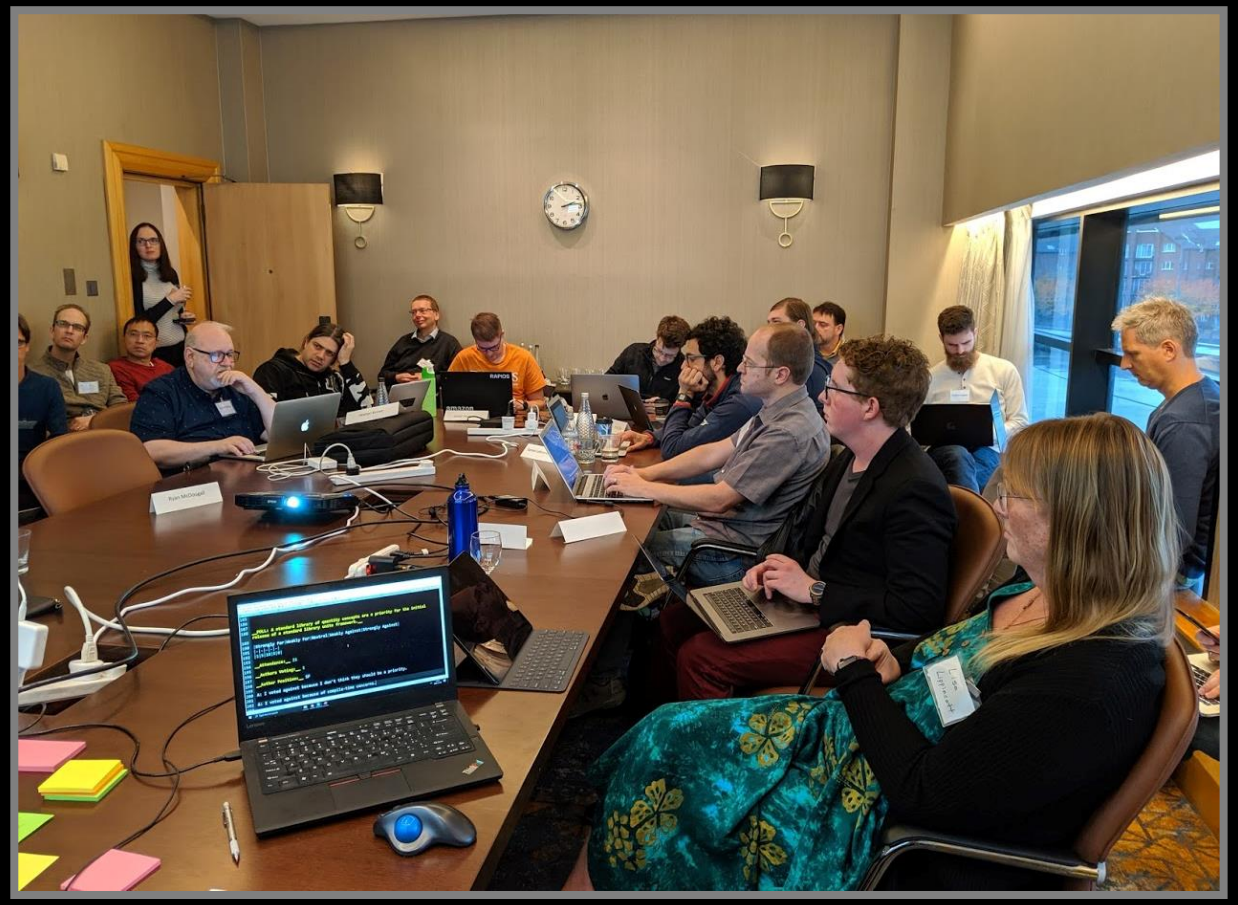
BoostCon 2011



C++Now Student/Volunteer Program

#include <C++>

The Standard C++ Committee

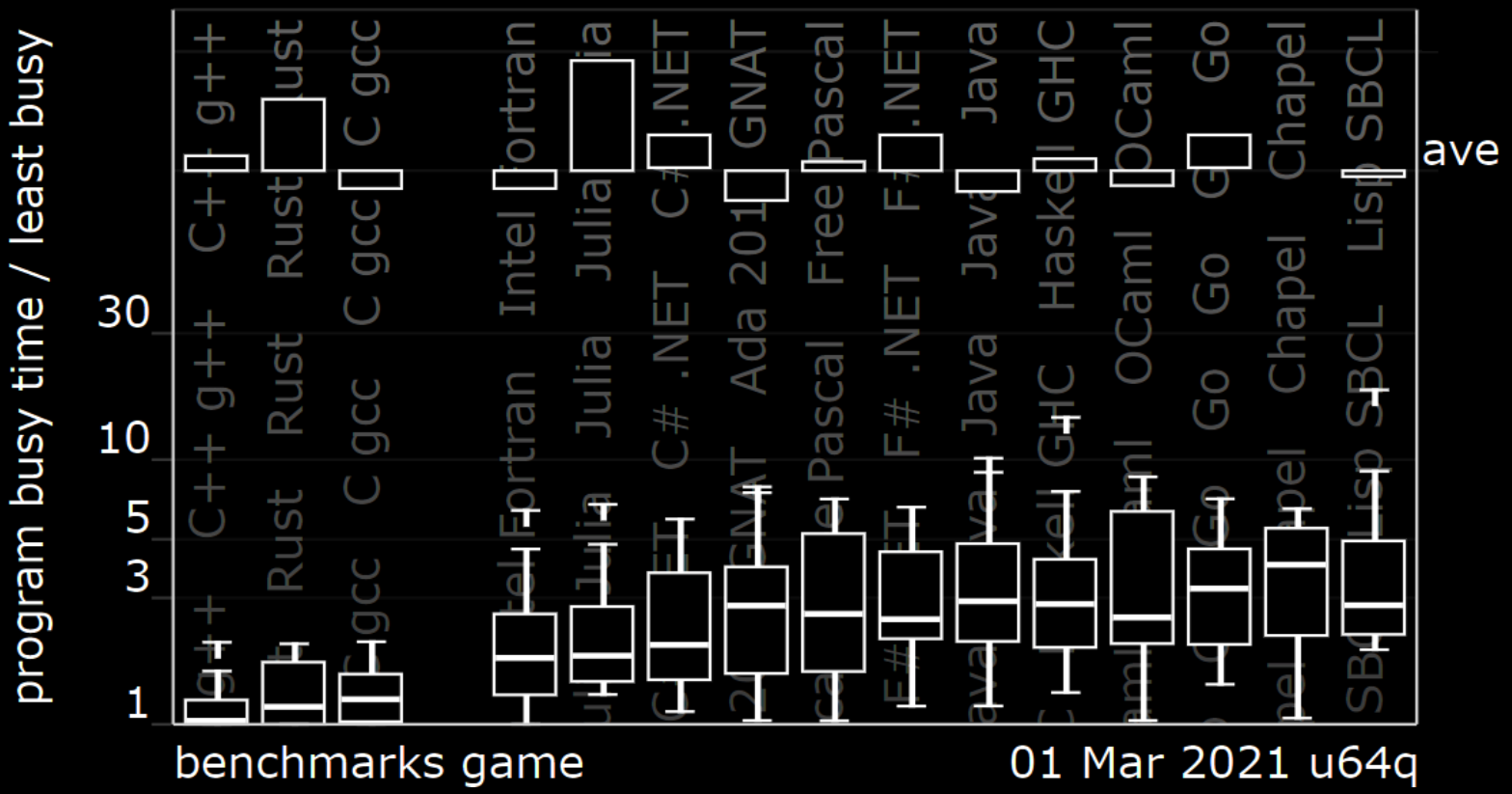


What Has Made C++ Successful?

What Has Made C++ Successful?

Performance?

What Has Made C++ Successful?
Performance?



Source: The Computer Language Benchmark Game

#include <C++>

What Has Made C++ Successful?

Portability?

What Has Made C++ Successful?

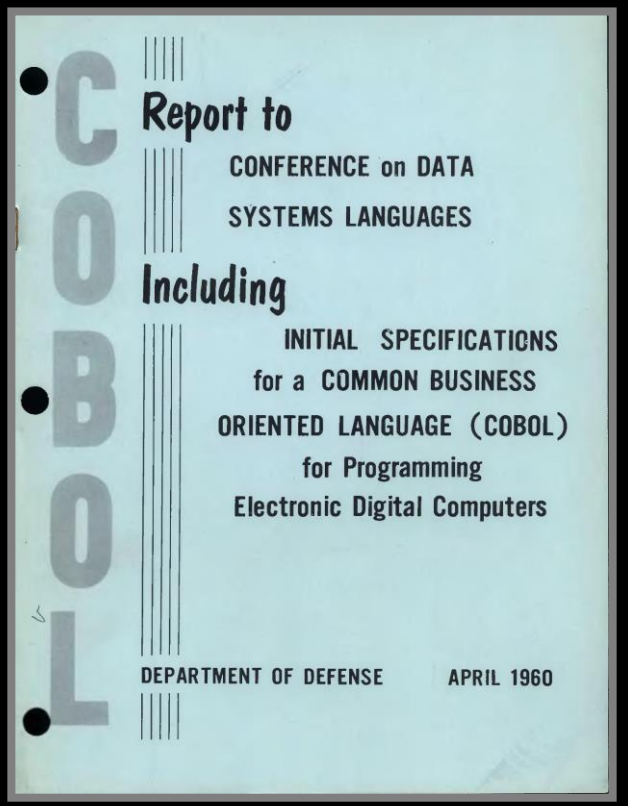
Portability?

What Has Made C++ Successful?

Stability?

What Has Made C++ Successful?

Stability?

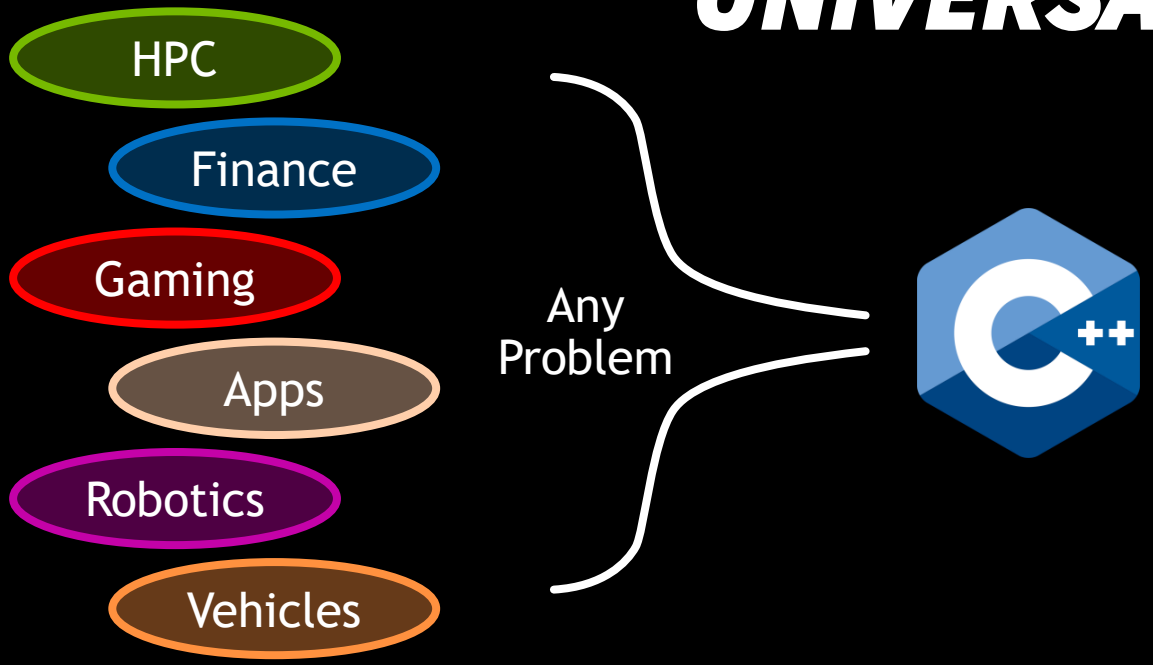


What Has Made C++ Successful?

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

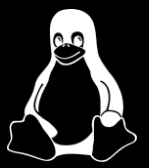
Any Problem

Any Paradigm

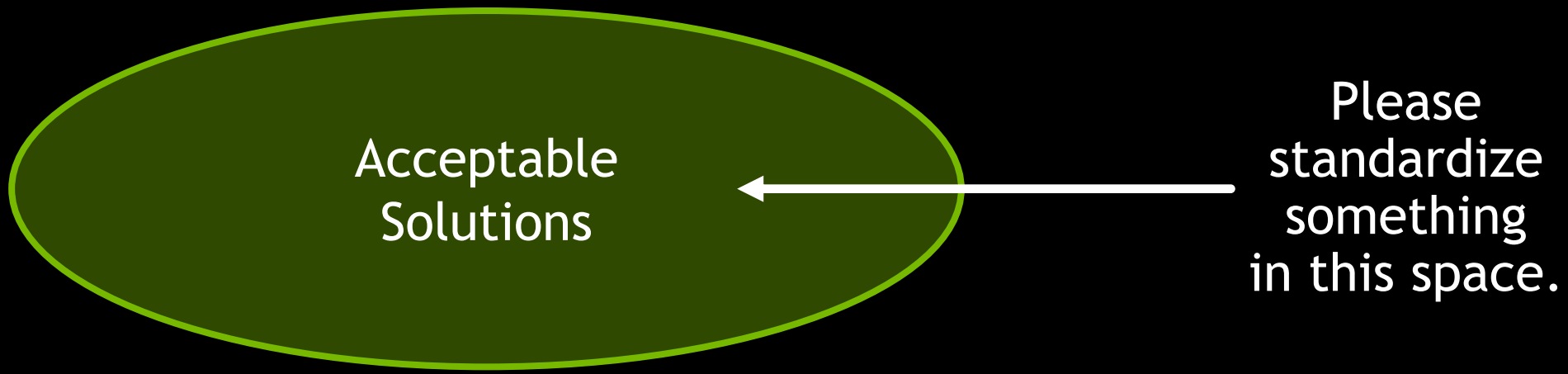
- Imperative
- Object Oriented
- Functional
- Generic
- Parallel
- Reactive

Any Platform

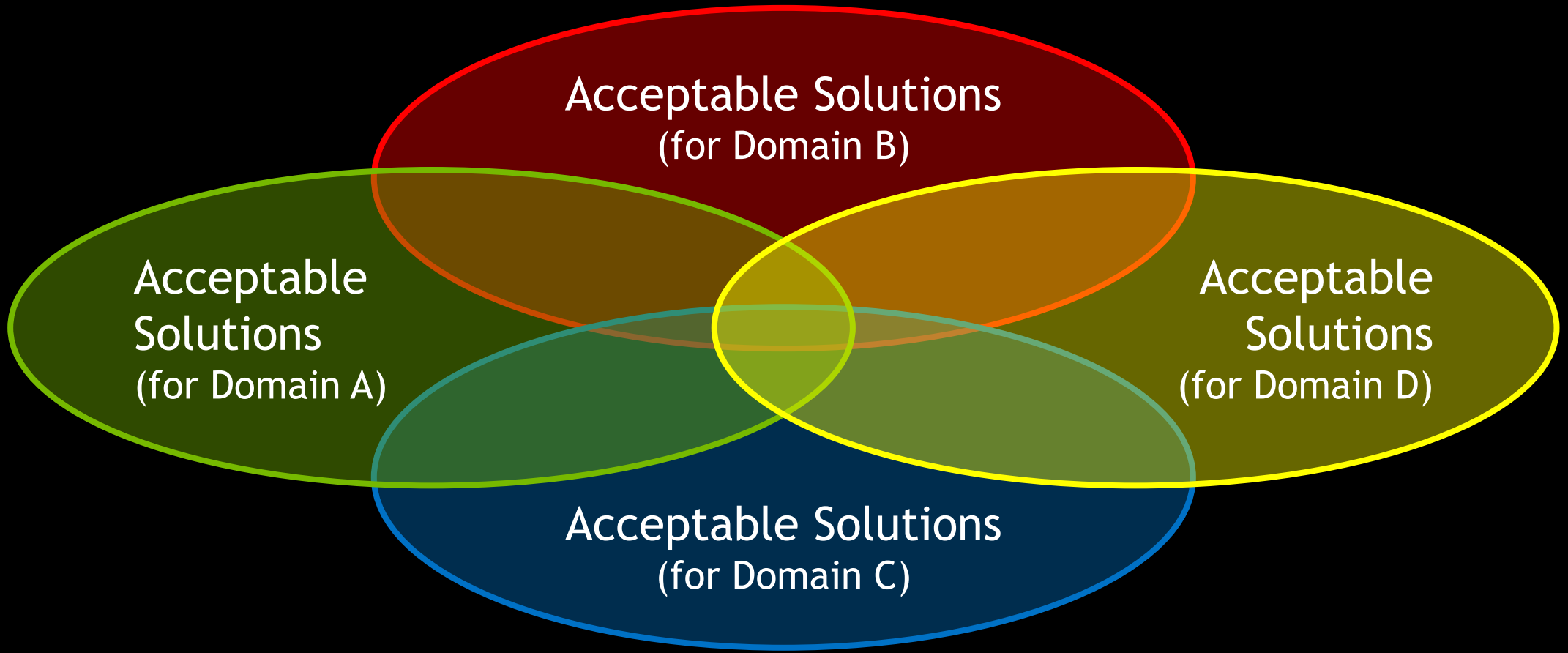




Our Individual Perception



Reality

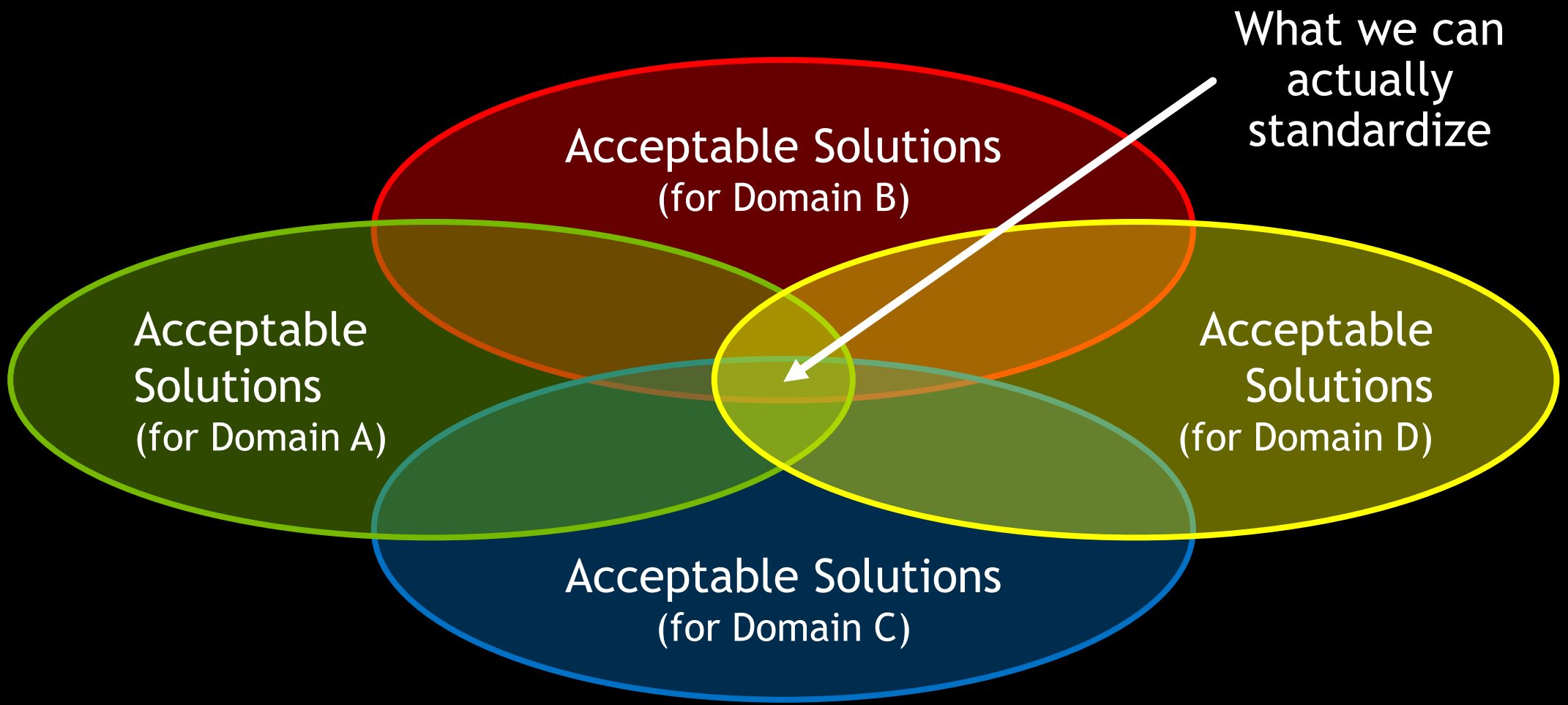


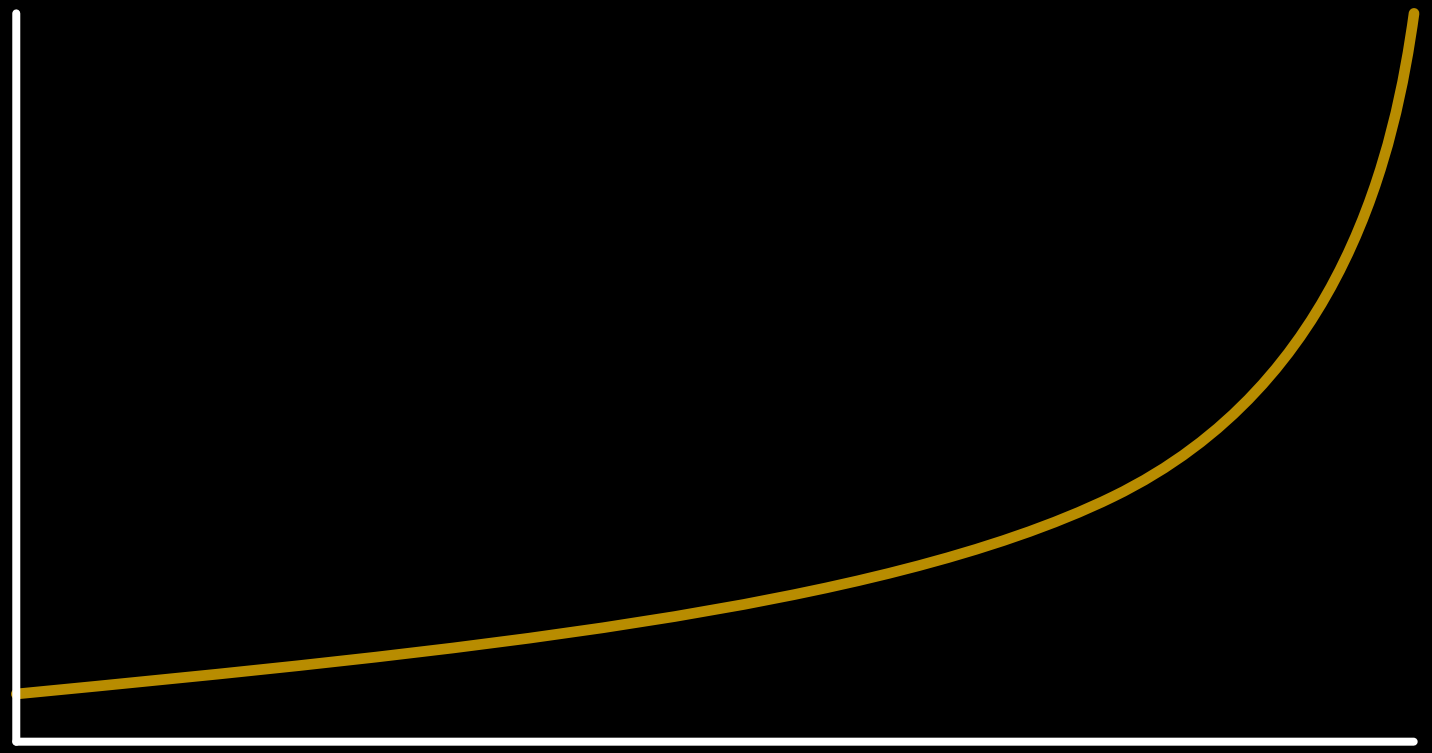
Use Case Sympathy (noun)

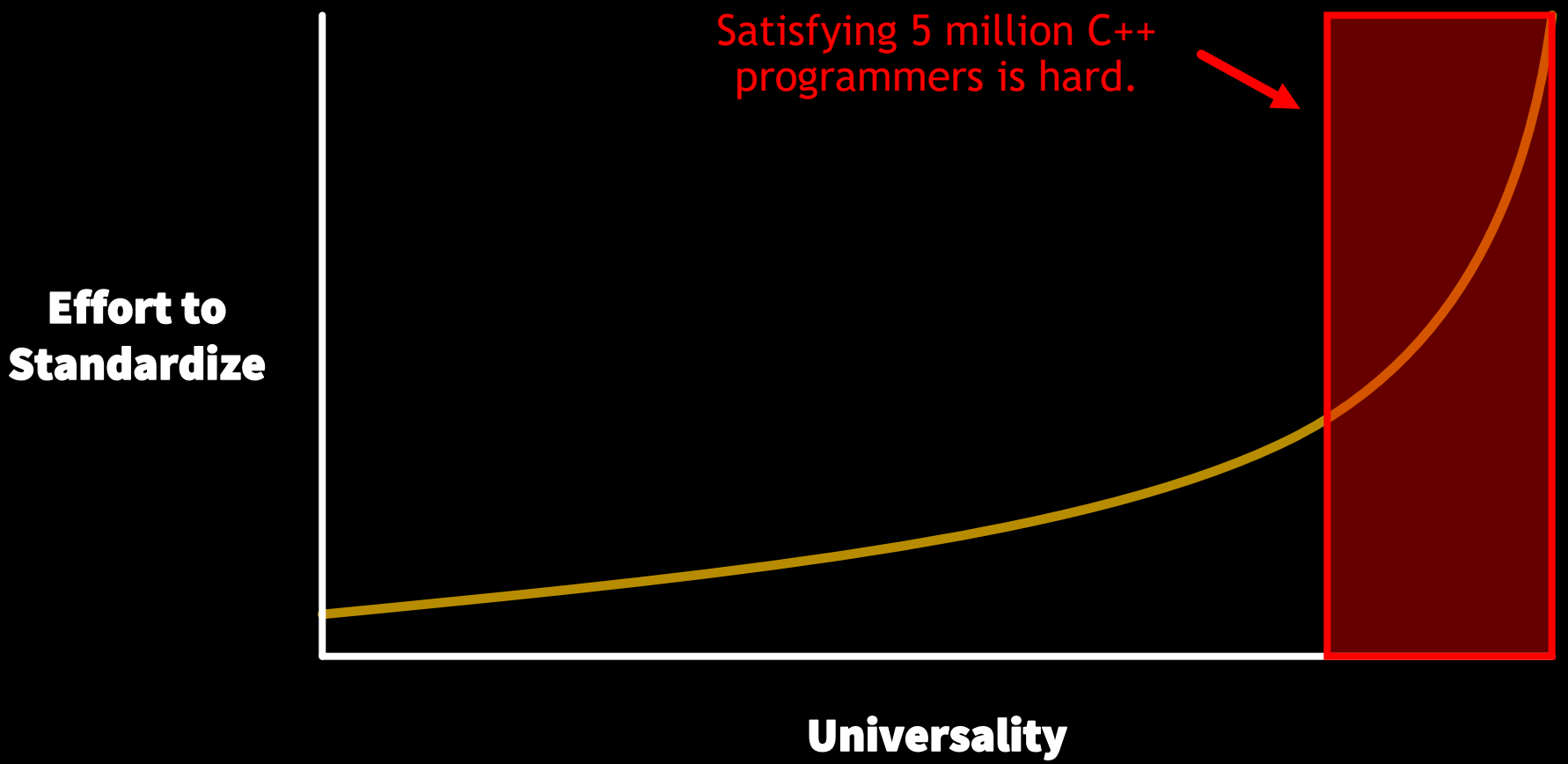
use case sym·pa·thy | yüz kās sim-pə-thē

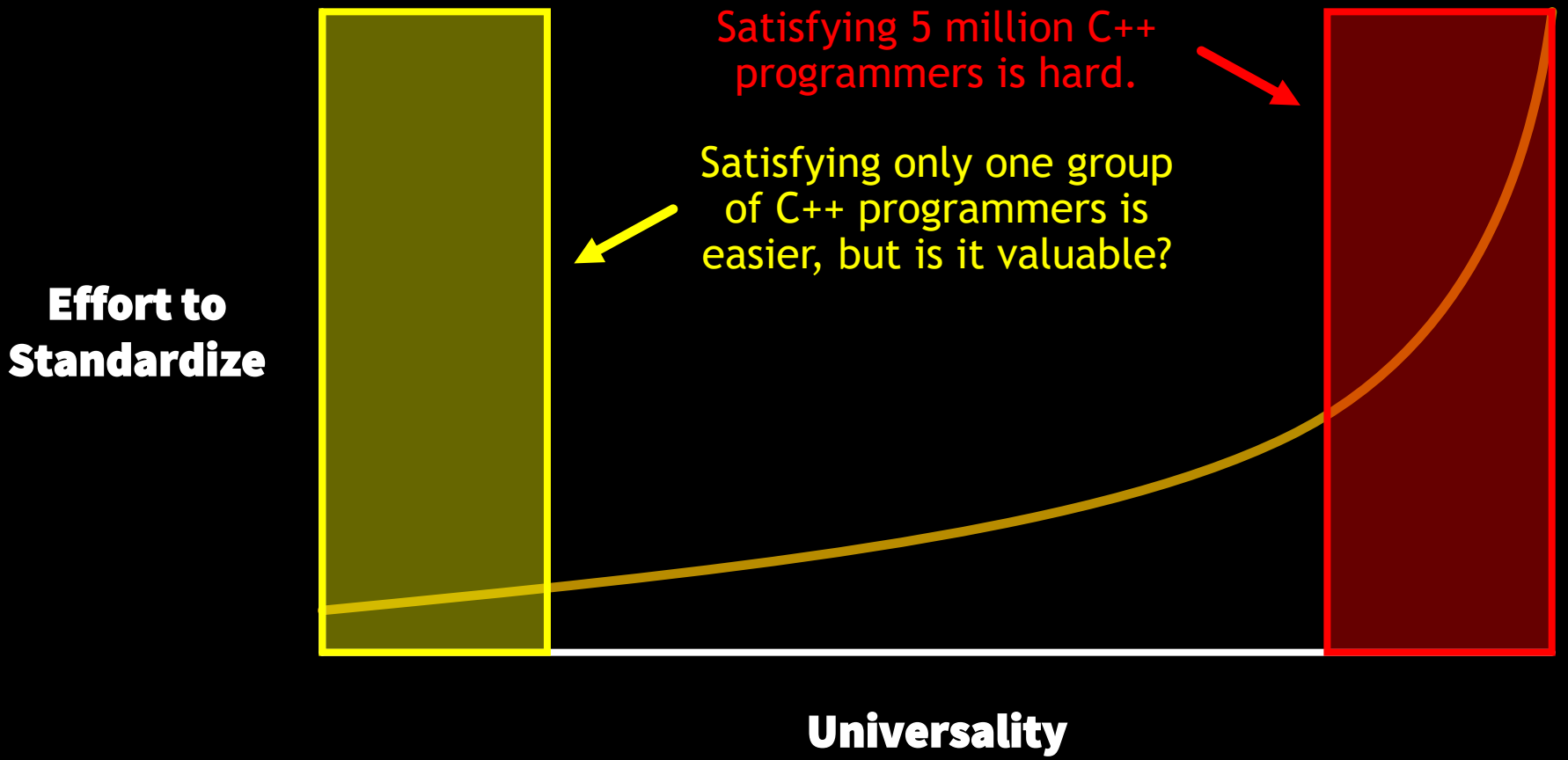
Accepting the importance and validity of use cases that you are not personally familiar with or believe in.

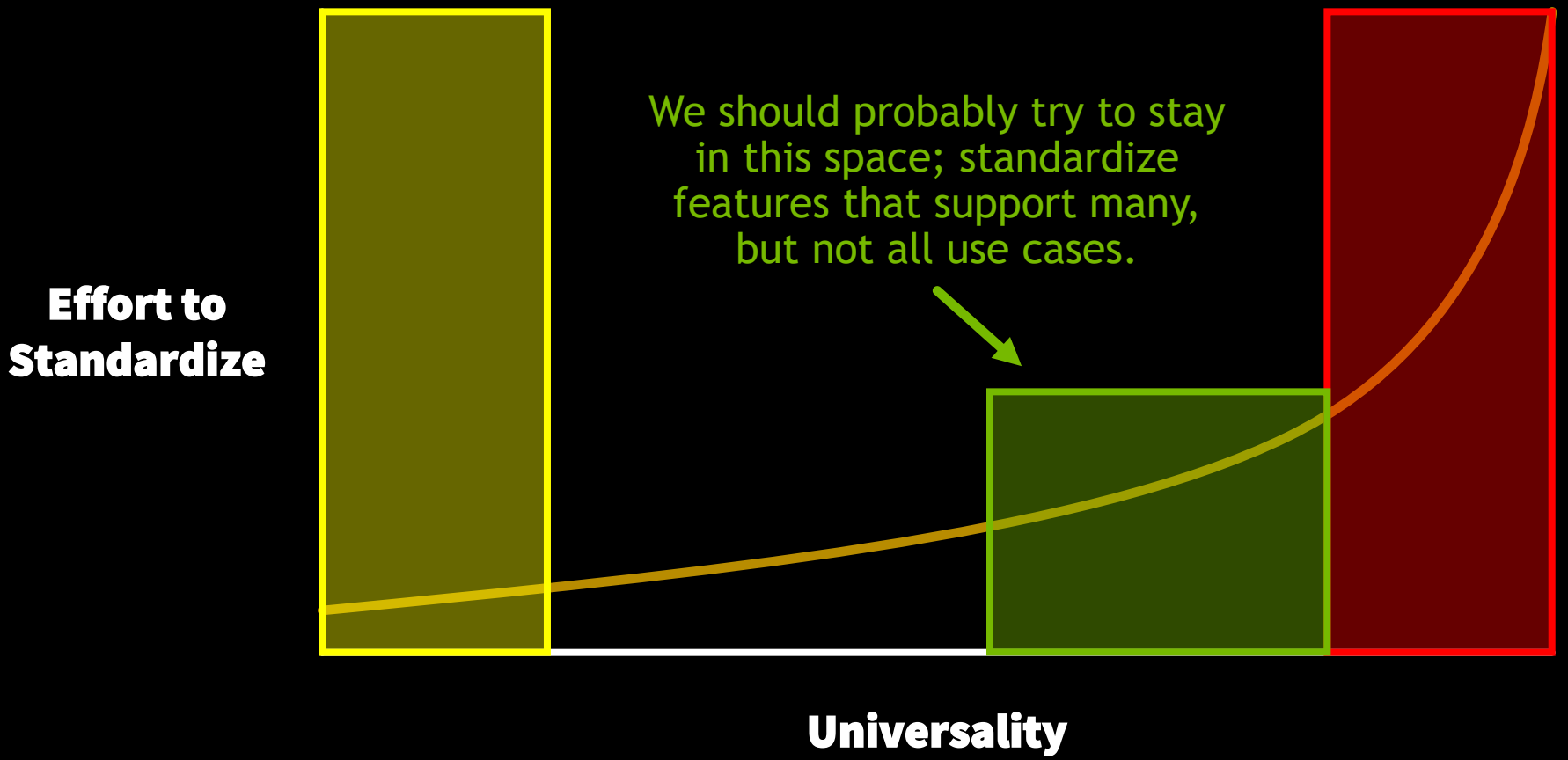
Reality







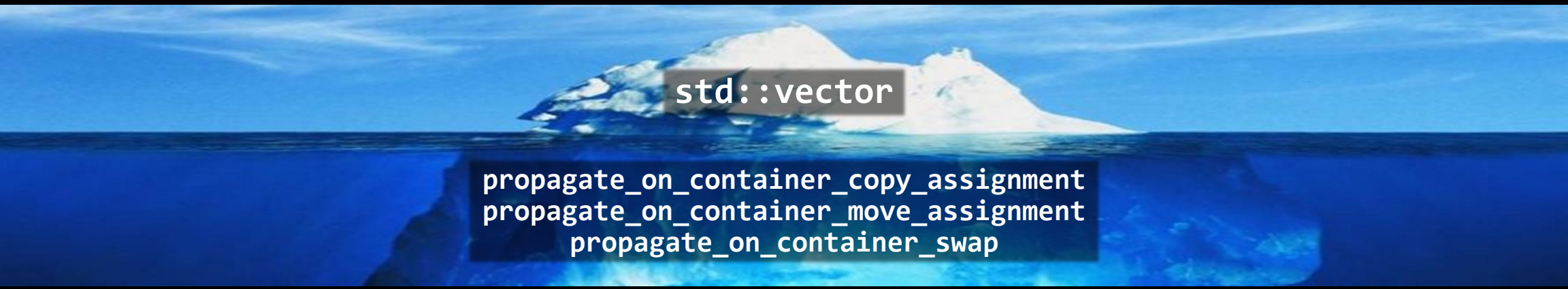




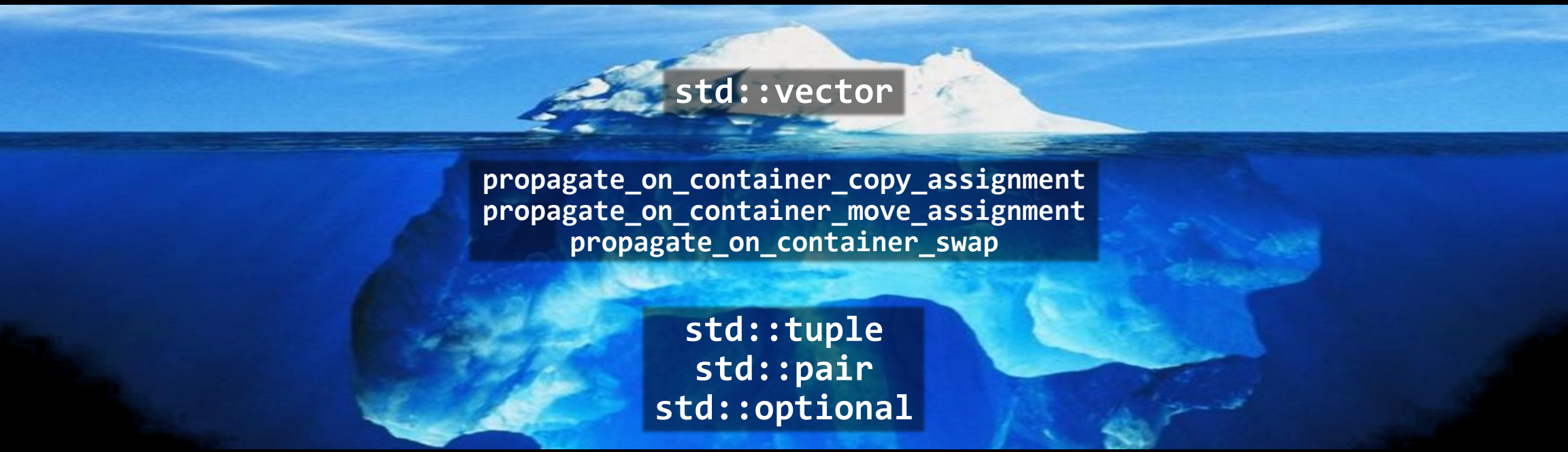
Should Everything in `std::` Support Allocators?

Should Everything in `std::` Support Allocators?

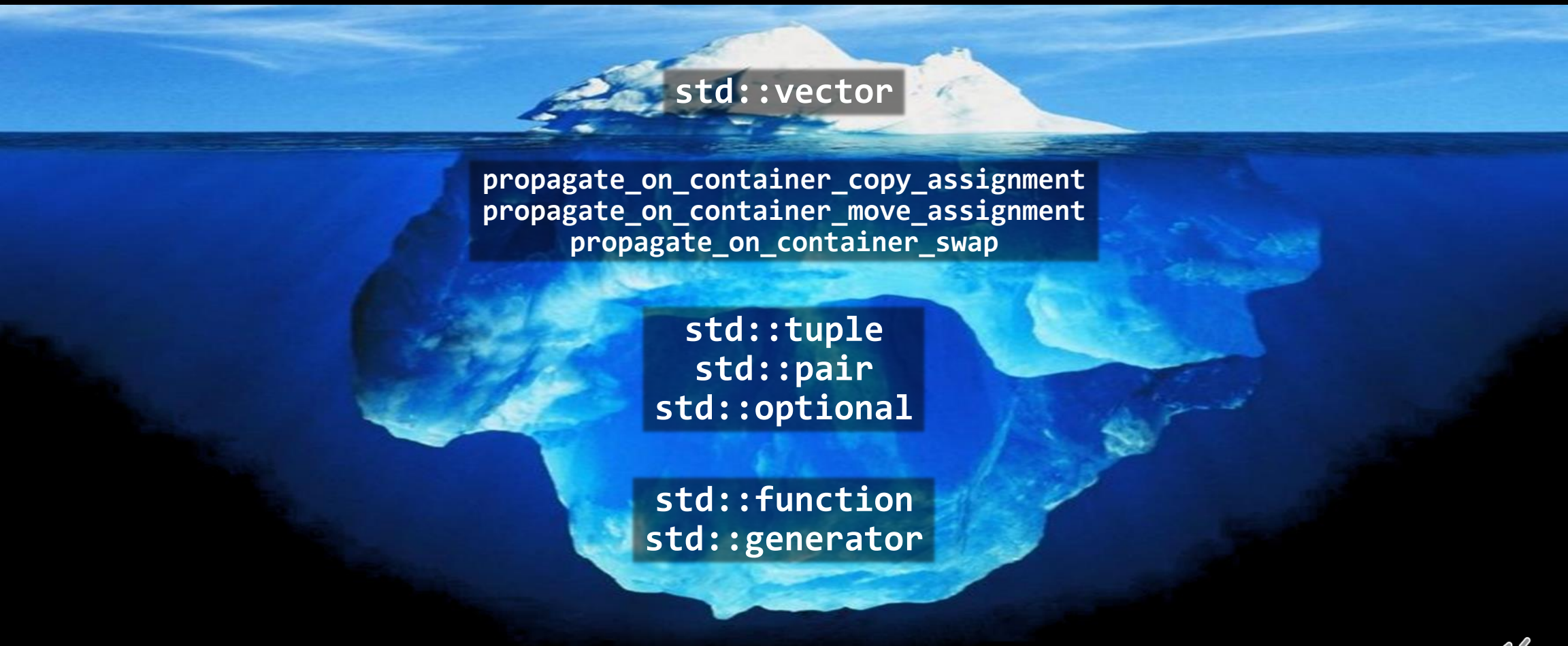
Should Everything in `std::` Support Allocators?



Should Everything in `std::` Support Allocators?



Should Everything in `std::` Support Allocators?



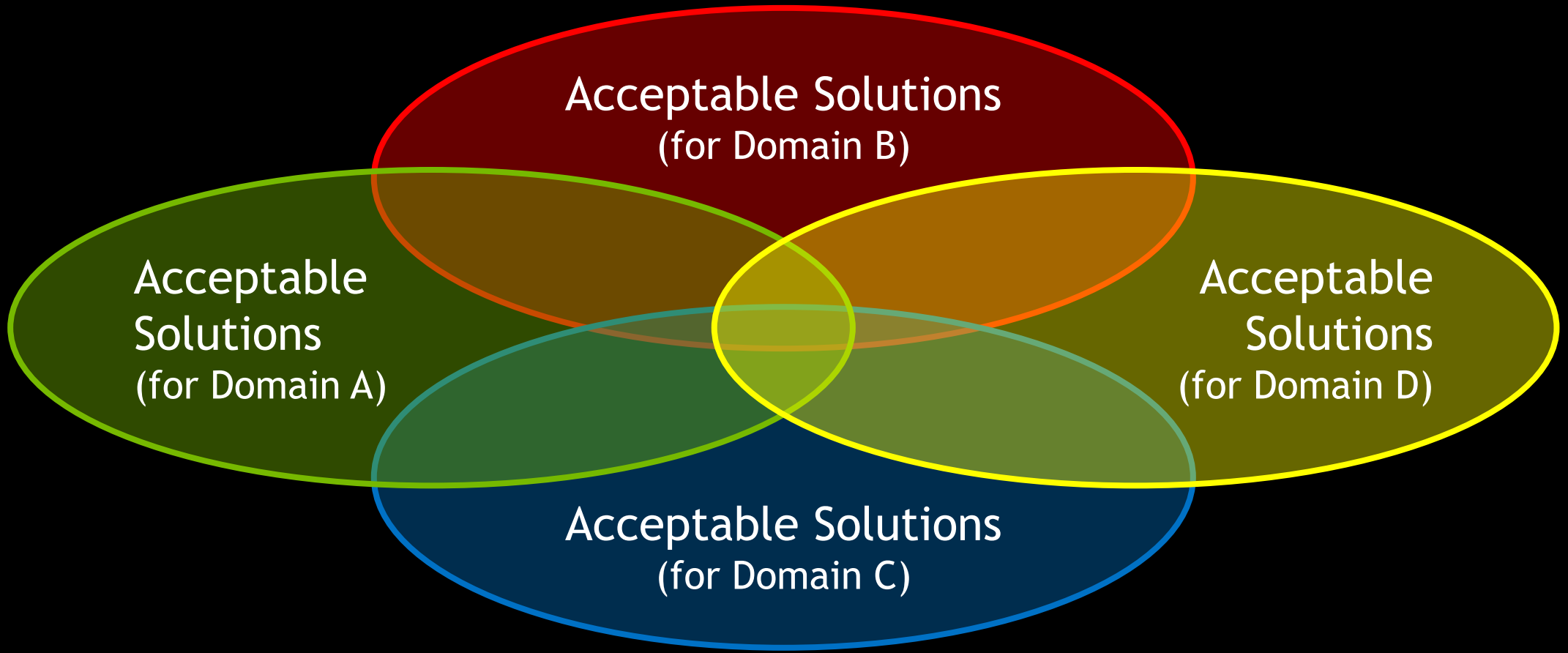
Should Everything in `std::` Have A Type Erased Form?

```
template <typename R>
  requires std::ranges::random_access_range<R>
void
my_algorithm(R&& r)
{
  // ...
}
```

Should Everything in `std::` Have A Type Erased Form?

```
void
my_algorithm(std::any_random_access_range<int> r)
{
  // ...
}
```

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

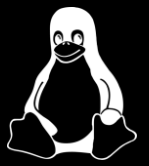
Functional

Generic

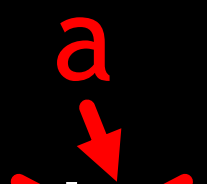
Parallel

Reactive

Any Platform



What is the C++ Standard Library?



What is ~~the~~ C++ Standard Library?

implementation

GCC’s libstdc++ is not **the** C++ Standard Library.

MSVC’s STL is not **the** C++ Standard Library.

LLVM’s libc++ is not **the** C++ Standard Library.

NVIDIA’s libnv++ is not **the** C++ Standard Library.

GCC’s libstdc++ is not **the** C++ Standard Library.

MSVC’s STL is not **the** C++ Standard Library.

LLVM’s libc++ is not **the** C++ Standard Library.

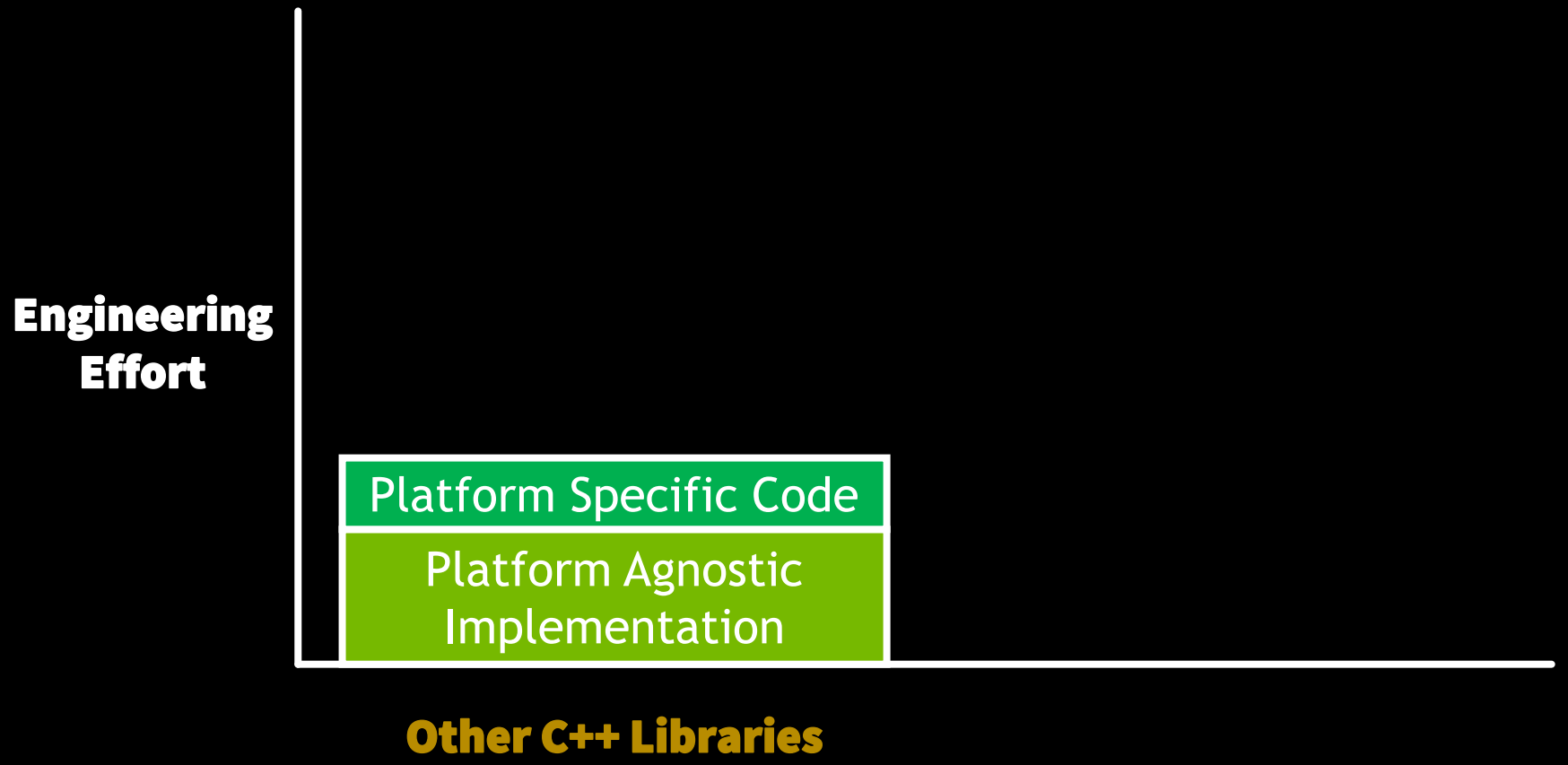
NVIDIA’s libnv++ is not **the** C++ Standard Library.

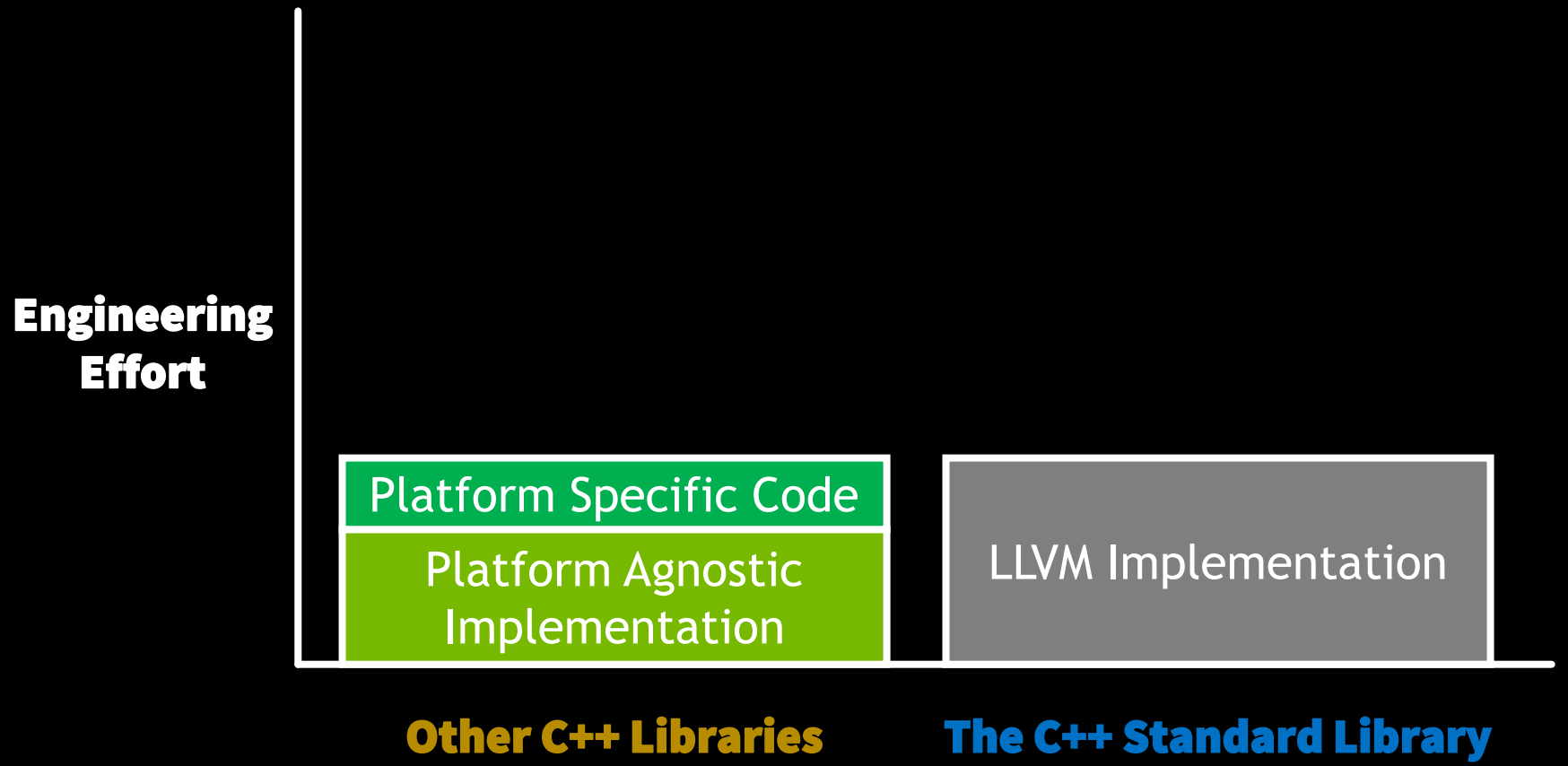
They are C++ Standard Library implementations.

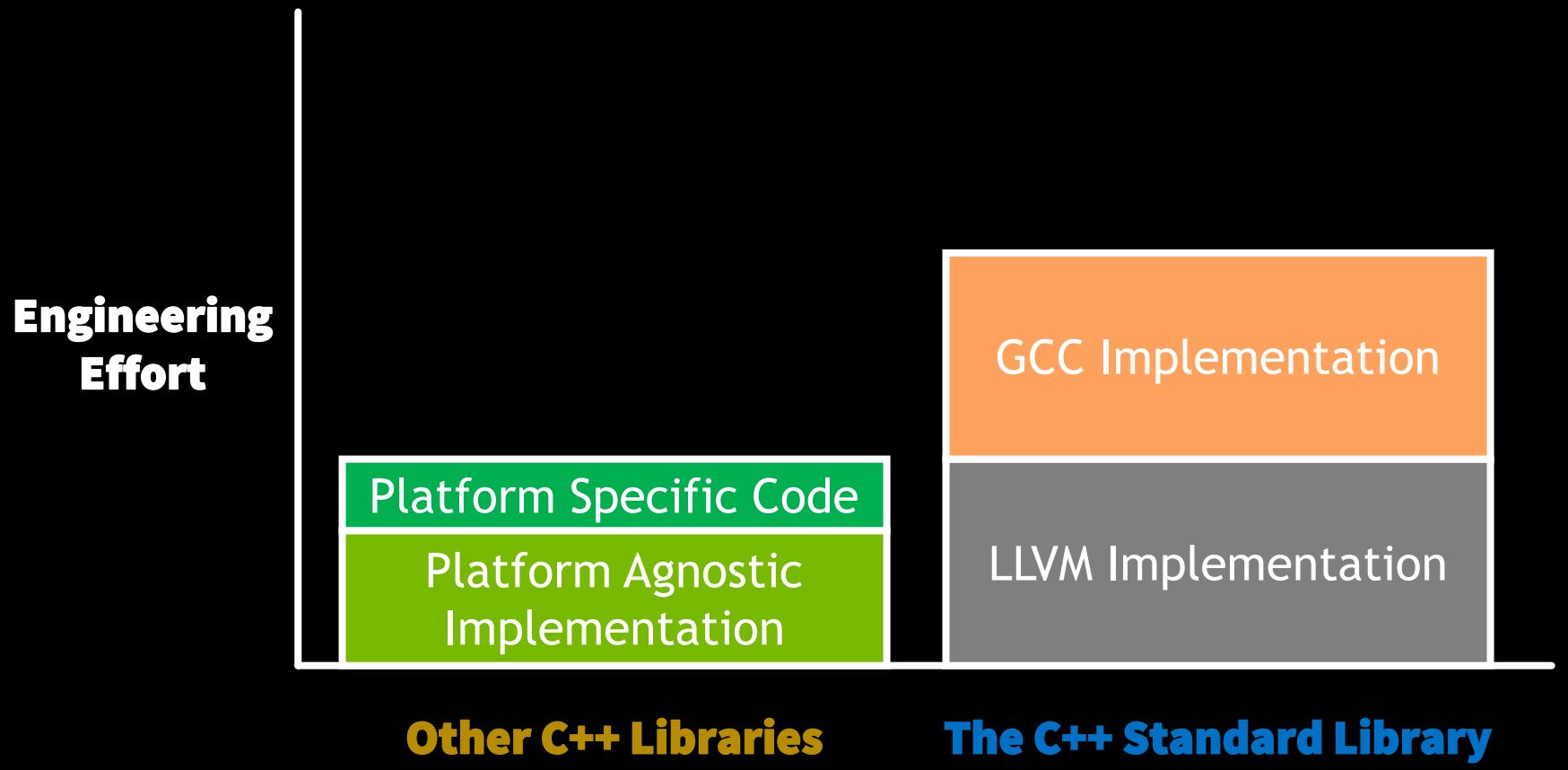
The C++ Standard Library is a *specification*.

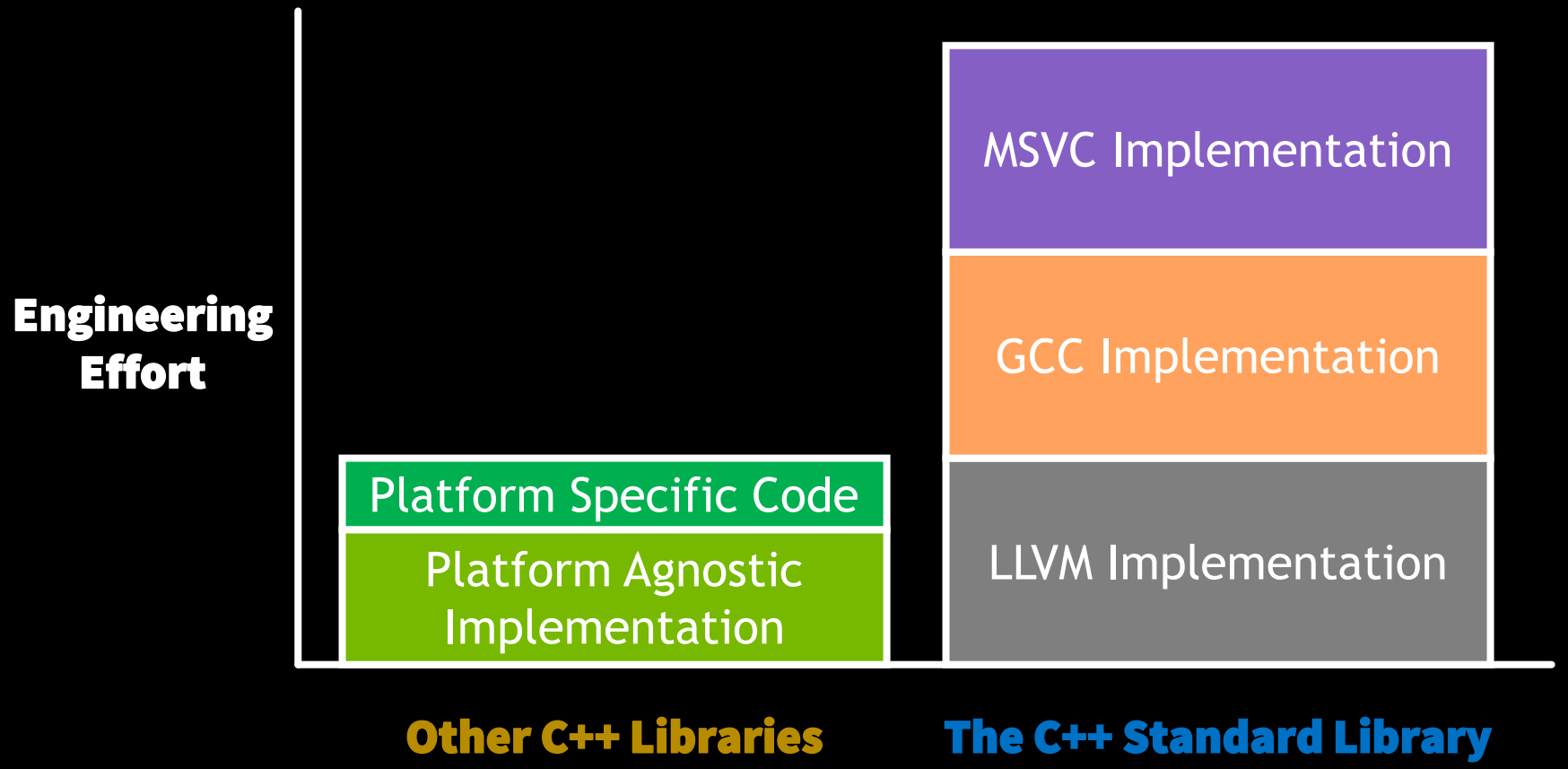
The C++ Standard Library is a *specification*.

That makes it an inefficient vehicle for delivering features.









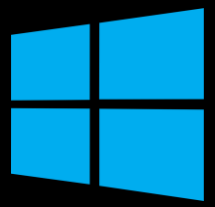
The C++ Standard is
descriptive, not ***prescriptive***.

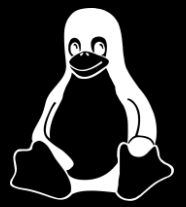
Domain A

Domain B

Domain C

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





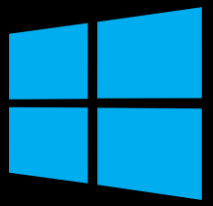
Domain A

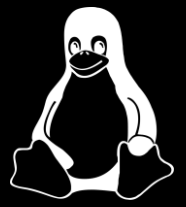
Domain B

Domain C

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

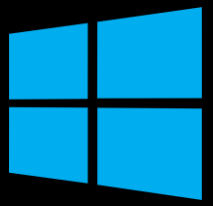
Domain A

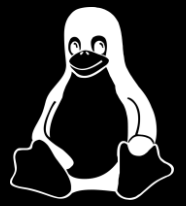
Domain B

Domain C

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.	

`std::vector::operator[]`

`std::string::operator[]`

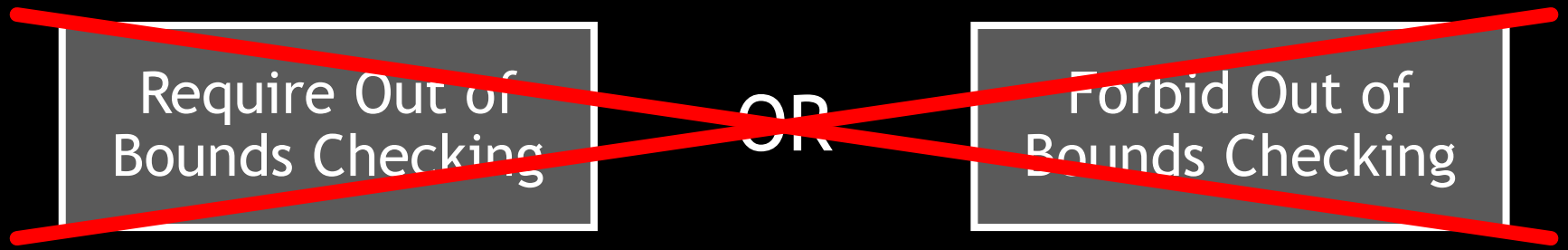
Require Out of Bounds Checking

OR

Forbid Out of Bounds Checking

`std::vector::operator[]`

`std::string::operator[]`

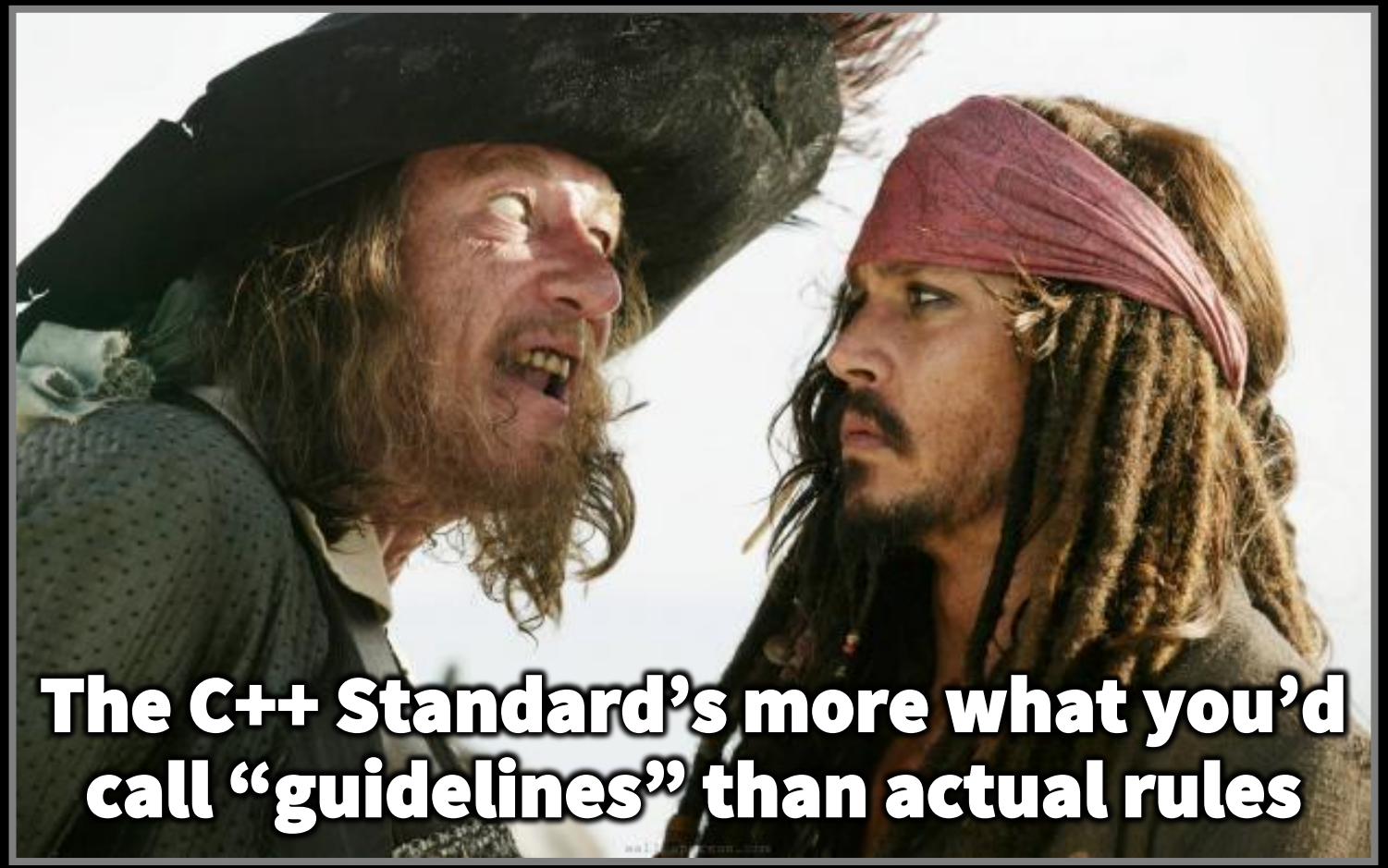


`std::vector::operator[]`

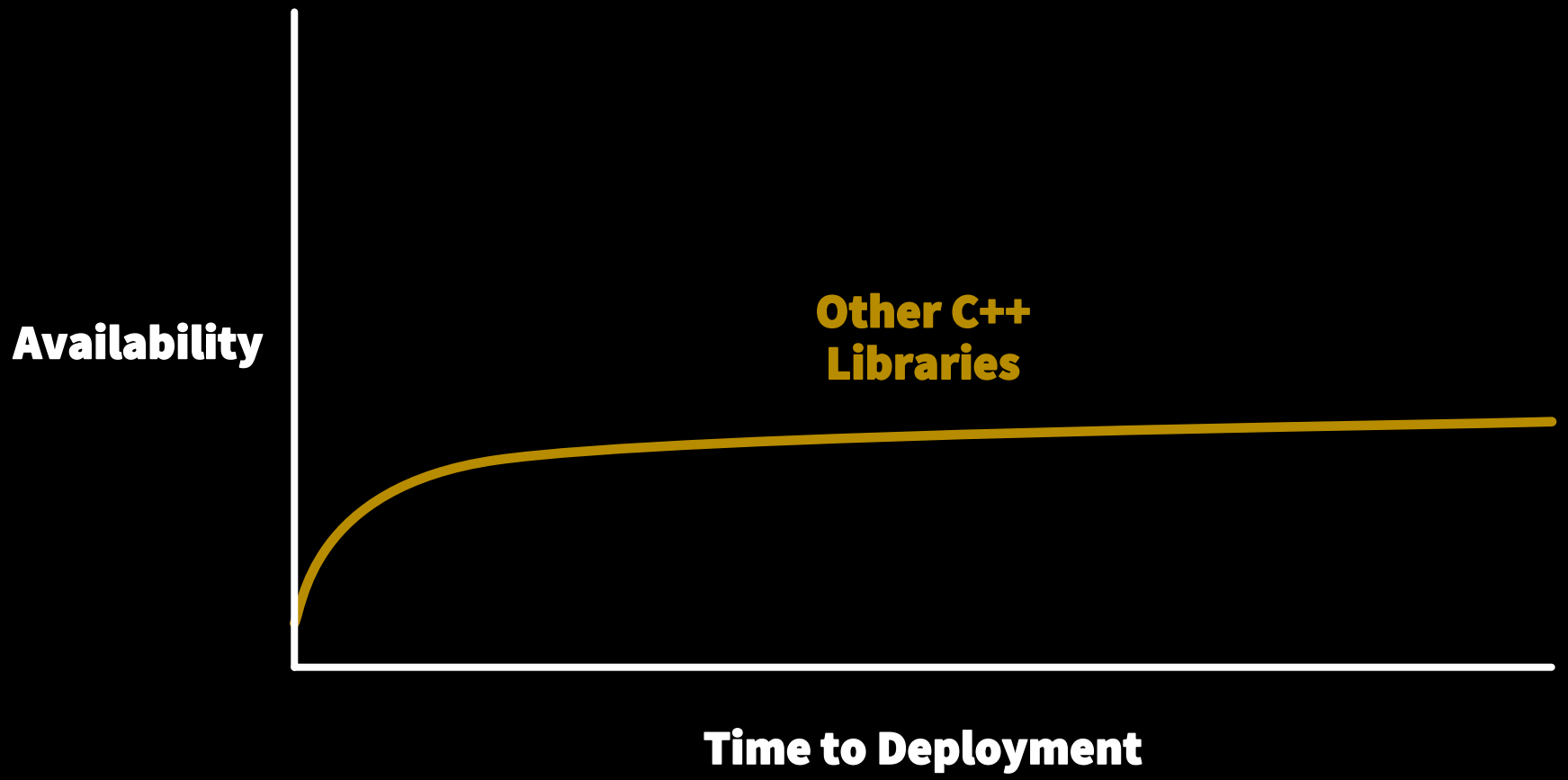
`std::string::operator[]`

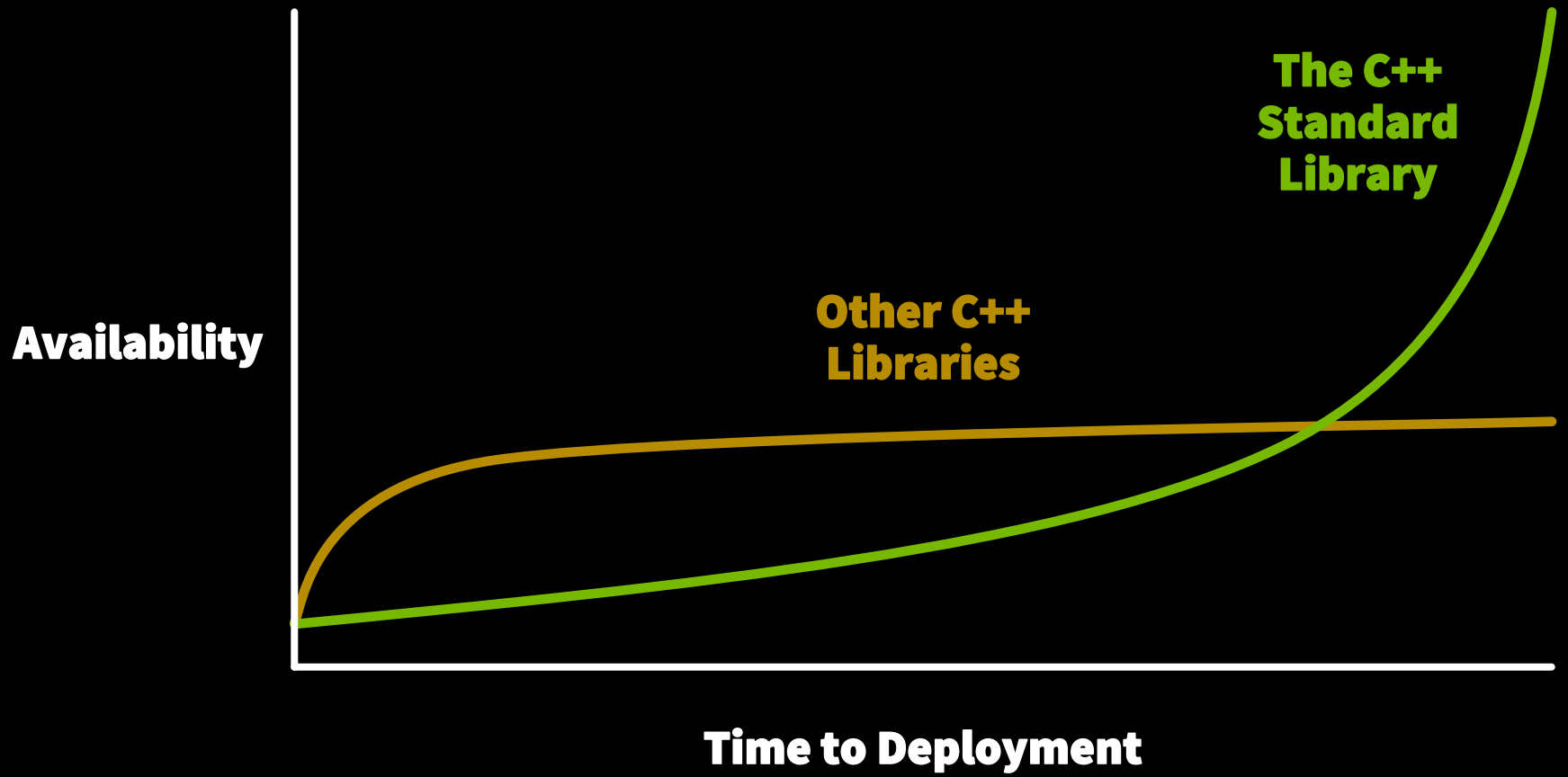
Permit Out of Bounds Checking

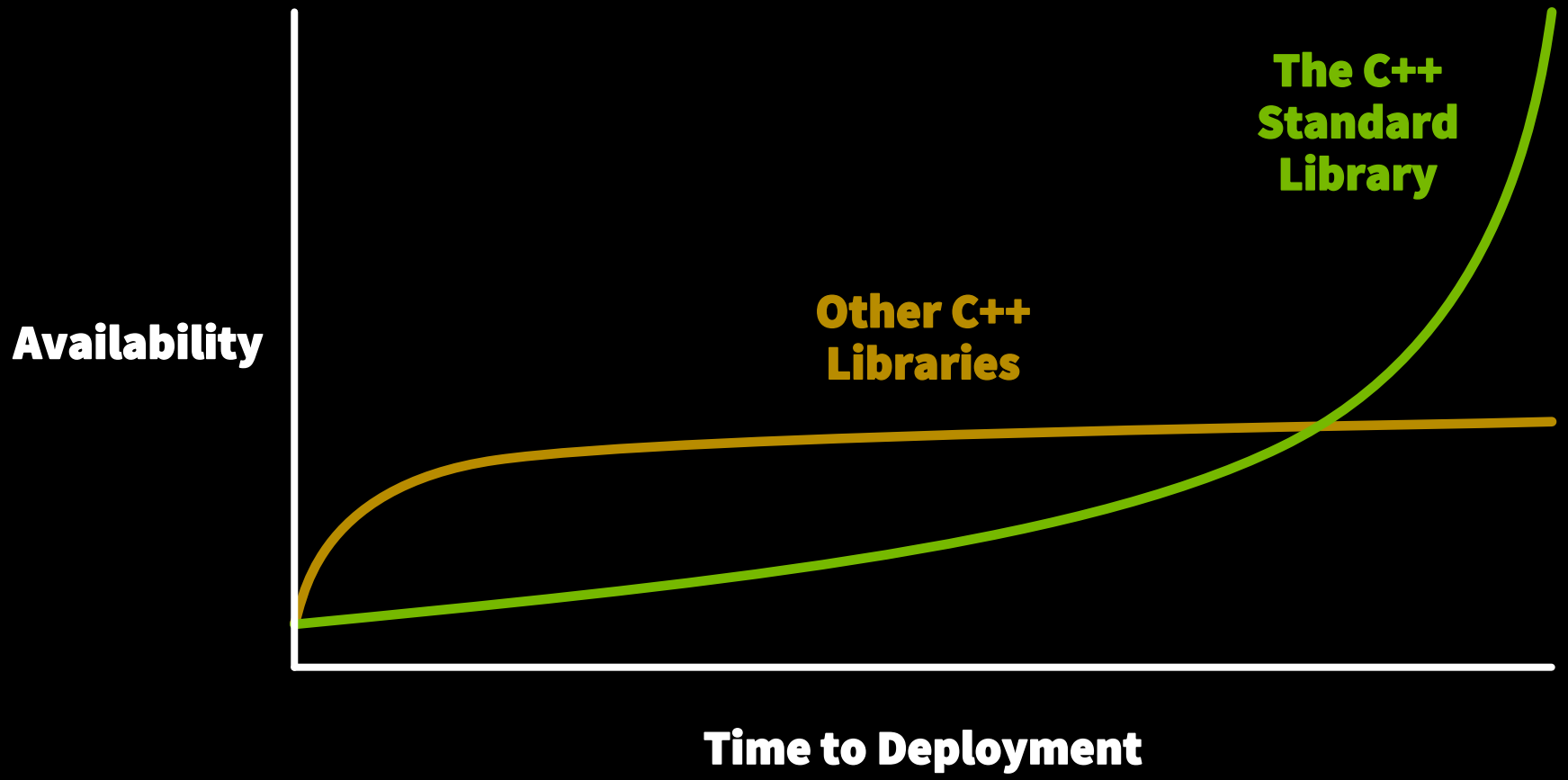
Implementation Freedom



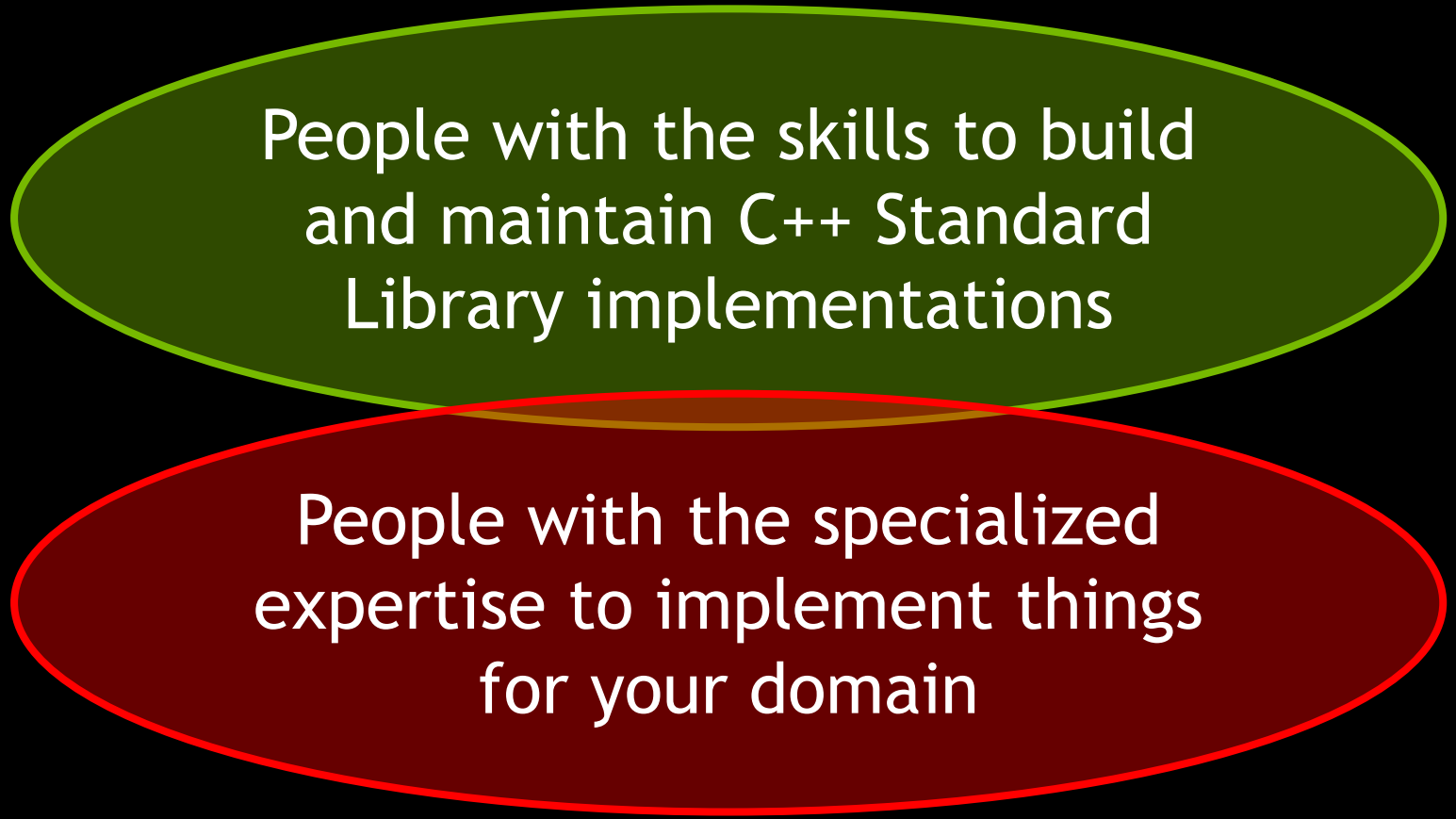
Standardization takes time.



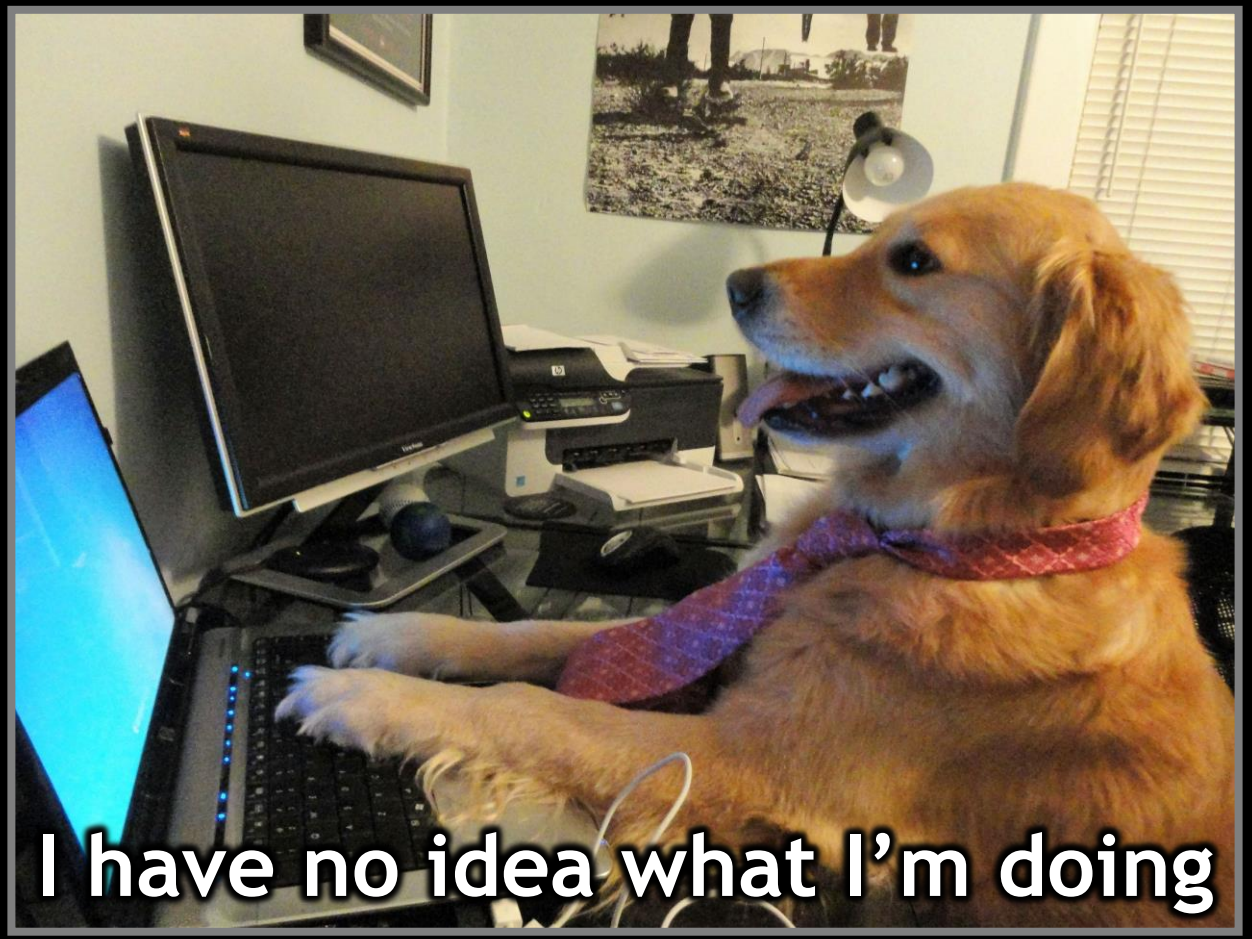




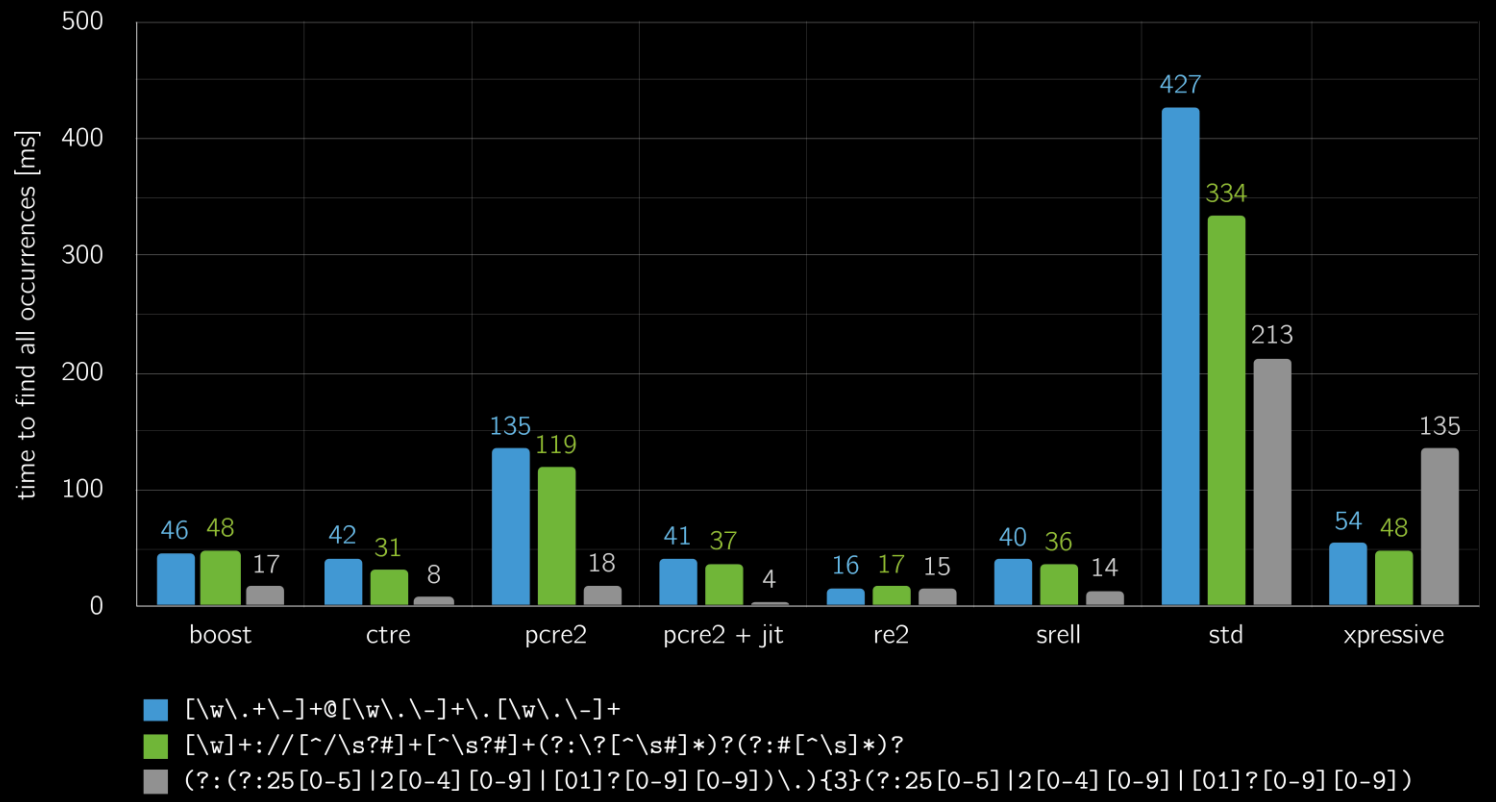
`std::` Implementors Aren't Domain Experts



`std::` Implementors Aren't Domain Experts



`std::regex` Performance



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

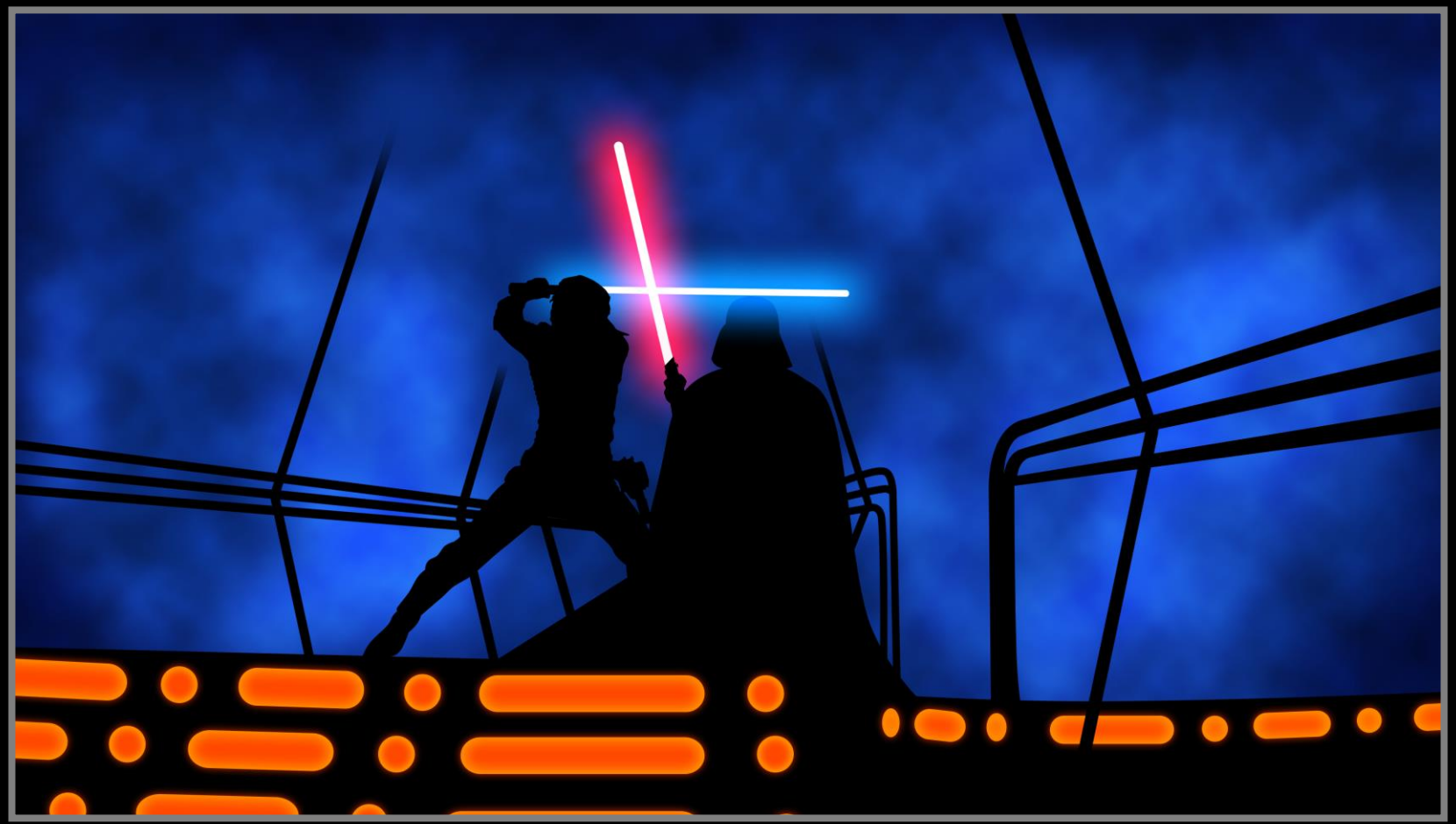
Implementers are experts at:

- 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



Hyrum's Law:

With a sufficient number of users,
it doesn't matter what you promise in the contract:
all observable behaviors of your system
will be depended on by somebody.

Casey Carter #BLM
@CoderCasey

This reminds me of the change to the destructor of std::string that @MalwareMinigun had to back out because a customer program that was destroying the same string twice broke. 🤮

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.

C++ Language ABI:

Binary representation & conventions
for language facilities.

- Function calling conventions.
- …

C++ Language ABI:

Binary representation & conventions
for language facilities.

- Function calling conventions.
- Name mangling.
- …

C++ Language ABI:

Binary representation & conventions
for language facilities.

- Function calling conventions.
- Name mangling.
- Layout and size of types.
- …

C++ Language ABI:

Binary representation & conventions
for language facilities.

- Function calling conventions.
- Name mangling.
- Layout and size of types.
- Layout of virtual tables.
- …

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.
- …

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++ 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.
- …

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.
- …

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.
- …

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.
- …

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.
- …

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.
- …

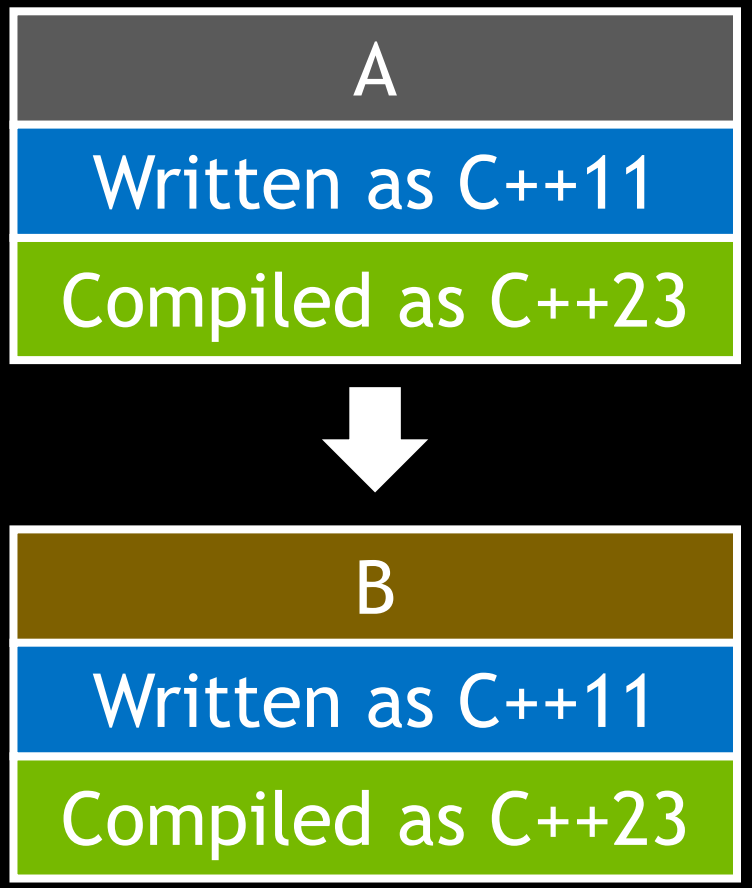
API Stability: Existing syntax and semantics should rarely change.

API Stability: Existing syntax and semantics should rarely change.

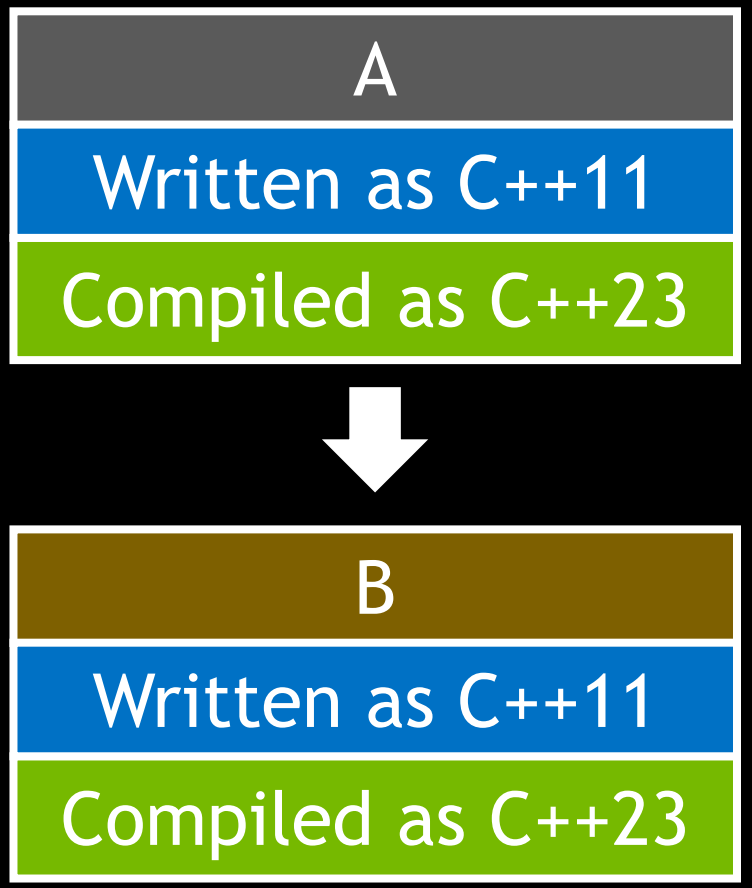
ABI Stability: Binary representations of existing facilities should rarely change.

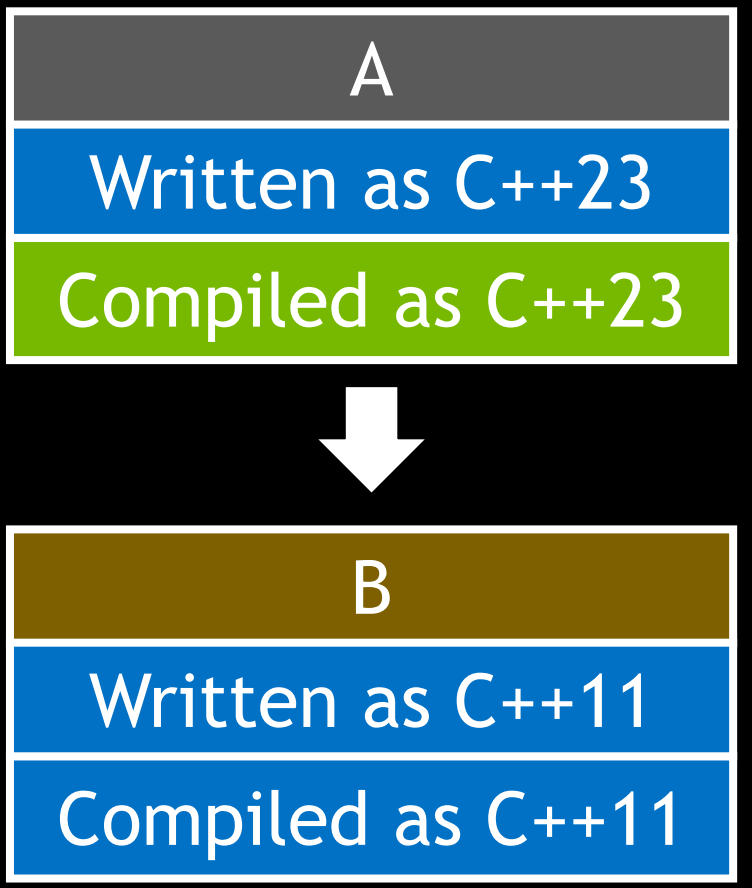
Backward Compatibility:
Newer Builds, Older Code

Backward Compatibility



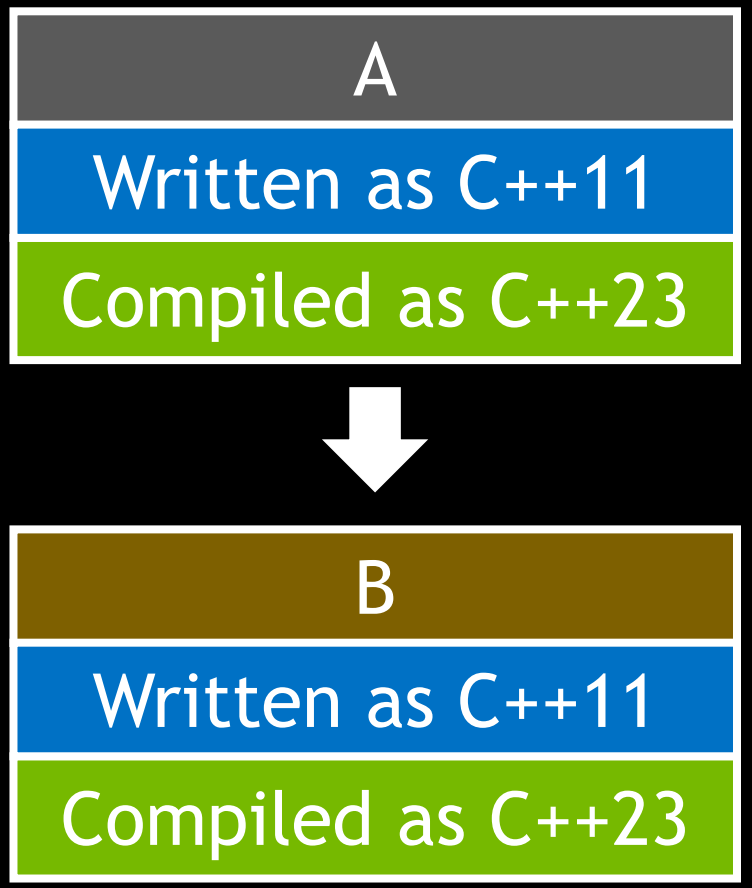
Backward Compatibility



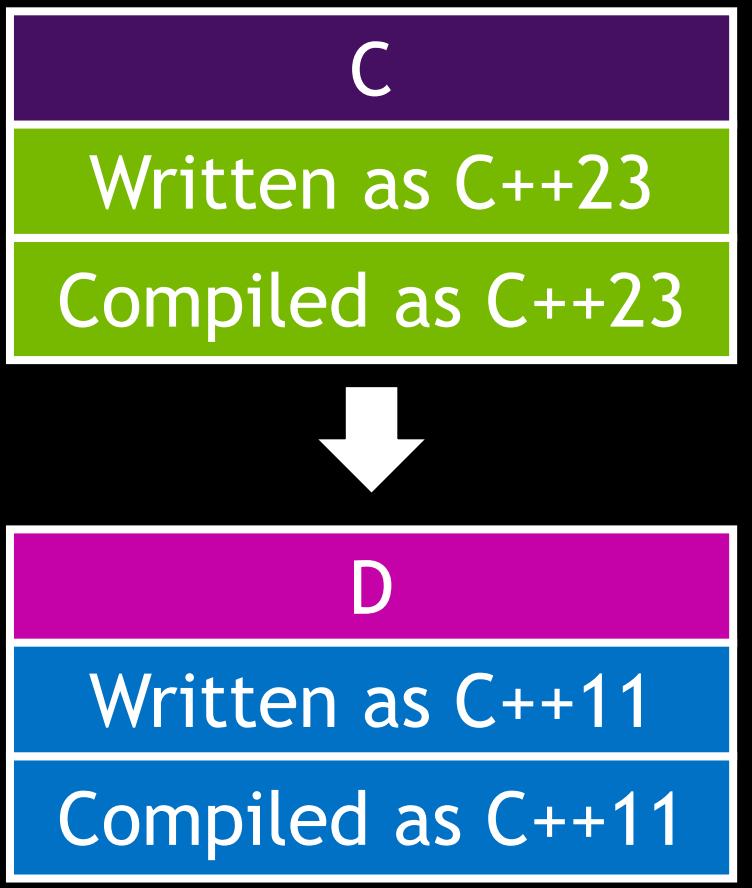


Backward Compatibility

Backward Compatibility



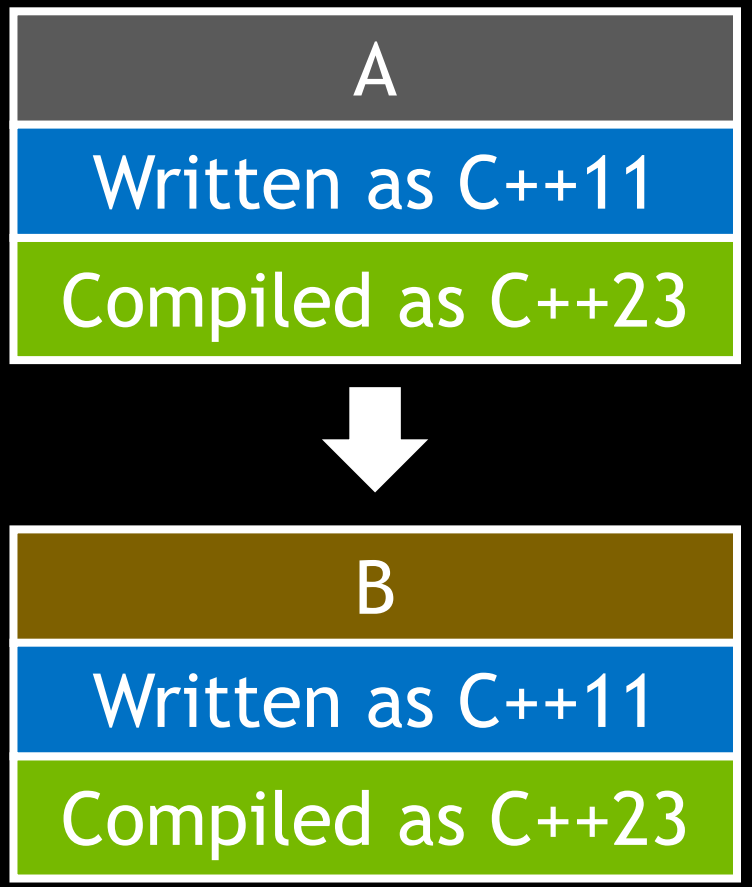
Forward Compatibility



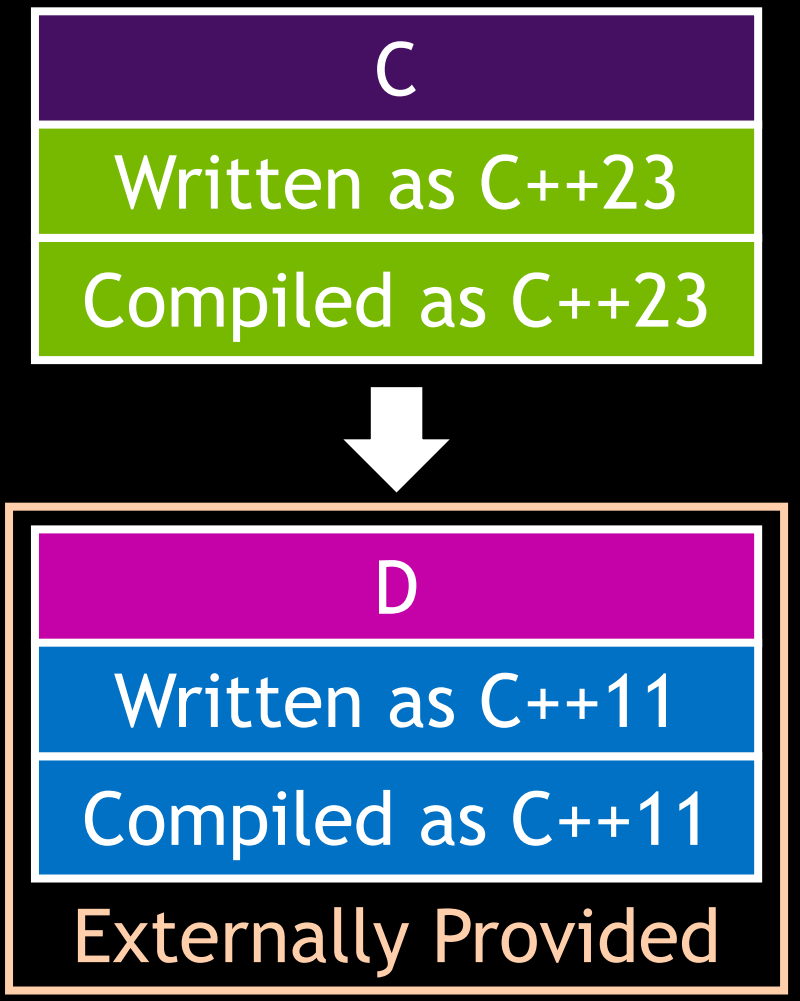
Backward Compatibility:
Newer Builds, Older Code

Forward Compatibility:
Older Builds, Newer Code

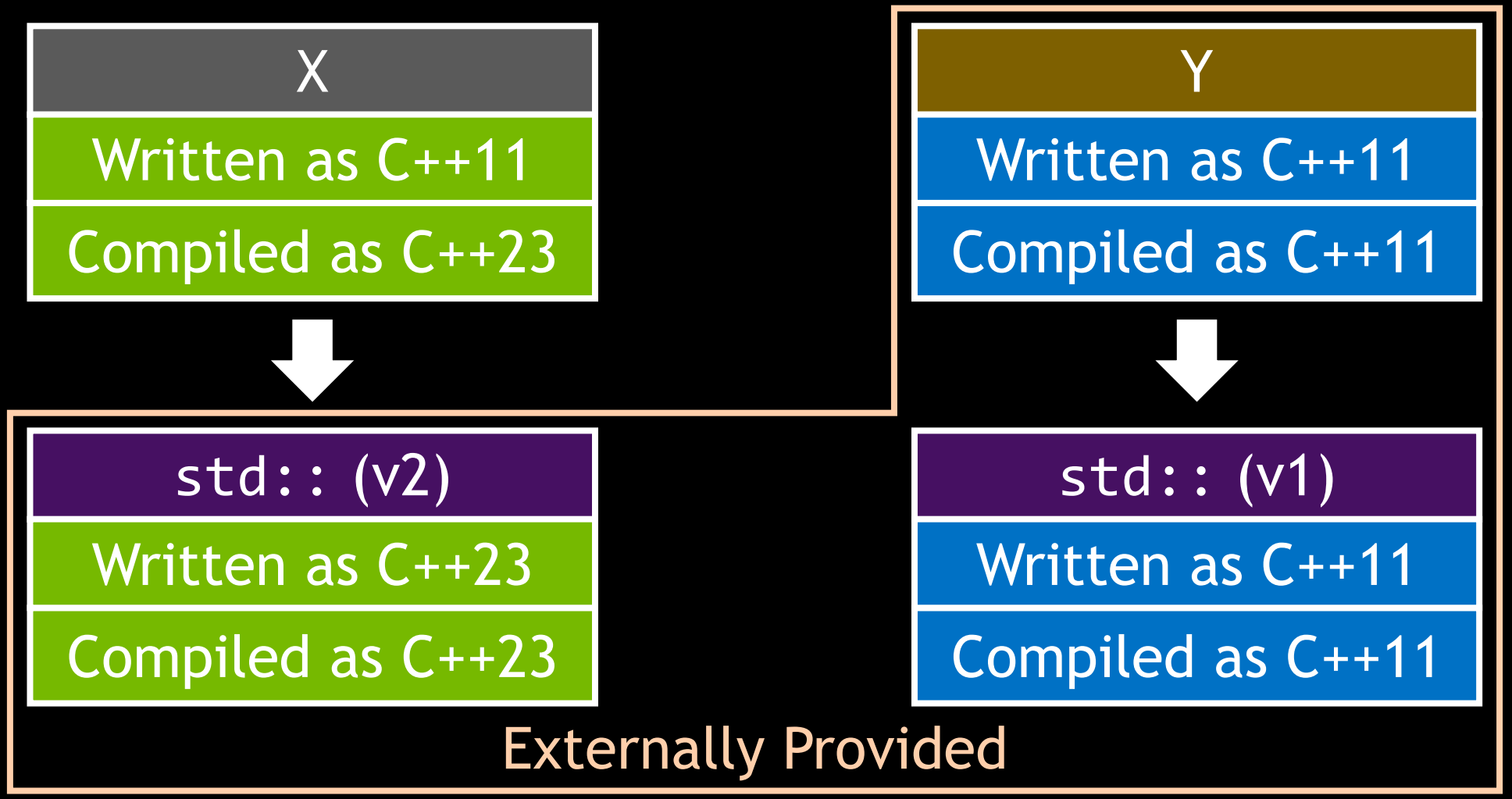
Backward Compatibility



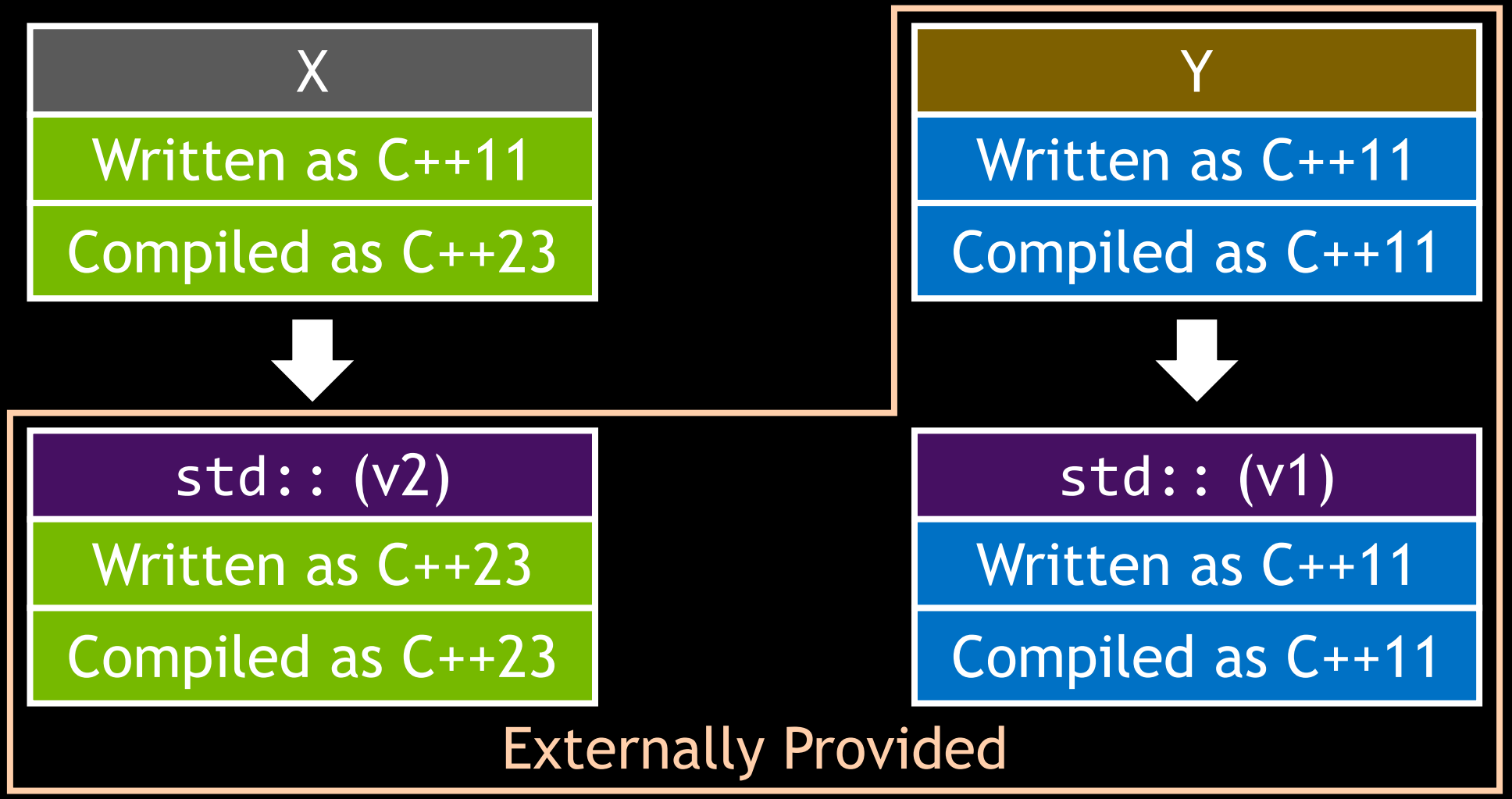
Forward Compatibility



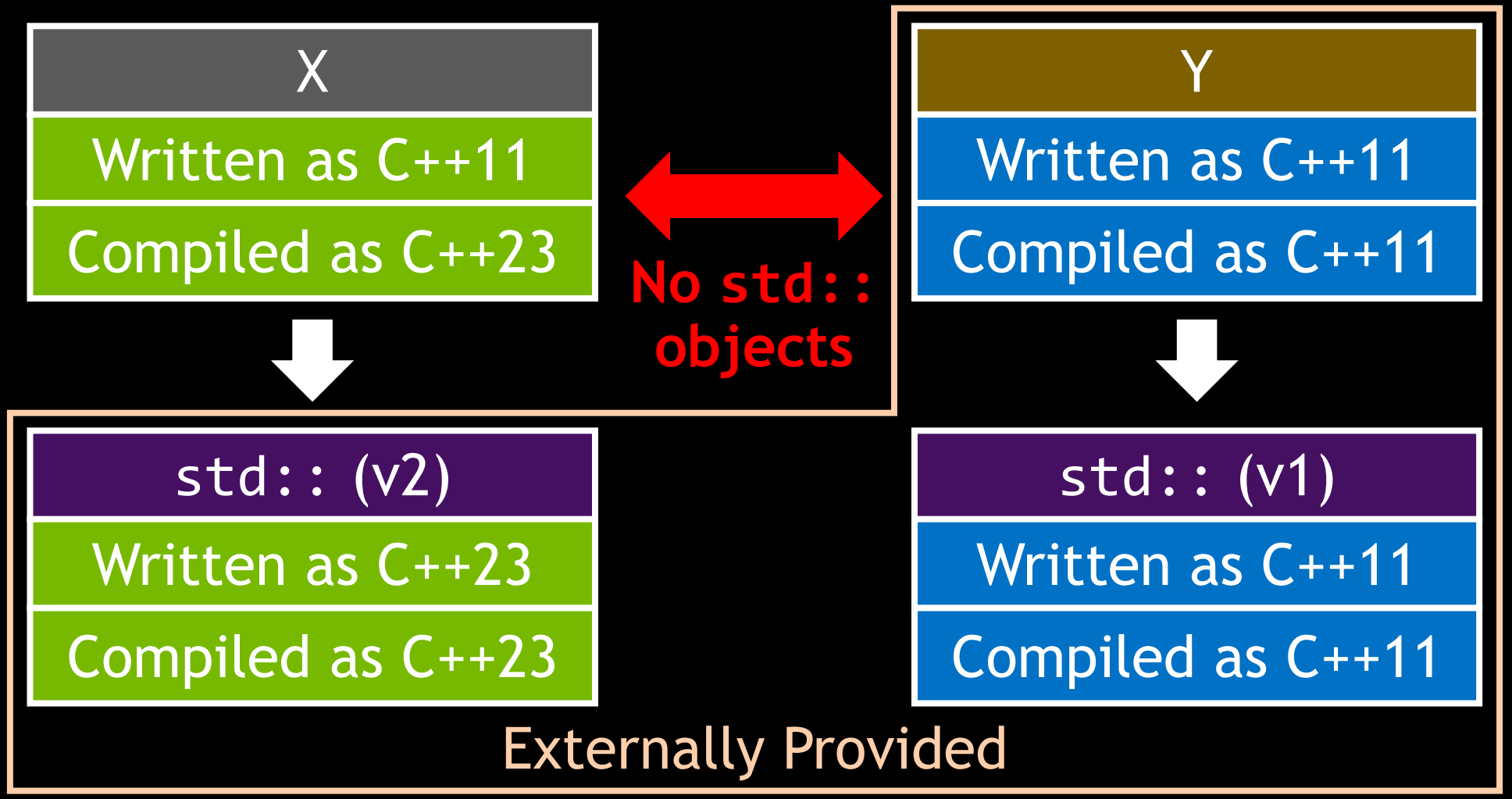
Dependencies Support All Versions



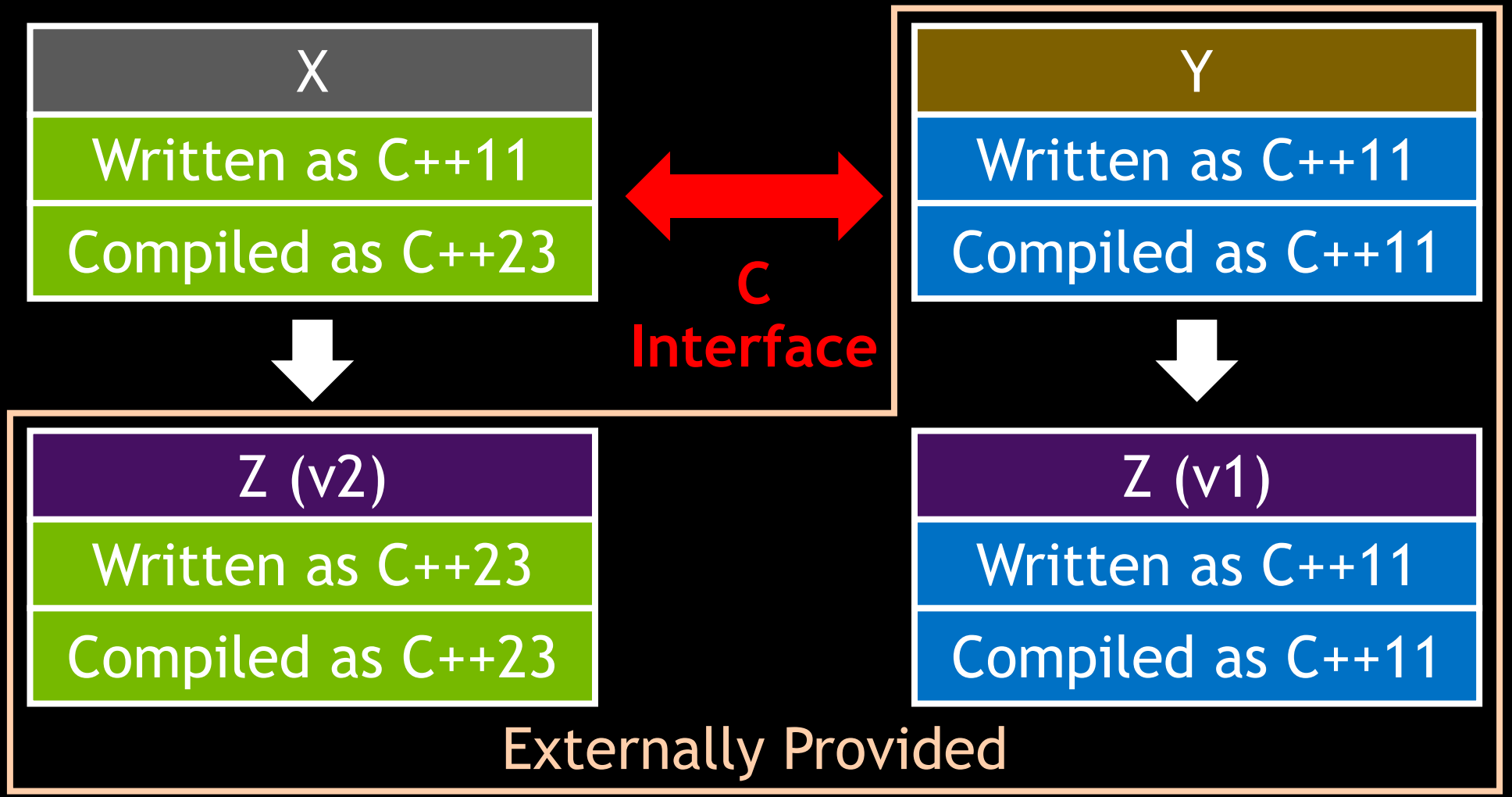
Dependencies Support All Versions



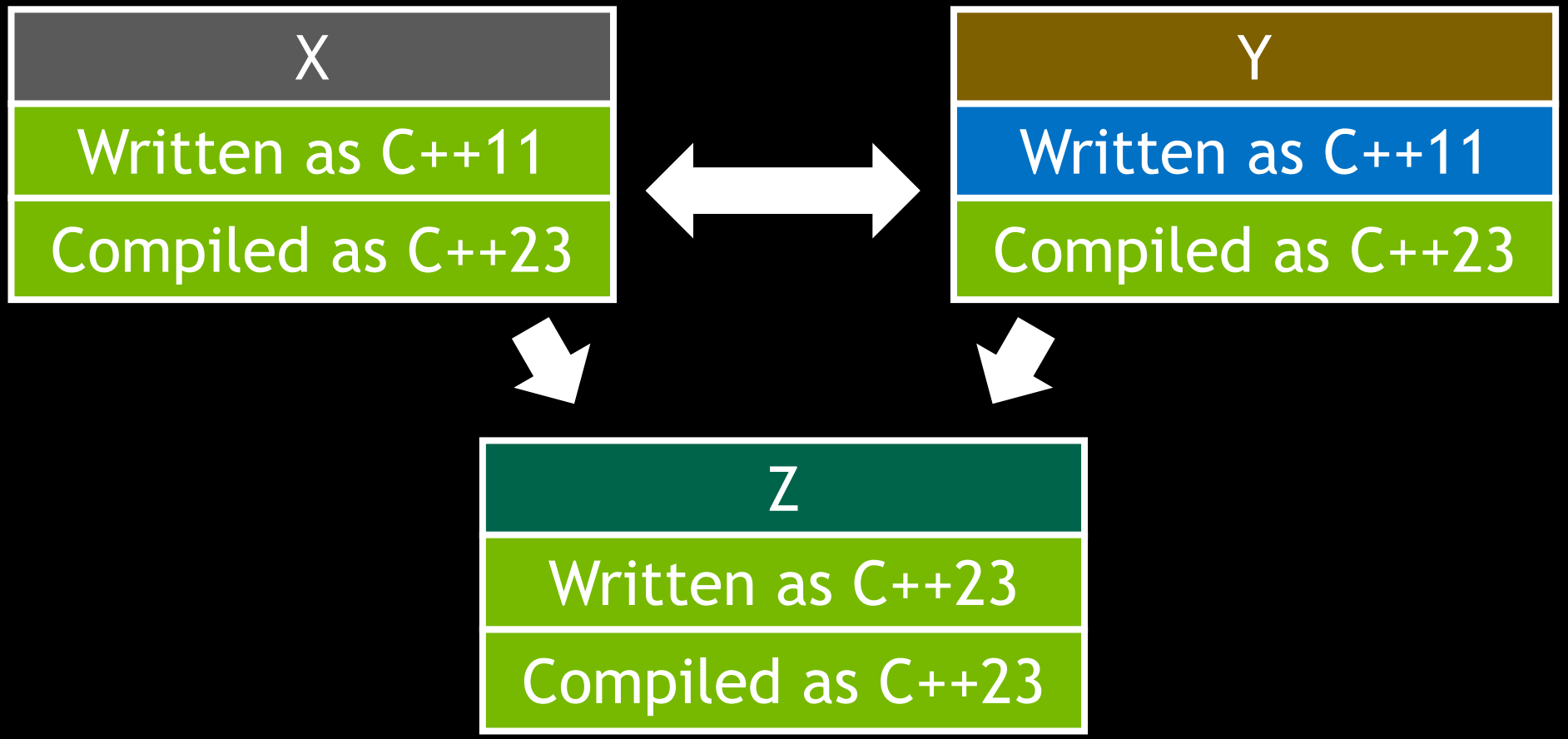
Dependencies Support All Versions



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?

How does a breaking change manifest?

- Build Time?
  - Compile Time?

How does a breaking change manifest?

- Build Time?
  - Compile Time?
  - Link Time?

How does a breaking change manifest?

- Build Time?
  - Compile Time?
  - Link Time?
- Run Time?

How does a breaking change manifest?

- Build Time?
  - Compile Time?
  - Link Time?
- Run Time?
  - Graceful?

How does a breaking change manifest?

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

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?
}
```

constexpr std:: Values Expose You To ABI Breaks

```
struct ticket_mutex {
  alignas(std::hardware_destructive_interference_size)
    std::atomic<int> in;
  alignas(std::hardware_destructive_interference_size)
    std::atomic<int> out;
};
```

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

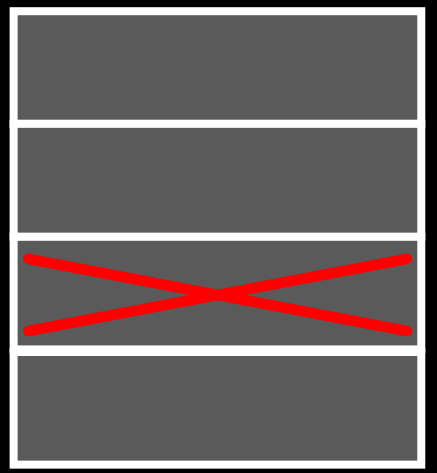
```
namespace std {

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
>;
```

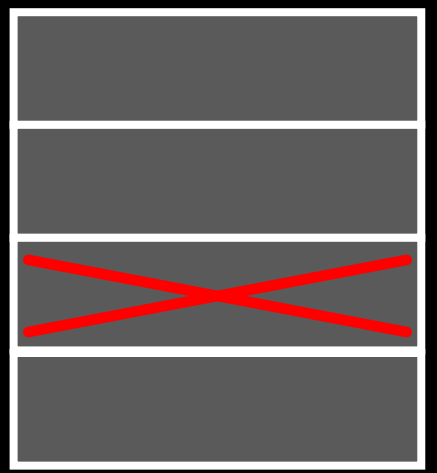
Polymorphism In `std::` Exposes You To ABI Breaks

Removing virtual functions causes layout changes.

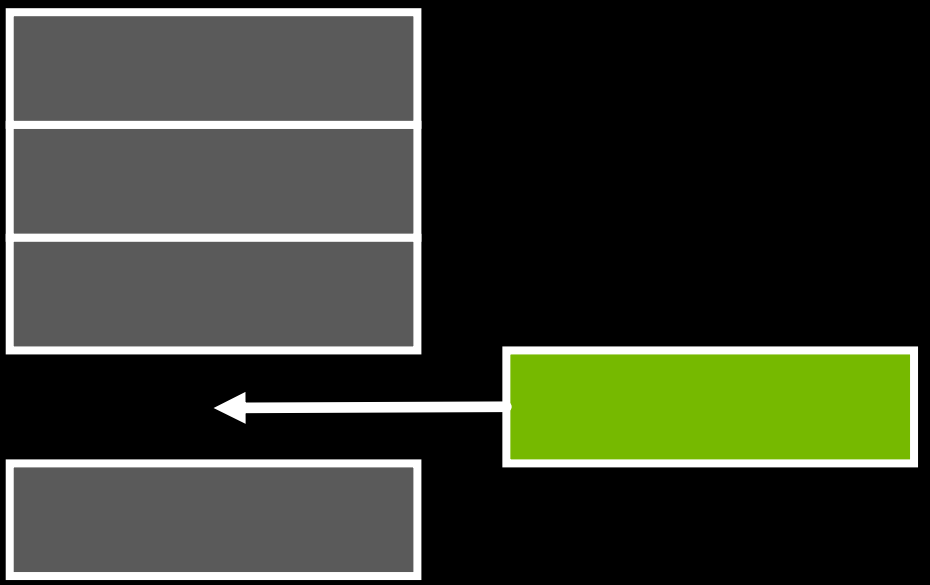


Polymorphism In `std::` Exposes You To ABI Breaks

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::list::size Complexity

C++03: std::list::size can have linear complexity. No size data member needed.

```
template <
  class T,
  class A = std::allocator<T>>
class std::list {
  __list_node<T> root;

};

sizeof(std::list<int>) == 16
```

C++11 std::list::size Complexity

C++03: `std::list::size` can have linear complexity. No size data member needed.

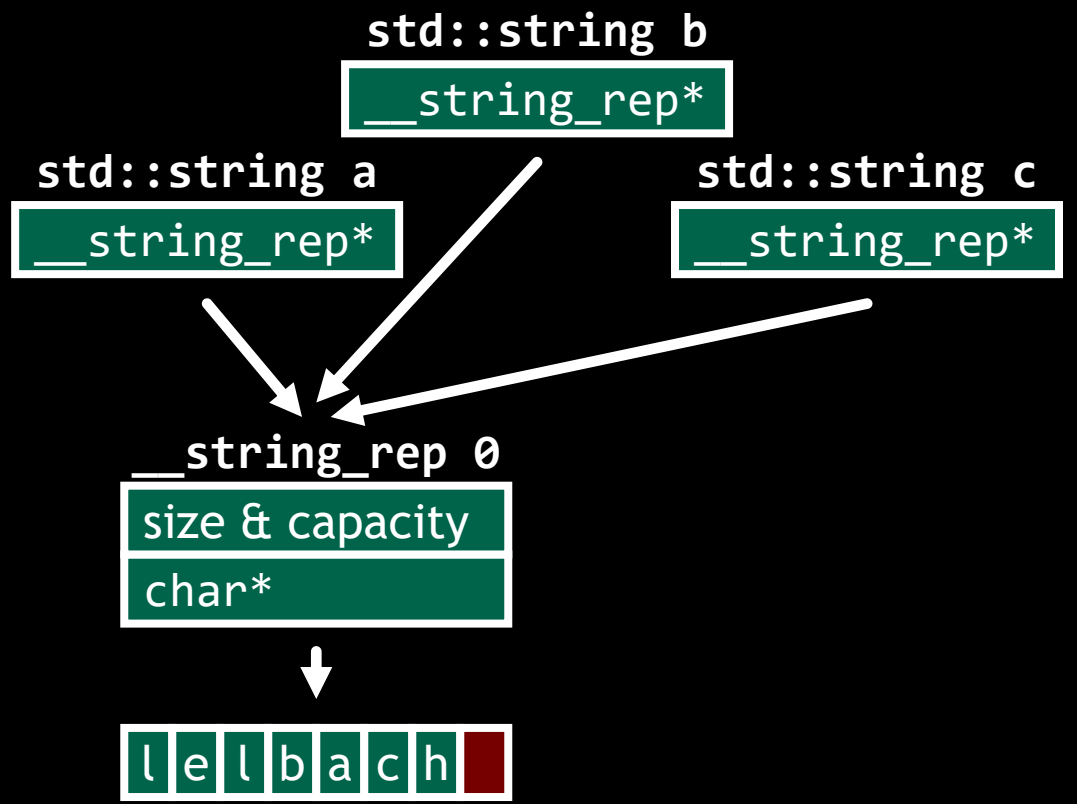
```
template <
  class T,
  class A = std::allocator<T>>
class std::list {
  __list_node<T> root;

};

sizeof(std::list<int>) == 16
```

C++11: `std::list::size` must have linear complexity. A size data member is required.

```
template <
  class T,
  class A = std::allocator<T>>
class std::list {
  __list_node<T> root;
  std::size_t size;
};

sizeof(std::list<int>) == 24
```

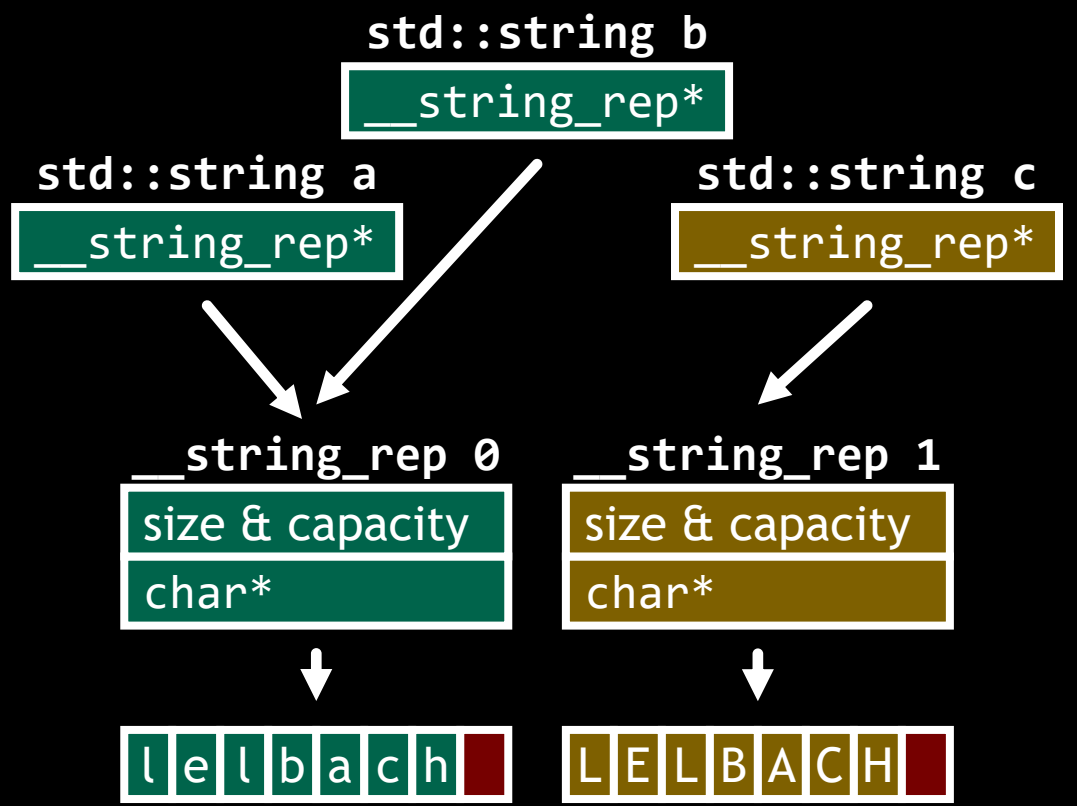
C++11 `std::string` COW Prohibition

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



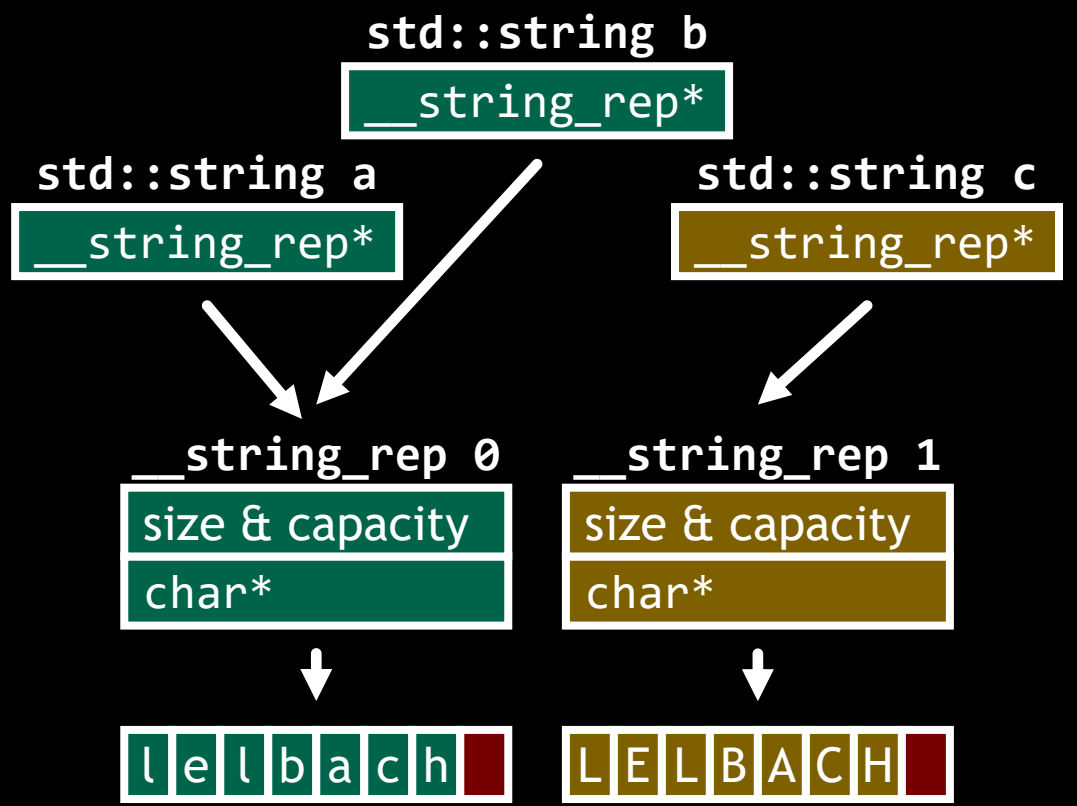
C++11 `std::string` COW Prohibition

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



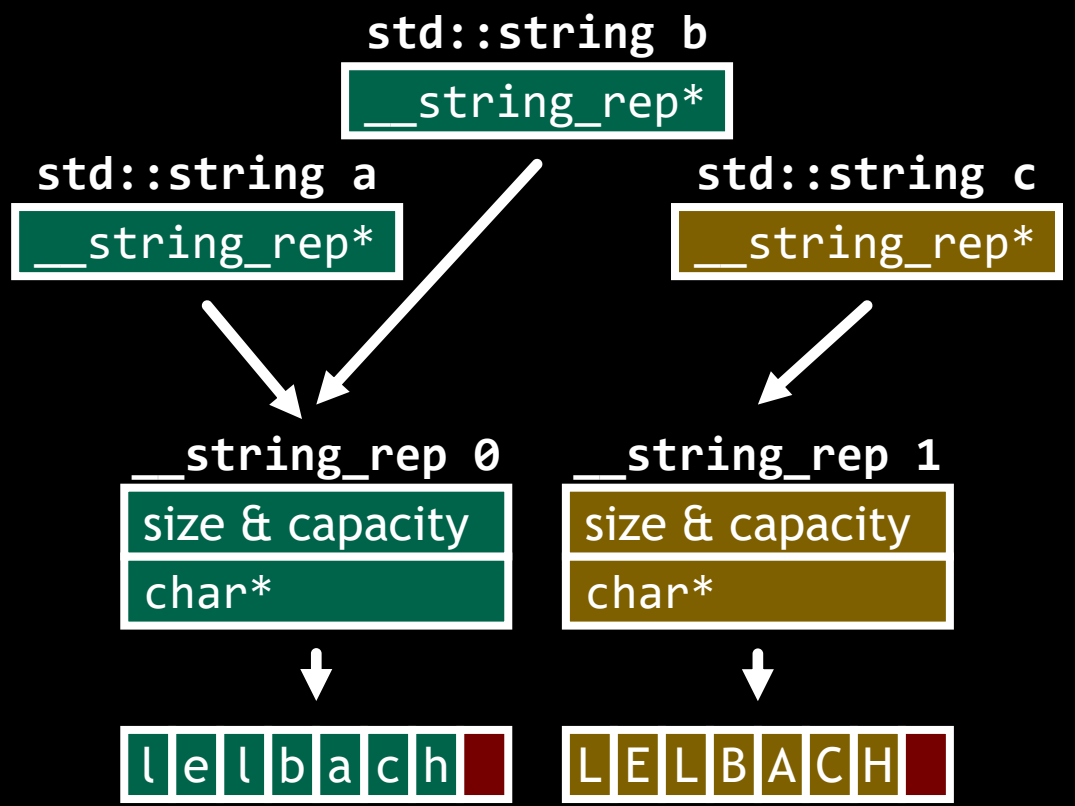
C++11 `std::string` COW Prohibition

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

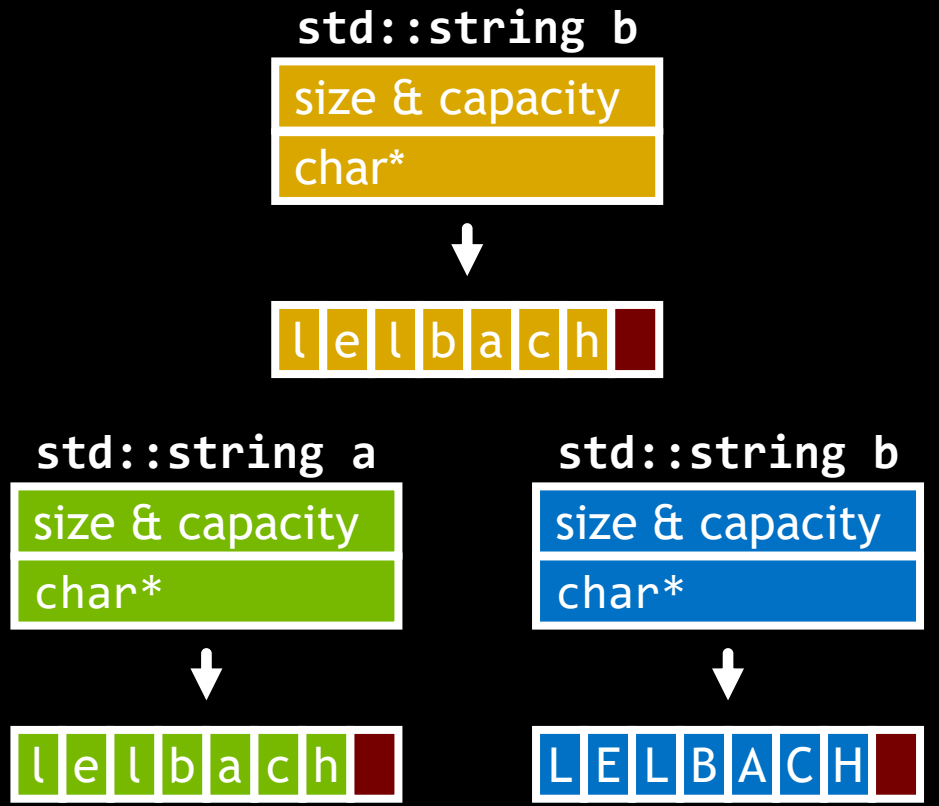


C++11 `std::string` COW Prohibition

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

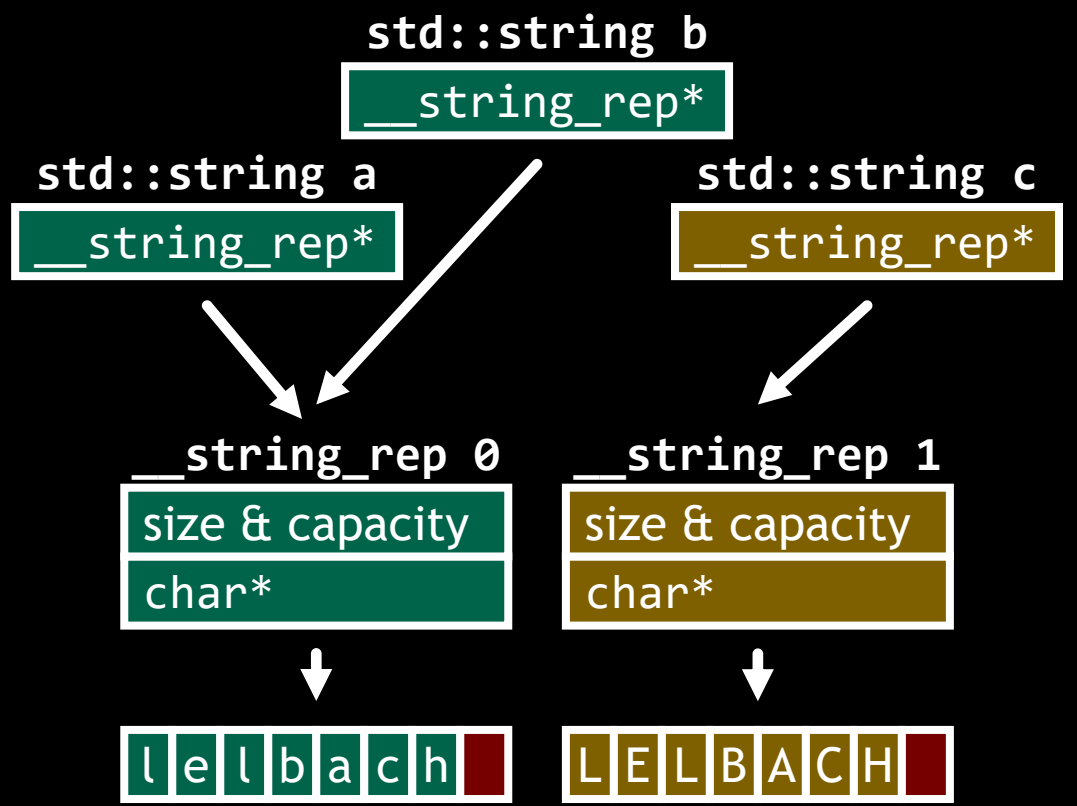


But in C++11, we prohibited it.

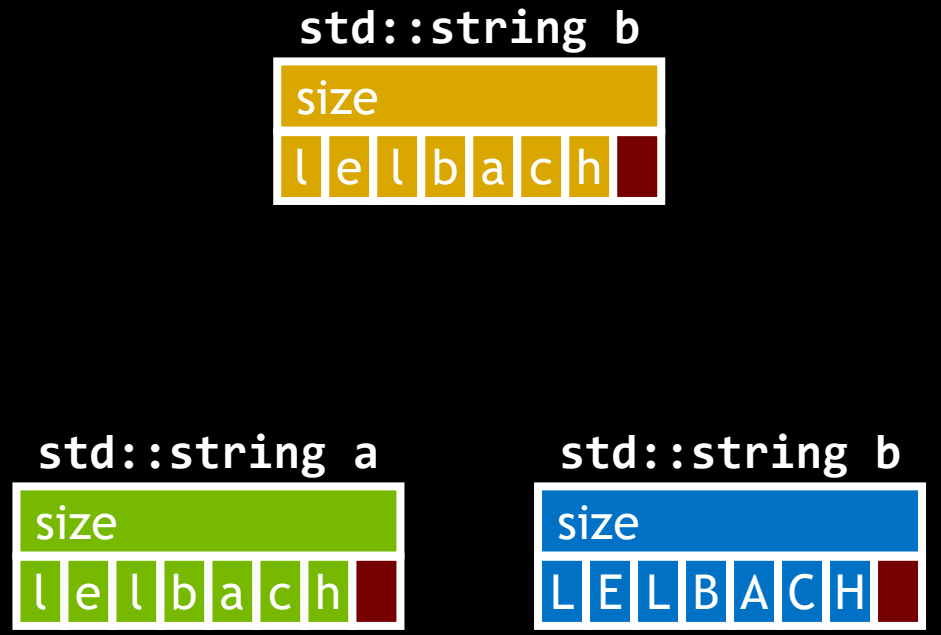


C++11 `std::string` COW Prohibition

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



But in C++11, we prohibited it.



Where the C++11 `std::list` and `std::string` breaks the right decision?

C++17 Variadic `std::lock_guard`

```
template <class MutexType>
struct std::lock_guard;


{
  std::lock_guard<std::mutex> l0(mtx0);
  std::lock_guard<std::mutex> l1(mtx1);
  // ...
}
```

C++17 Variadic `std::lock_guard`

```
template <class... MutexTypes>
struct std::lock_guard;


{
  std::lock_guard l(mtx0, mtx1);
  // ...
}
```

C++17 Variadic `std::lock_guard`

`template <class MutexType>`
`struct std::lock_guard {};`
 	
`void f(std::lock_guard<std::mutex>);`
	`_Z1fSt10lock_guardISt5mutexE`

`template <class... MutexTypes>`
`struct std::lock_guard {};`
 	
`void f(std::lock_guard<std::mutex>);`
	`_Z1fSt10lock_guardIJSt5mutexEE`

C++17 Variadic `std::lock_guard`

`template <class MutexType>`
`struct std::lock_guard {};`
 	
`void f(std::lock_guard<std::mutex>);`
	`_Z1fSt10lock_guardISt5mutexE`

`template <class... MutexTypes>`
`struct std::scoped_lock {};`
 	
`void f(std::scoped_lock<std::mutex>);`
	`_Z1fSt11scoped_lockIJSt5mutexEE`

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;
```

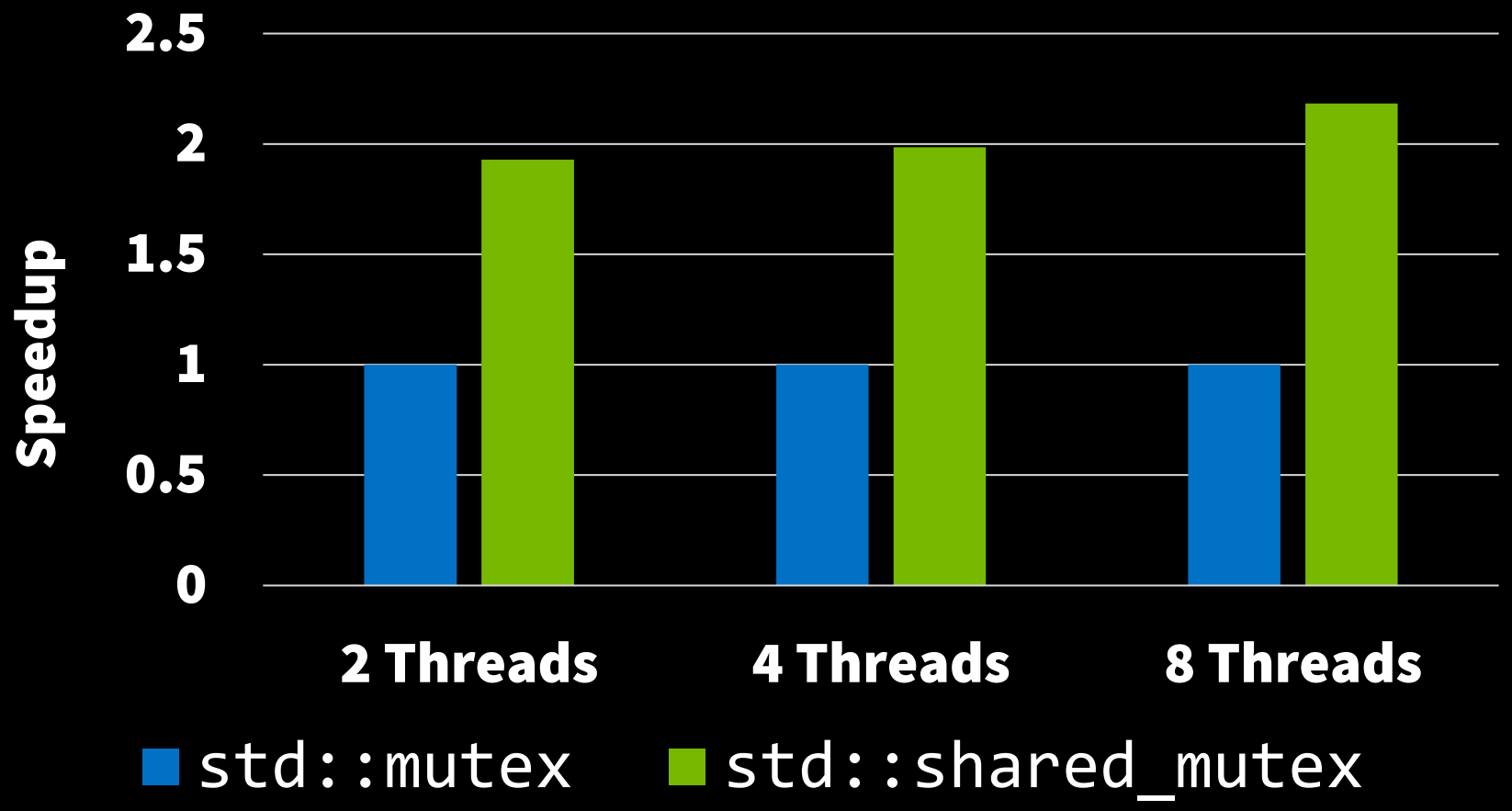
```
virtual bool
equivalent(error_code const&, int) const noexcept;
```

```
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
```

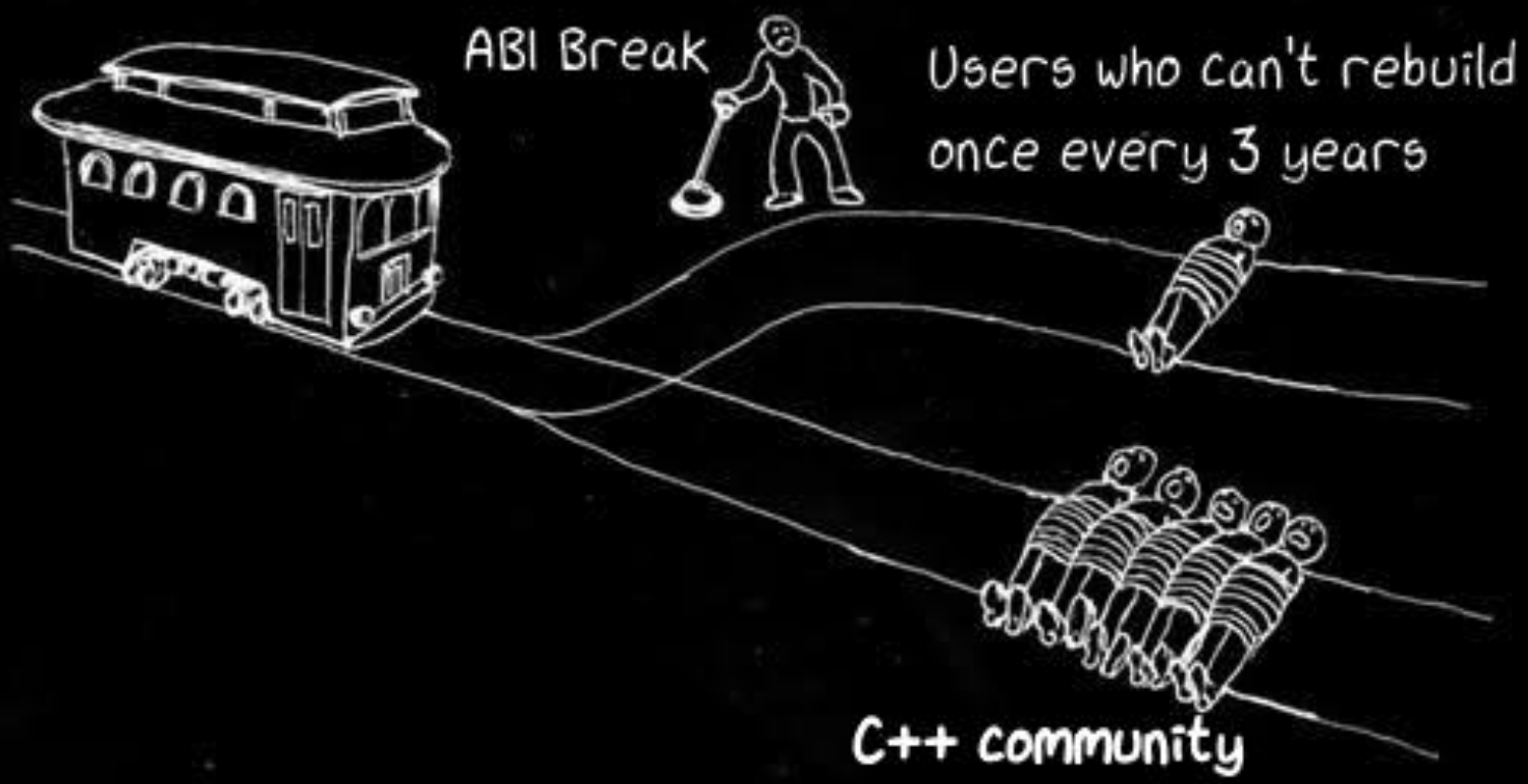
The C++ Standard Library is:

- Good at stability.

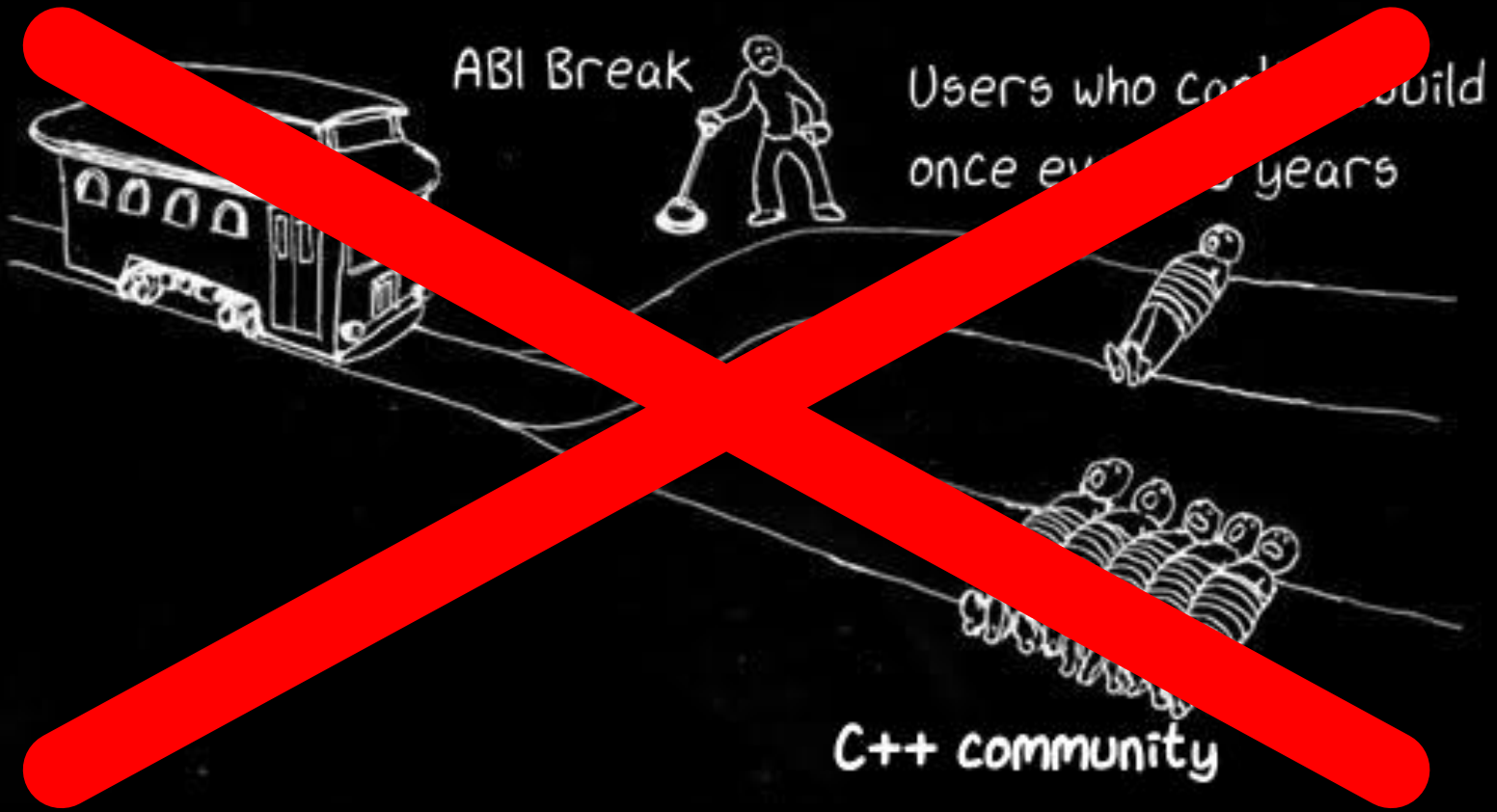
The C++ Standard Library is:

- 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 __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	☑	☒

abi_tag

```
namespace std {
inline namespace __cxxNN __attribute__((abi_tag)) {

// ...

}}
```

Viral;
if something with an ABI tag is in a function's signature, then the function gets an ABI tag.

abi_tag

```
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`

abi_tag

```
namespace std {
inline namespace __cxxNN
  __attribute__((abi_tag)) {

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: `_Z1fB7__cxx11v`

B	Compiled as C++23

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

Expected Mangling: `_Z1fB7__cxx23v`

abi_tag

```
namespace std {
inline namespace __cxxNN
  __attribute__((abi_tag)) {

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	☑	☒

std2::

Introduce new version of existing `std::` features which are not compatible, but might be interoperable.

```
void f(std::__cxx20::string& s);
void f(std::__cxx23::string& s);

void f(std::string& s);
void f(std2::string& s);
```

Are these ambiguous, or necessary?

std2:: Approach

Any solution equivalent to
“duplicate and maintain multiple generations of the same facilities”.

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.

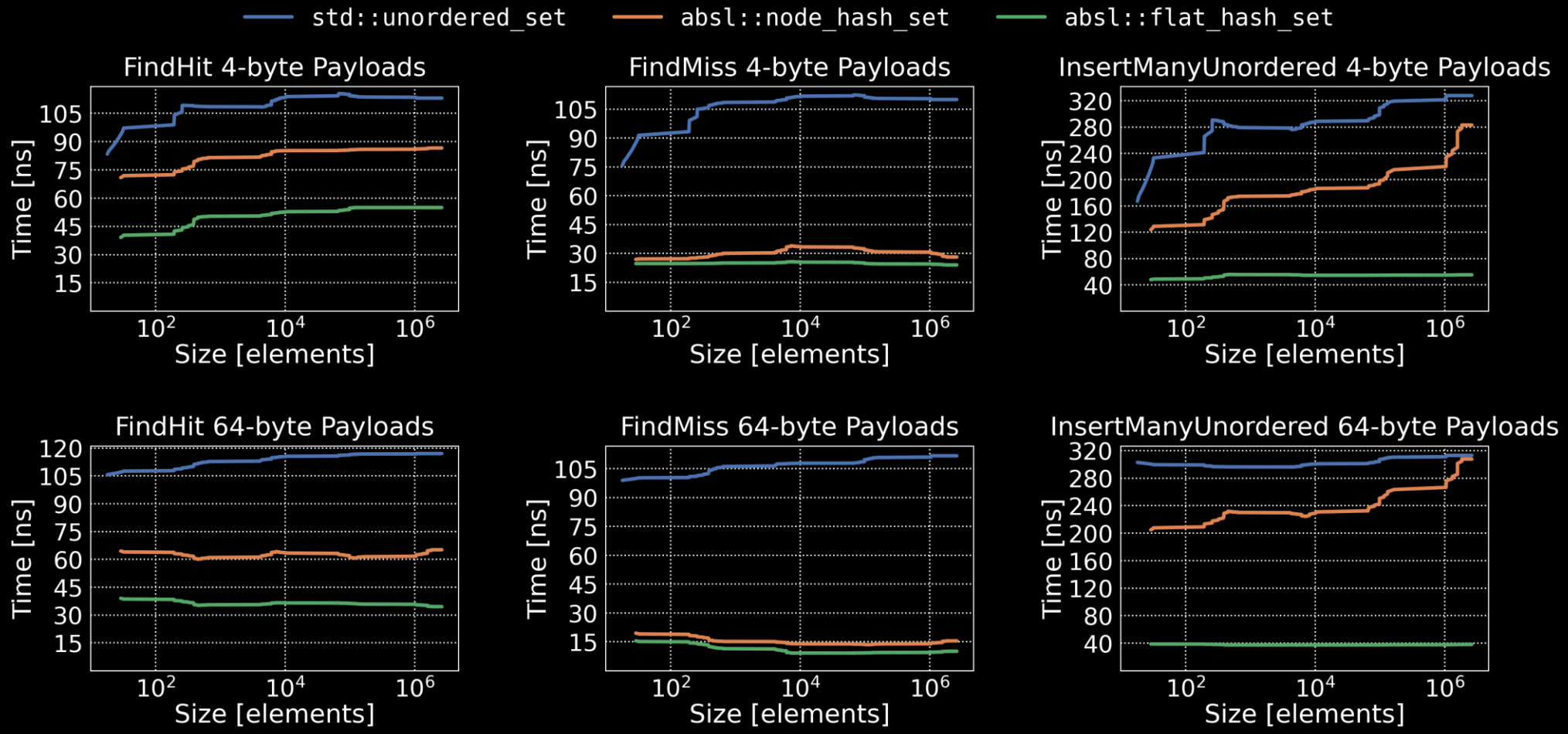
Anything that goes into the C++ Standard Library must stand the test of time.

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.

Good Implementation Experience	Better Implementation Experience
Prototype.	Production.
Preceding or similar to the standard.	Written from or conforming to the standard.
For one platform.	For multiple platforms.
In any publicly available codebase.	In a C++ Standard Library codebase.

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…

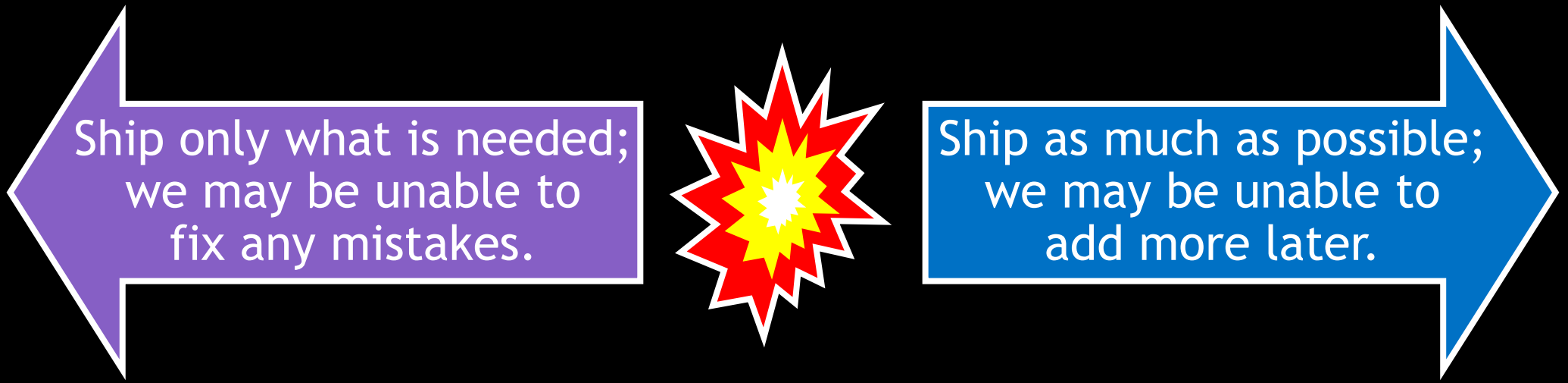
Ship only what is needed; we may be unable to fix any mistakes.

Ship as much as possible; we may be unable to add more later.

Smaller Scope

Larger Scope

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



Smaller Scope

Larger Scope

Incrementalism is key to
C++ Standard Library Evolution.

We’re bad at changing things,
but we’re good at extending things.

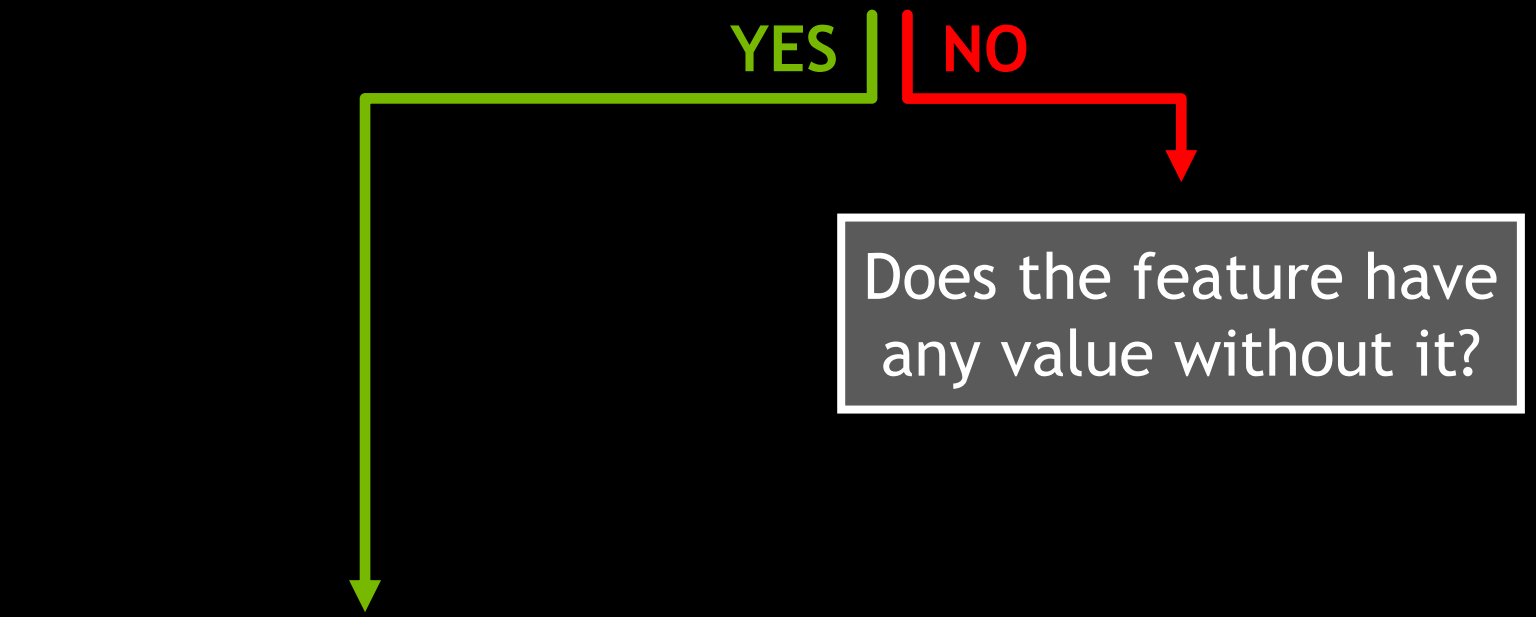
If we don’t include this in the initial release of the feature, is it a breaking change to add it later?

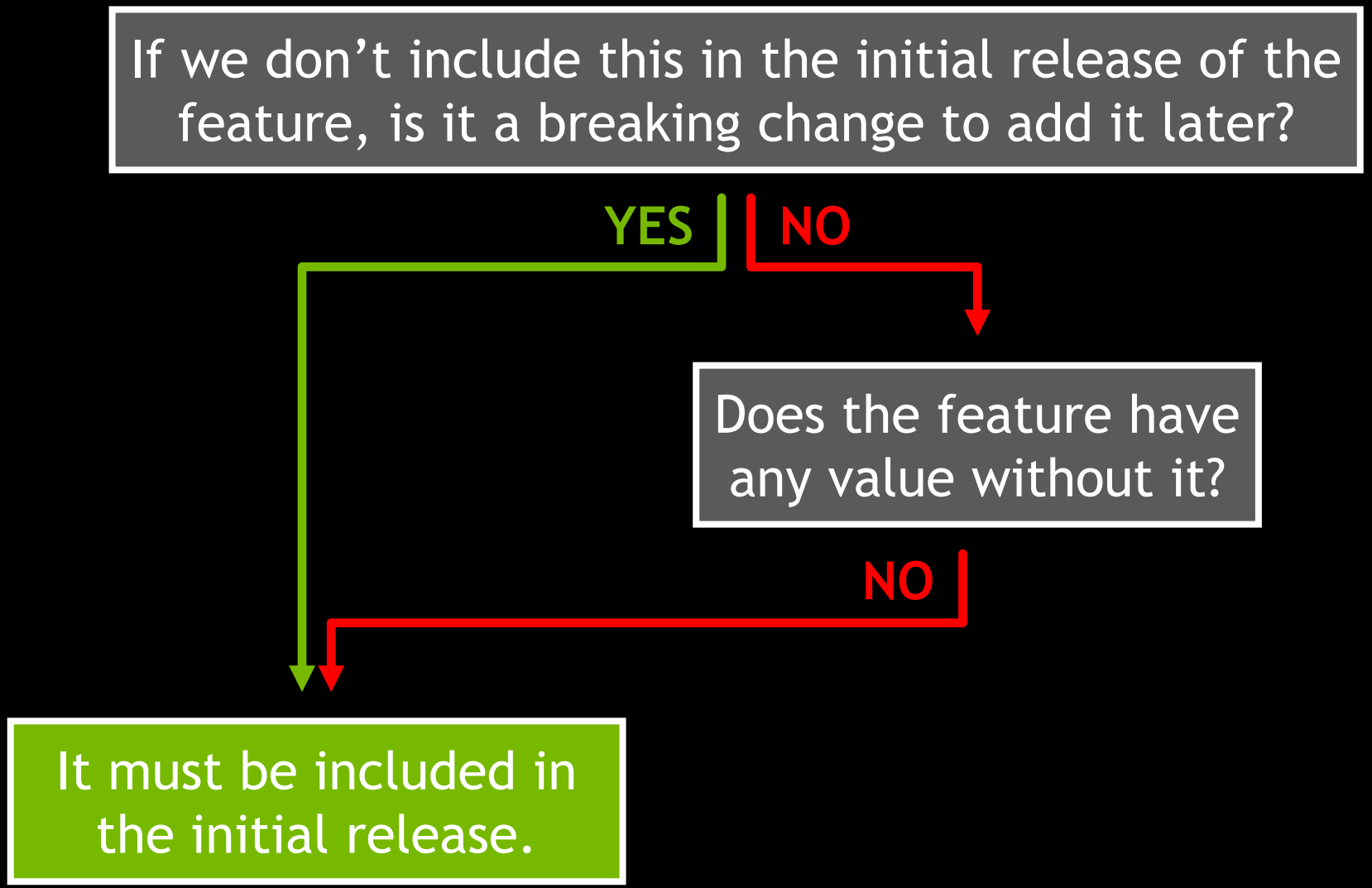
If we don’t include this in the initial release of the feature, is it a breaking change to add it later?

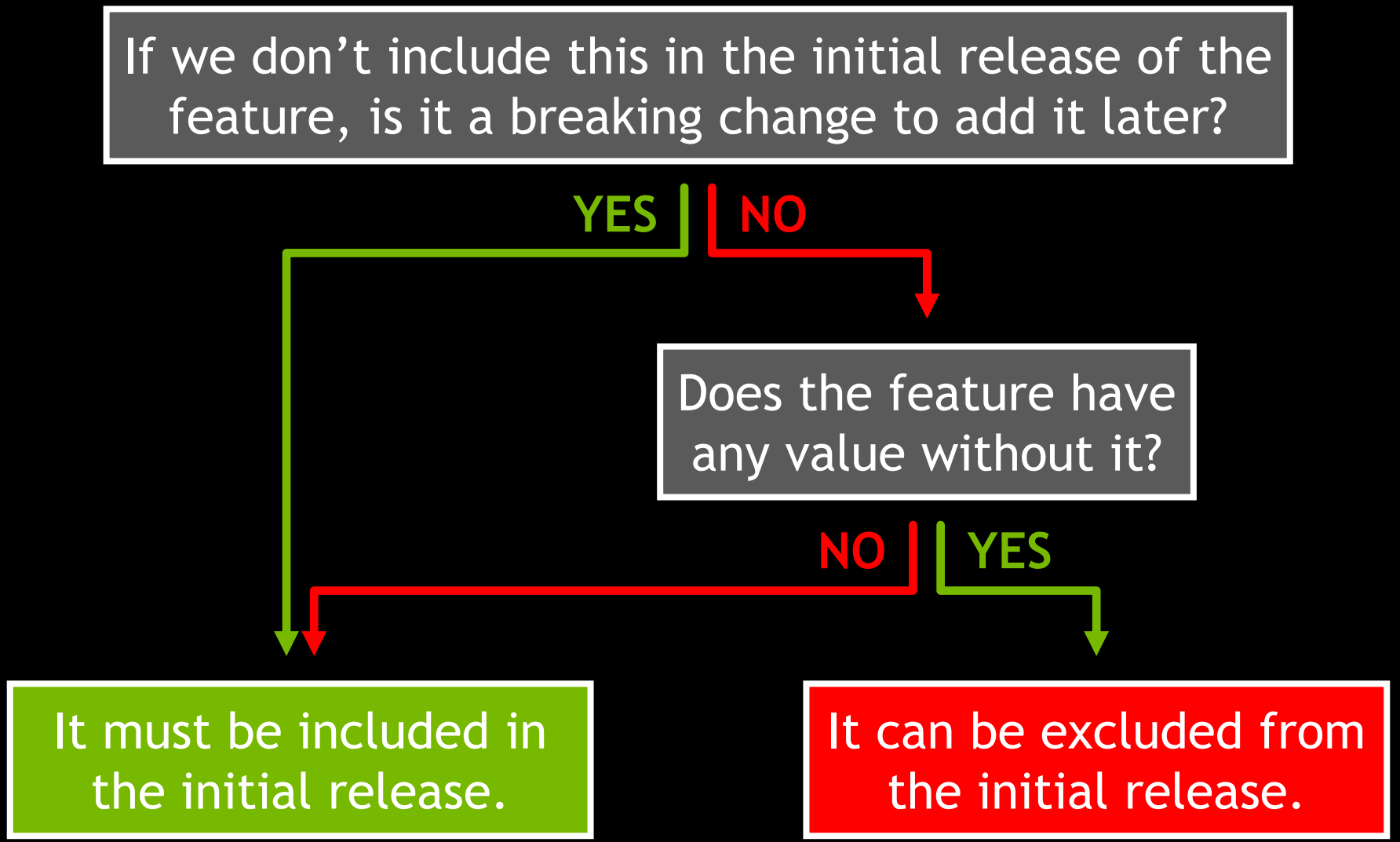
YES

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?







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.

Interface Vocabulary:

Concepts, types, and operations that commonly appear in C++ interfaces.

Common definitions means different codebases can interoperate.

Interface Vocabulary:

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

Interface 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);
```

Interface 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);
```

Interface Vocabulary:

- Concepts

Interface Vocabulary:

- Concepts
- Containers and Views

Interface Vocabulary:

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

Interface Vocabulary:

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

Tooling Vocabulary:

Facilities that tools want to recognize.

Tooling Vocabulary:

Facilities that tools want to recognize.

- MSVC iterator debugging.
- GDB container pretty printing.

Tooling Vocabulary:

Facilities that tools want to recognize.

- MSVC iterator debugging.
- GDB container pretty printing.
- Clang thread safety analysis.

The Necessity Thesis:

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

- Language Support
- Portability
- Vocabulary

What belongs in the C++ Standard Library?

C++ Standard Library priorities for 2020s:

C++ Standard Library priorities for 2020s:

- Asynchrony & Parallelism

C++ Standard Library priorities for 2020s:

- Asynchrony & Parallelism
- Input & Output

C++ Standard Library priorities for 2020s:

- Asynchrony & Parallelism
- Input & Output
- Text Processing

C++ Standard Library priorities for 2020s:

- Asynchrony & Parallelism
- Input & Output
- Text Processing
- Metaprogramming & Reflection

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.

The Burden Of Being In `std::`

- Must be ABI stable.
- Must support every C++ compiler and platform.
- Must support every corner case.
- Must support every compilation mode.
- …

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?

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

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

YES

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

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.

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.

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

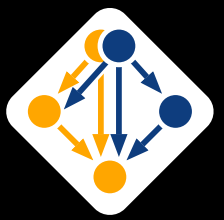
Okay let's talk.

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.

What would we even standardize?

What would we even standardize?

- C++ package metadata formats?
- …

What would we even standardize?

- C++ package metadata formats?
- A C++ package manager?
- …

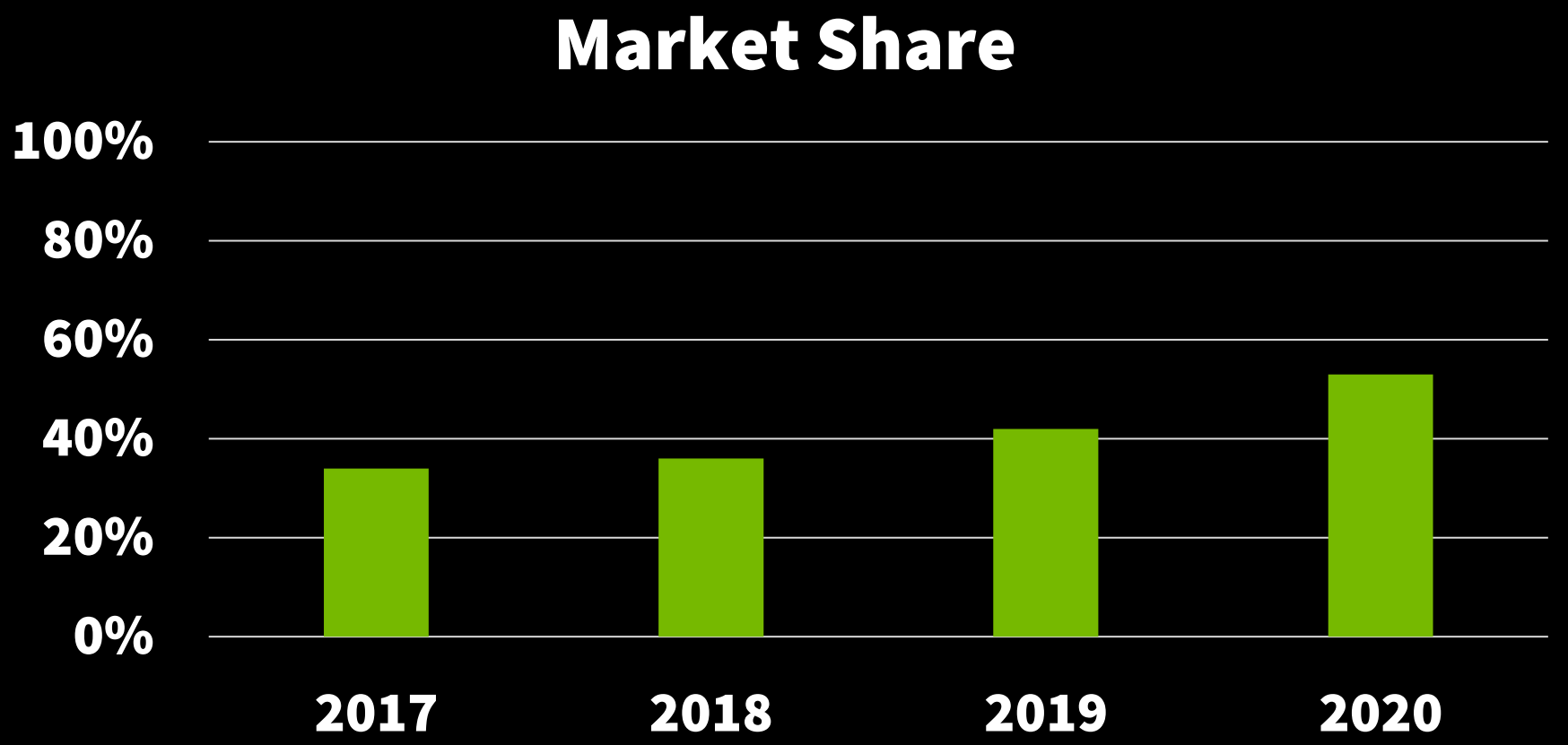
What would we even standardize?

- C++ package metadata formats?
- A C++ package manager?
- Centralized C++ package repository?
- …

What would we even standardize?

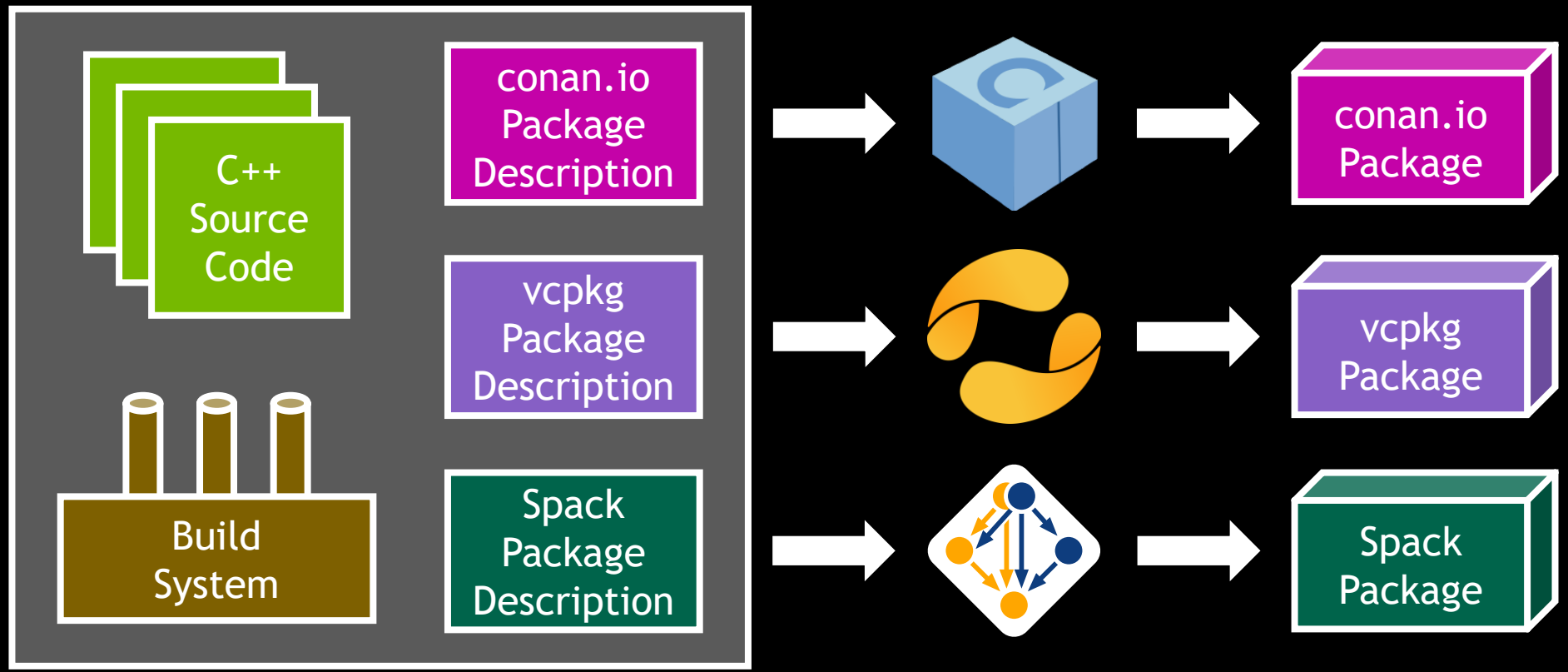
- C++ package metadata formats?
- A C++ package manager?
- Centralized C++ package repository?
- A C++ build system?
- …

CMake Is The Standard C++ Build System

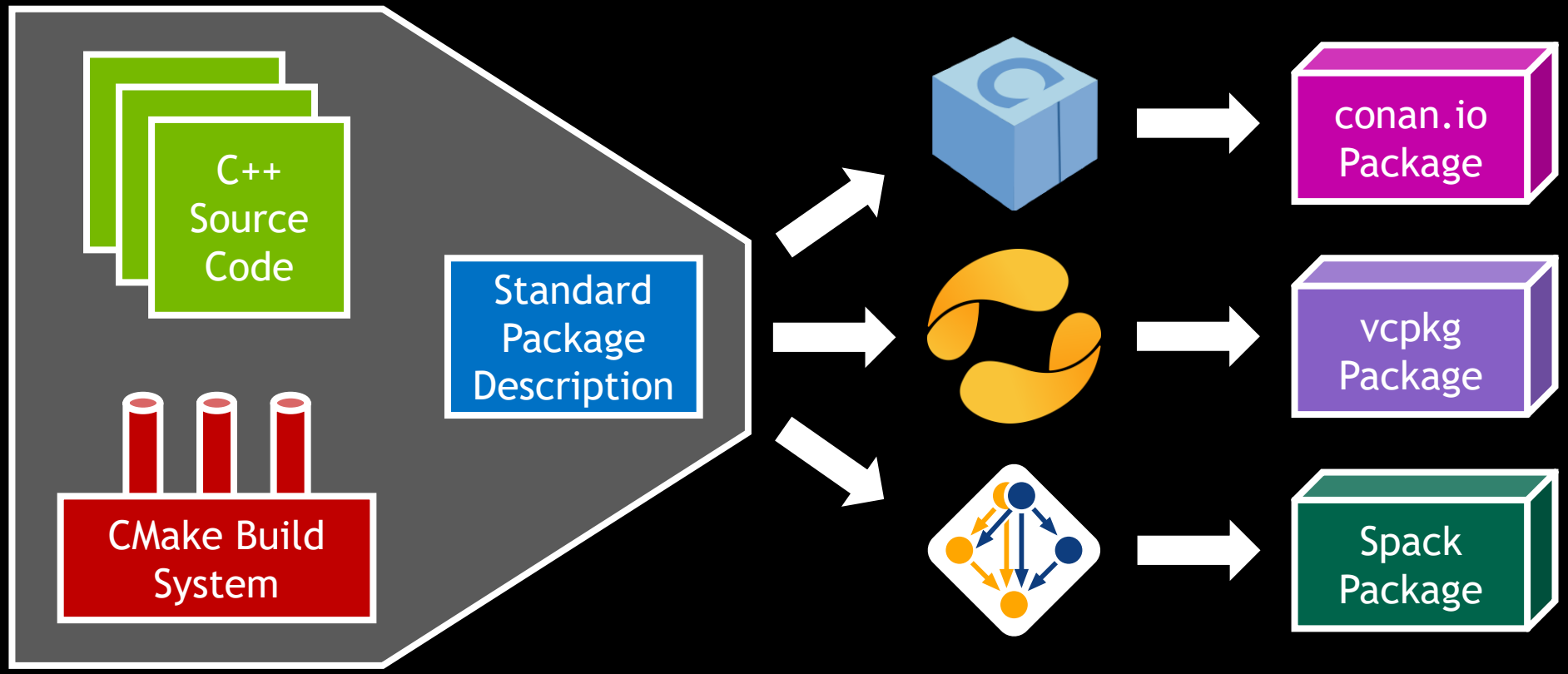


Source: jetbrains.com/lp/devecosystem-2020/cpp, etc

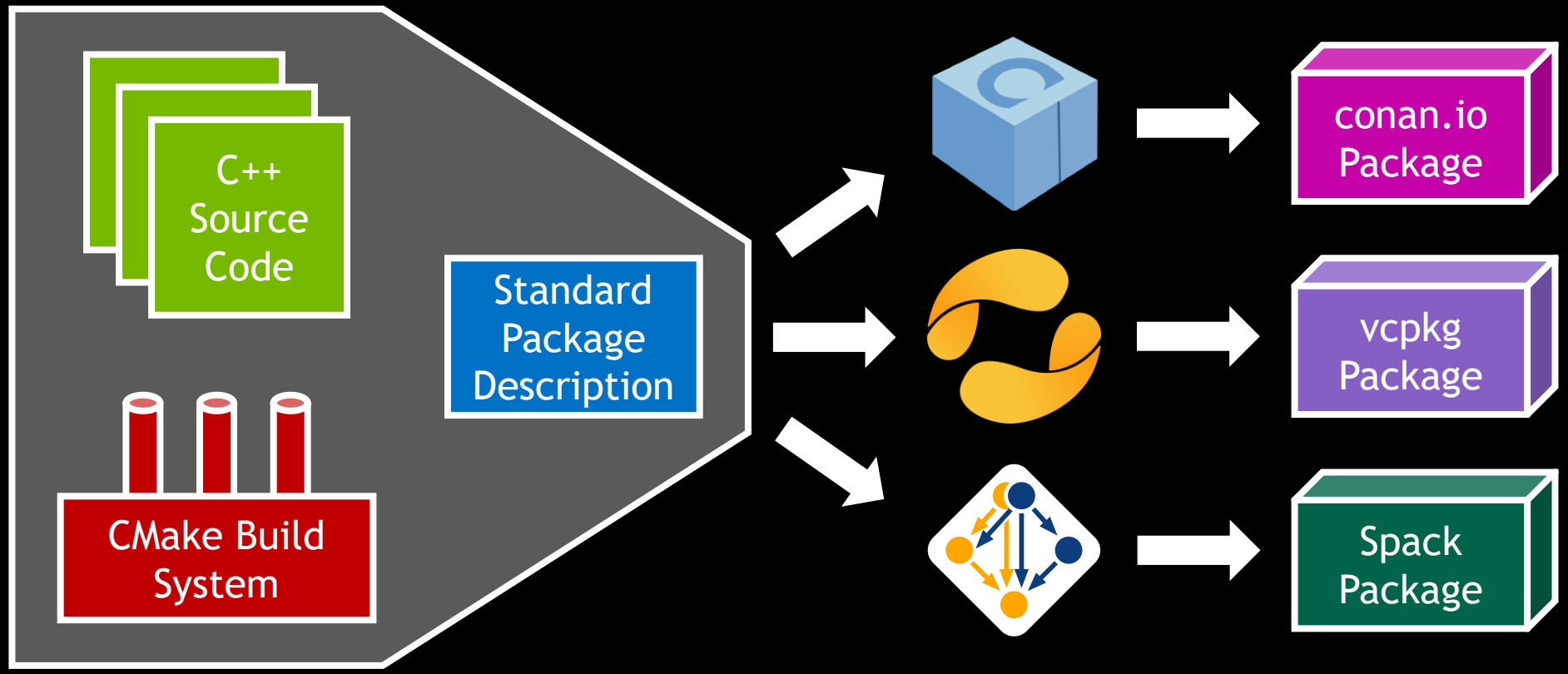
Standardizing C++ Package Management



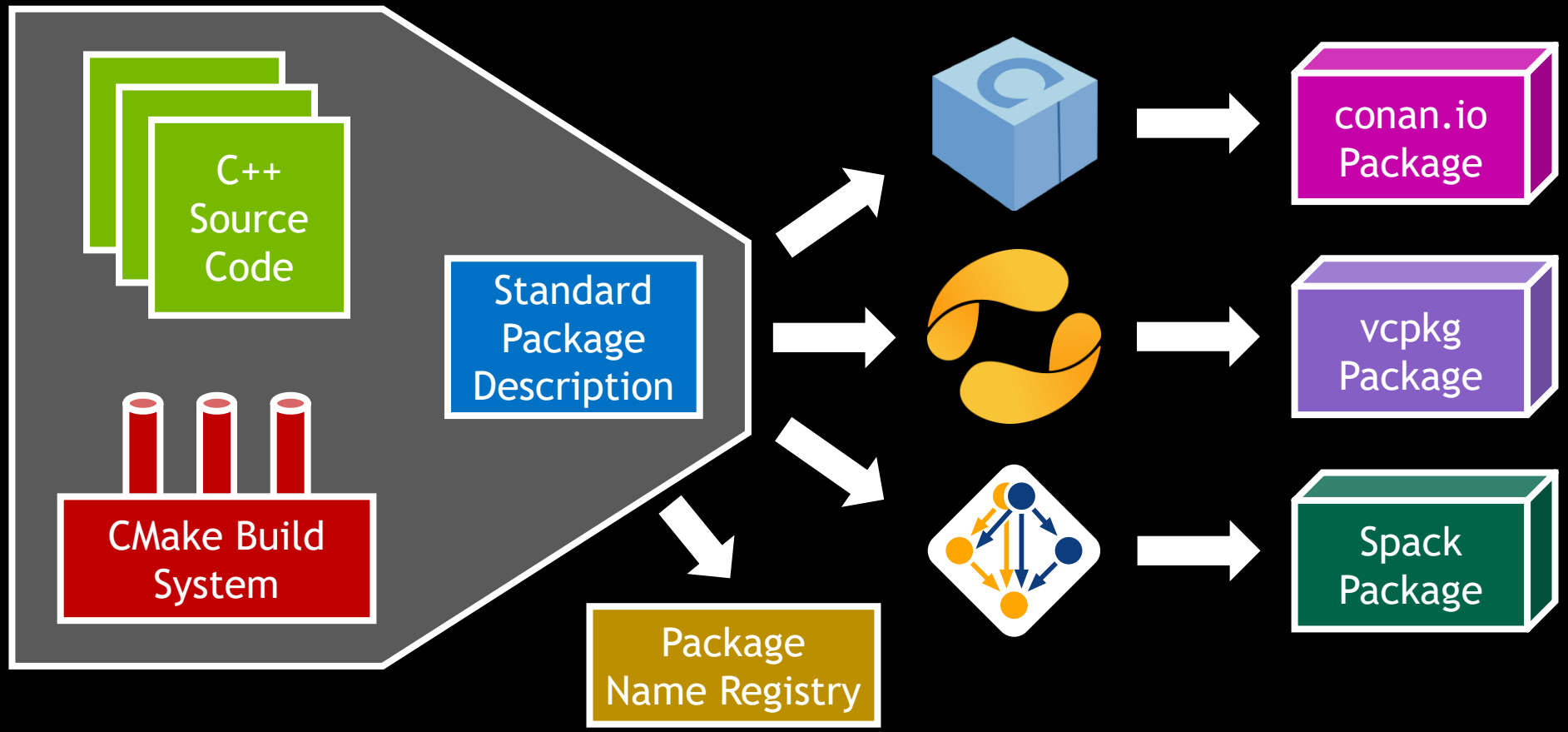
Standardizing C++ Package Management



Standardizing C++ Package Management



Standardizing C++ Package Management



C++ Standard Library

- Comes with toolchain by default.
- Works everywhere.

- Implementations duplicates work.
- 10-20 year stability guarantees.
- Must support a broad range of platforms.

C++ Standard Library

- Comes with toolchain by default.
- Works everywhere.

- Implementations duplicates work.
- 10-20 year stability guarantees.
- Must support a broad range of platforms.

External Libraries

- Single implementation.
- Narrower scope.
- Flexible stability guarantees.

- Must be acquired separately; aren't available by default.

C++ Standard Library	C++ Collections	External Libraries
➢ Comes with toolchain by default. ➢ Works everywhere.	➢ Optionally comes with toolchain. ➢ Single implementation. ➢ Flexible stability guarantees.	➢ Single implementation. ➢ Narrower scope. ➢ Flexible stability guarantees.
➢ Implementations duplicates work. ➢ 10-20 year stability guarantees. ➢ Must support a broad range of platforms.		➢ Must be acquired separately; aren't available by default.

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.

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.

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++ now