Cooperative C++ Evolution

Toward a Typescript for C++

HERB SUTTER

Roadmap

cppfront: Recap

- **Safety** for C++ — 50× — esp. guaranteed program-meaningful initialization
- **Simplicity** for C++ — 10× — esp. for programmers

cppfront: What's new

- Types, reflection, metafunctions, …
- Simplification through generalization

Compatibility for C++

- Why • What kind • What plan

green-field language

invent new idioms/styles

new modules

new ecosystem/packagers

compatibility bridges

From CppCon 2022

refresh C++ itself

make C++ guidance default

make C++ modules default

keep C++ ecosystem/packagers

keep C++ compatibility

also valuable!

this talk

our focus today

From CppCon 2022

What could we do if we had a cleanly demarcated "bubble of new code," via an alternate syntax ***for C++?***

syntax… #2 ?

reduce complexity 10×
increase safety 50×
improve toolability 10×
evolve more freely for another 30 years

What if we could do "***C++11 feels like a new language***" again, for the whole language?

Theme: 100% pure C++… just nicer

Nearly all slide code is **screenshots**

Major improvement via directed evolution

10× simpler — metric: 90% of today's guidance not needed

50× safer — metric: 98% fewer type/bounds/init/lifetime CVEs

Friction-free interop

Always generate 100% C++20/23 code

Use any C++ code directly, no wrapping/marshaling/thunking/…

Easily adoptable

100% binary compatibility always

100% source compatibility always available,
but "zero-overhead" (only pay for it if you use it)

"write 1 line in any C++ file and start seeing benefit"

Make it easy to do what we already teach

Defaults are what we already teach programmers to do

Diagnostics flag what we already teach not do + what to do instead

Caveats

From CppCon 2022

My personal experiment

My hope: To start a conversation about what could be possible ***within C++***'s own evolution to rejuvenate C++

cppfront

Cpp2 → Cpp1 compiler

Thank you!

github.com/hsutter/ **cppfront**

> 700 issues and PRs

> 120 contributors

Joe Abbate, Abhinav00, Robert Adam, Adam, Konstantin Akimov, Aaron Albers, Alex, Federico Aponte, Graham Asher, Ashmate, Peter Barnett, Sean Baxter, Jo Bates, Joscha Benz, Jan Bielak, Borys Boiko, Simon Buchan, Michael Clausen, Michał Cichoń, cpp-niel, crim4, ct-clmsn, Joshua Dahl, Dylam de la Torre, Denis, Matthew Deweese, dmicsa, dobkeratops, Deven Dranaga, Anton Dyachenko, Konstantin F, farmerpiki, Igor Ferreira, Stefano Fiorentino, fknauf, Robert Fry, Artie Fuffkin, Gabriel Gerlero, Jaroslaw Glowacki, Matt Godbolt, William Gooch, Víctor M. González, Terence J. Grant, GrigorenkoPV, Piotr Halas, HALL9kv0, Morten Hattesen, Neil Henderson, Michael Hermier, h-vetinari, ILoveGoulash, Stefan Isak, Tim Keitt, Vanya Khodor, Hugo Lindström, Ferenc Nandor Janky, jazn, jgarvin, Dominik Kaszewski, kelbon, Marek Knápek, Emilia Kond, Vladimir Kraus, Ahmed Al Lakani, Junyoung Lee, Greg Marr, megajocke, mike919192, Uenal Mutlu, Thomas Neumann, Sebastian Nibisz, Niel, Jim Northrup, Daniel Oberhoff, orent, Jussi Pakkanen, PaTiToMaSteR, Slobodan Pavlic, Johel Ernesto Guerrero Peña, Bastien Penavayre, Daniel Pfeifer, Piotr, Davide Pomi, Andrei Rabusov, rconde01, realgdman, Alex Reinking, Pierre Renaux, Raffaele Rialdi, Alexey Rochev, RPeschke, Sadeq, Max Sagebaum, Filip Sajdak, satu, Tobias Schlüter, SecureFromScratch, Wolf Seifert, shemeshg, Tor Shepherd, Luke Shore, Nick Smith, Francis Grizzly Smit, Zenja Solaja, Soraphis, Sören Sprößig, Benjamin Summerton, Hypatia of Sva, SwitchBlade, Paweł Syska, Nigel Tao, Ramy Tarchichy, tkielan, Marzo Sette Torres Junior, Nick Treleaven, Jan Tusil, Shizeeg Unadequatov, userxfce, Ariane van der Steldt, Ezekiel Warren, Kayla Washburn, Tyler Weaver, Will Wray

Implemented and live-demo'd last time

Safety (goal: 50× fewer CVEs due to type, bounds, lifetime, and init safety)

- **Bounds and null checking by default,** incl. when using existing types from Cpp2
- **Guaranteed initialization-before use** with program-meaningful values
- And: **contracts** (`pre`, `post`, `assert`), default **`const`**, default **nodiscard**, default `new` is **make_unique**, no pointer math, …

Simplicity (goal: 10× less to know)

- Context-free, order independent
- **"Zero-overhead" backward compatibility**
- Declarative parameter passing, multiple/named return values
- Unified safe conversions (`is`, `as`) and pattern matching (`inspect`)
- Unified function call: `x.f(y)` can use `f(x,y)`
- Uniform capture: lambdas, contracts, strings

and propose them (with today's syntax) to the ISO C++ committee and C++ conferences

Lifetime	gc_arena	<=>	Reflection & metaclasses	Value-based exceptions	Parameter passing	Patmat using `is` and `as`
P1179		P0515	P0707	P0709	d0708	P2392
CppCon 2015/18	CppCon 2016	CppCon 2017	CppCon 2017/18	CppCon 2019	CppCon 2020	CppCon 2021

Roadmap

cppfront: Recap

- Safety for C++ — 50× — esp. guaranteed program-meaningful initialization
- Simplicity for C++ — 10× — esp. for programmers

cppfront: What's new

- Types, reflection, metafunctions, …
- Simplification through generalization

Compatibility for C++

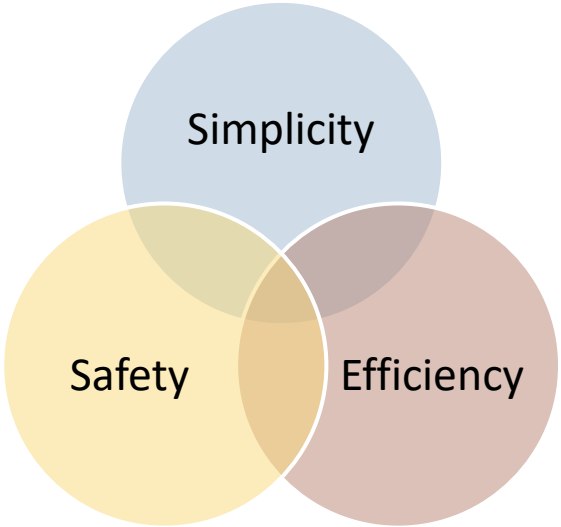
- Why • What kind • What plan

Sometimes you can have it all

These are not always in tension

Judicious abstraction

⇒ **directly express intent**

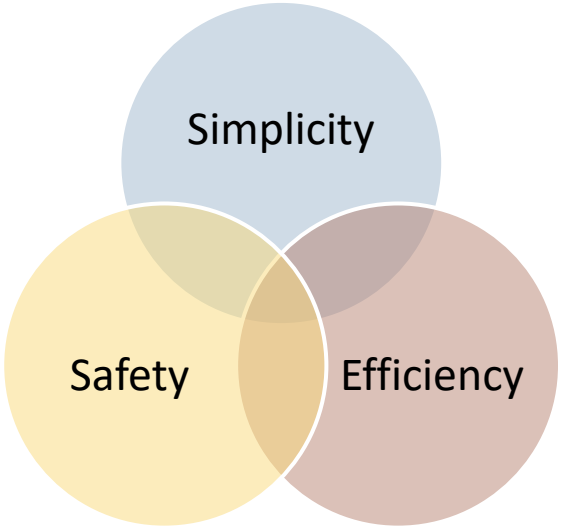


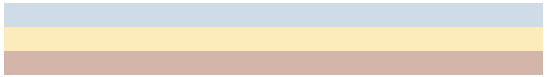
Sometimes you can have it all

These are not always in tension

Judicious abstraction

⇒ **directly express intent**



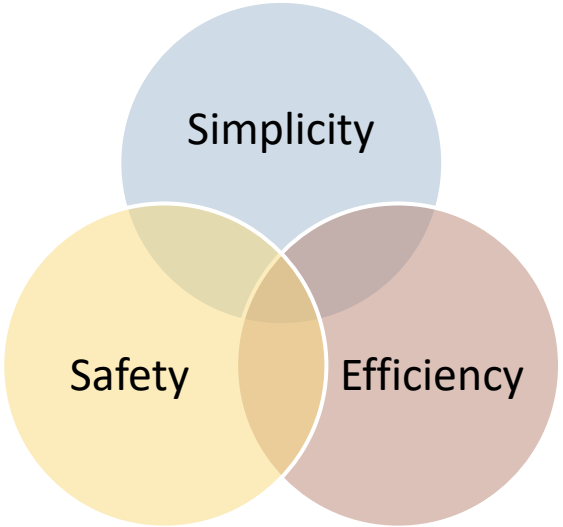


Sometimes you can have it all

These are not always in tension

Judicious abstraction

⇒ **directly express intent**




```
if min <= index < max {
    ...
}
```

Chained comparisons

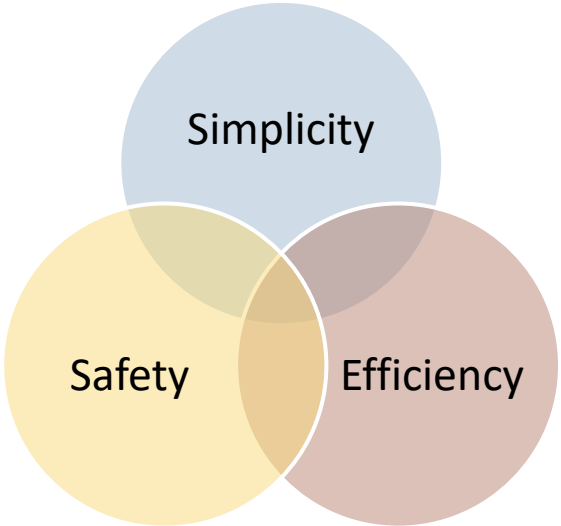
Simple	DRY & say what you mean
Safe	Cpp2 allows mathematically sound (transitive) chains, but not `a >= b < c` or `d != e != f`
Efficient	Single evaluation for all terms

Sometimes you can have it all

These are not always in tension

Judicious abstraction

⇒ **directly express intent**




```
outer: while x>0 next x-- {
    :::
    continue outer;
    :::
}
```

Named break and continue

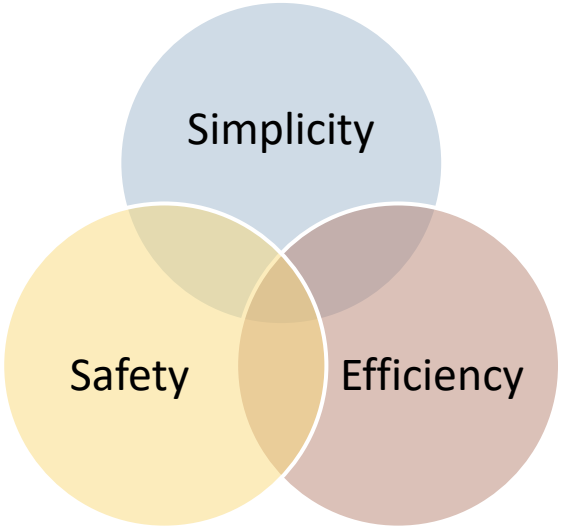
Simple	No extra var, directly expressed
Safe	Structured, reduces demand for (even a structured) `goto`
Efficient	Can't make it faster by hand

Sometimes you can have it all

These are not always in tension

Judicious abstraction

⇒ **directly express intent**




```
main: (args) =
  std::cout <<
    "This exe is (args[0])$";
```

std and main

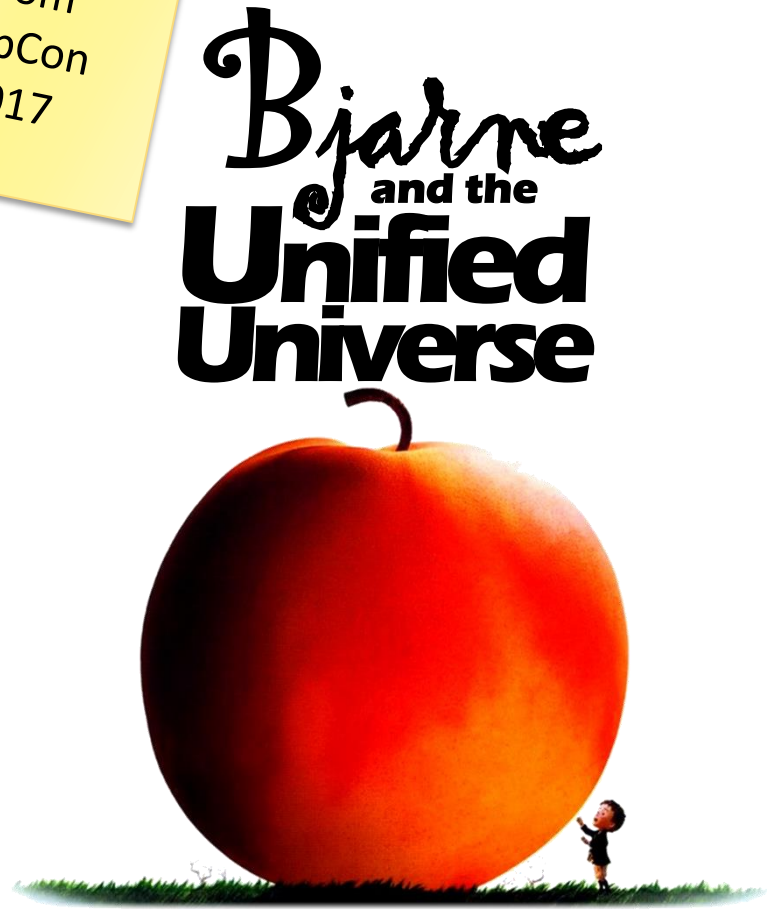
Simple	`std` always available Omit `->int` or `{}` if not needed Convenient string interpolation
Safe	`vector<string_view>` Bounds checking by default
Efficient	"Zero-overhead": pay only for what you use, can't do better by hand (contemplating: `-fno-alloc`)

Now Playing

From CppCon 2017

Now Playing

From CppCon 2017



Now Playing

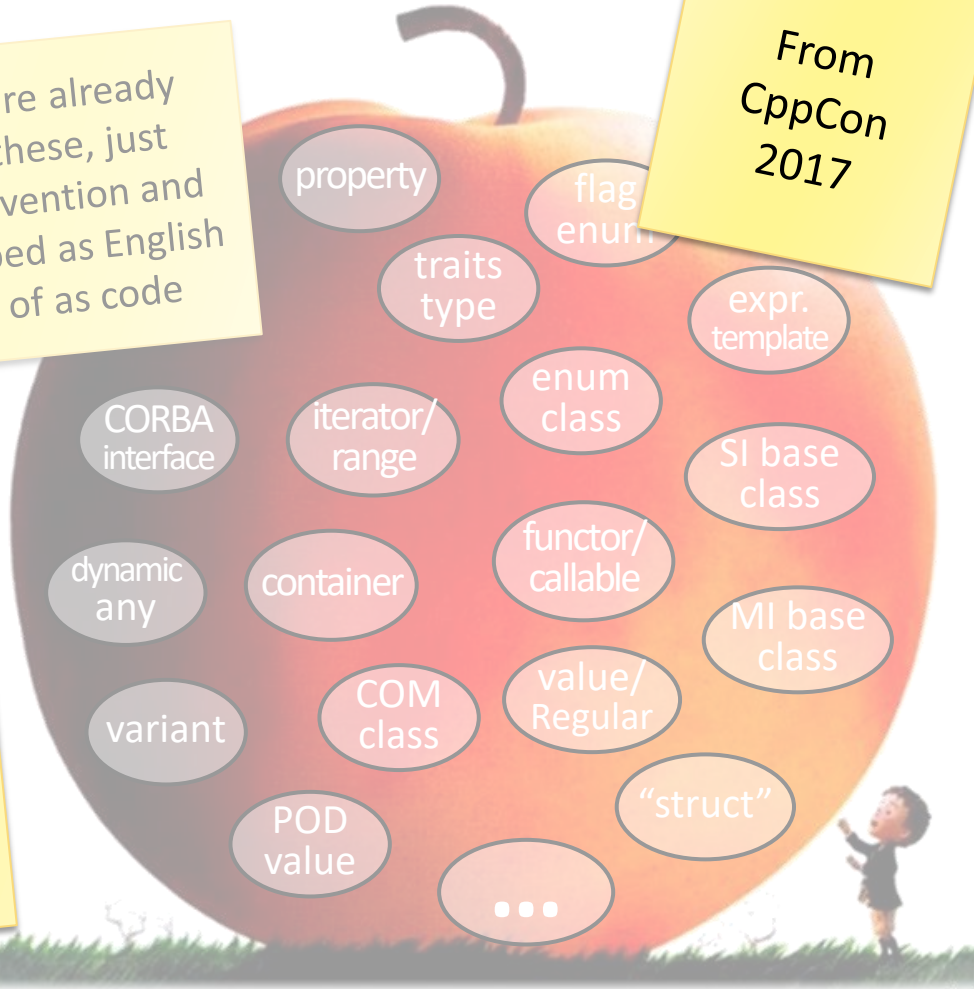
- The C++ type system is unified!

Now Playing

- The C++ type system is unified!

note: we're already writing these, just (a) by convention and (b) described as English instead of as code

From CppCon 2017



Metaclasses goal in a nutshell:

to **name a subset** of the universe of classes having **common characteristics**,

express that subset using **compile-time code**, and

make **classes easier to write** by letting class authors **use the name as a generalized opt-in** to get those characteristics.

The language at work

From CppCon 2017

Source code

```
class Point {
  int x, y;
};
```

```
struct MyClass : Base {
  void f() { /*...*/ }
  // ...
};
```

Compiler

```
for (m : members)
  if (!v.has_access())
    if(is_class())
      v.make_private();
    else // is_struct()
      v.make_public();

for (f : functions) {

  if (f.is_virtual_in_base_class()
      && !f.is_virtual())
    f.make_virtual();

  if (!f.is_virtual_in_base_class()
      && f.specified_override())
    ERROR("does not override");

  if (f.is_destructor())
    if (members_dtors_noexcept())
      f.make_noexcept();

}
```

Definition

```
class Point {
private:
  int x, y;
public:
  Point() =default;
  ~Point() noexcept =default;
  Point(const Point&) =default;
  Point& operator=(const Point&) =default;
  Point(Point&&) =default;
  Point& operator=(Point&&) =default;
};
```

```
class MyClass : public Base {
public:
  virtual void f() { /*...*/ }
  // ...
};
```

The language at work

From CppCon 2017

Source code

```
class Point {
  int x, y;
};
```

```
struct MyClass : Base {
  void f() { /*...*/ }
  // ...
};
```

Compiler

Q: What if you could write your own code here, and give a name to a group of defaults & behaviors?

(treat it as ordinary code, share it as a library, etc.)

Definition

```
class Point {
private:
  int x, y;
public:
  Point() =default;
  ~Point() noexcept =default;
  Point(const Point&) =default;
  Point& operator=(const Point&) =default;
  Point(Point&&) =default;
  Point& operator=(Point&&) =default;
};
```

```
class MyClass : public Base {
public:
  virtual void f() { /*...*/ }
  // ...
};
```

The langua

nothing too crazy!

just participating in interpreting the meaning of definitions

From CppCon 2017

Source code

```
class Point {
  int x, y;
};
```

not making the language grammar mutable

no grammar difference except allowing a metaclass name instead of general “class”

```
struct MyClass : Base {
  void f() { /*...*/ }
  // ...
};
```

could write your own code here, and give a name to a group of defaults & behaviors?

(treat it as ordinary code, share it as a library, etc.)

Definition

```
class Point {
private:
  int x, y;
public:
  Point() =default;
  ~Point() noexcept =default;
  Point(const Point&) =default;
  Point& operator=(const Point&) =default;
  Point(Point&&) =default;
  Point& operator=(Point&&) =default;
};
```

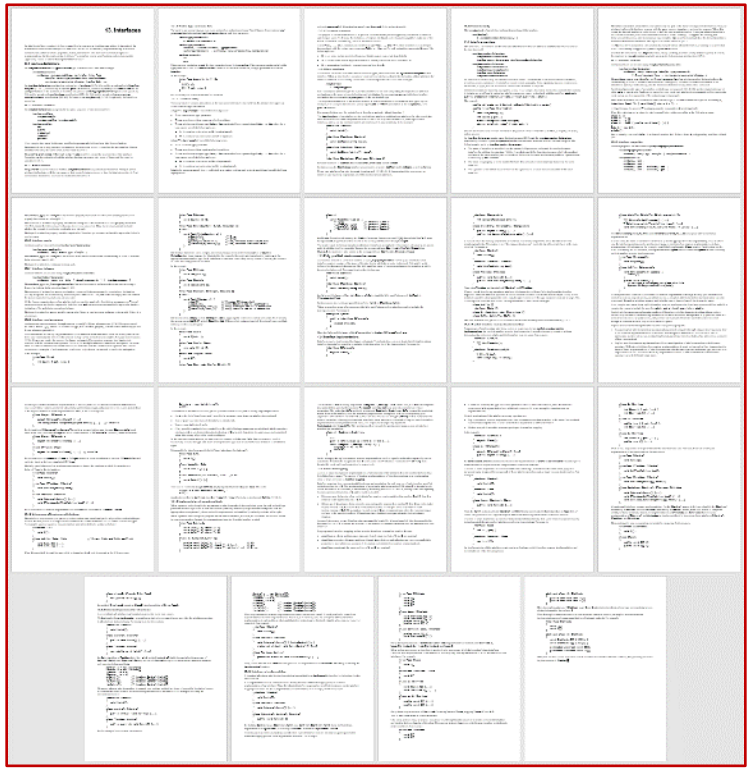
not making definitions mutable after the fact

no difference at all in classes, no bifurcation of the type system

interface (implementation)

From ACCU 2017 and CppCon 2017

C# language: ~18pg, English

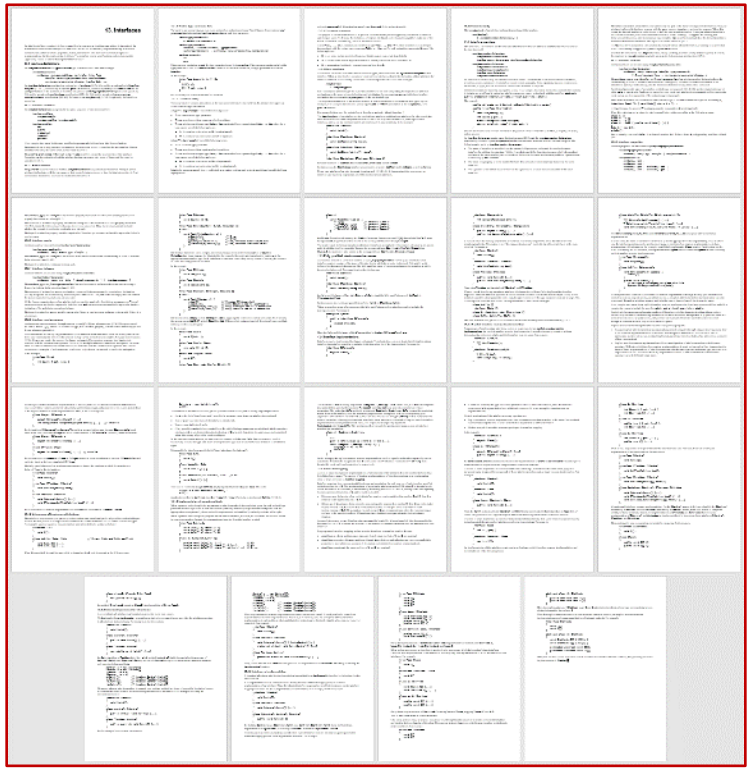


Proposed C++: ~10 lines, testa

```
$class interface {
   ~interface() noexcept { }
   constexpr {
      compiler.require($interface.variables().empty(),
         "interfaces may not contain data members");
      for (auto f : $interface.functions()) {
         compiler.require(!f.is_copy() && !f.is_move(),
             "interfaces may not copy or move; "
             "consider a virtual clone()");
         if (!f.has_access()) f.make_public();
         compiler.require(f.is_public(),
             "interface functions must be public");
         f.make_pure_virtual();
      }
   }
};
```

interface (implementation)

C# language: ~18pg, English



cppfront today

```
interface: (inout t: meta::type_declaration) = {
    has_dtor := false;
    for t.get_members() do (inout m) {
        m.require( !m.is_object(),
                   "interfaces may not contain data ob
        if m.is_function() {
            mf := m.as_function();
            mf.require( !mf.is_copy_or_move(),
                        "interfaces may not copy or mo
            mf.require( !mf.has_initializer(),
                        "interface functions must not
            mf.require( mf.make_public(),
                        "interface functions must be p
            mf.default_to_virtual();
            has_dtor |= mf.is_destructor();
        }
    }
    if !has_dtor { t.add_virtual_destructor(); }
}
```

interface (implementation)

```
interface: (inout t: meta::type_declaration) = {
    has_dtor := false;
    for t.get_members() do (inout m) {
        m.require( !m.is_object(),
                   "interfaces may not contain data objects");
        if m.is_function() {
            mf := m.as_function();
            mf.require( !mf.is_copy_or_move(),
                        "interfaces may not copy or move; consider a virtual clone() instead");
            mf.require( !mf.has_initializer(),
                        "interface functions must not have a function body; remove the '=' initializer");
            mf.require( mf.make_public(),
                        "interface functions must be public");
            mf.default_to_virtual();
            has_dtor |= mf.is_destructor();
        }
    }
    if !has_dtor { t.add_virtual_destructor(); }
}
```

Example from Apr 30 blog post

herbsutter.com/2023/04/30/cppfront-spring-update/

```
// Shape is declaratively an abstract base class having only public
// and pure virtual functions (with "public" and "virtual" applied
// by default if the user didn't write an access specifier on a
// function, because "@interface" explicitly opts in to ask for
// these defaults), and a public pure virtual destructor (generated
// by default if not user-written)... the word "interface" carries
// all that meaning as a convenient and readable opt-in, but
// without hardwiring "interface" specially into the language
//
Shape: @interface type = {
    draw: (this);
    move: (inout this, offset: Point2D);
}
```

Example from Apr 30 blog post

herbsutter.com/2023/04/30/cppfront-spring-update/

```
// Point2D is declaratively a value type: it is guaranteed to have
// default/copy/move construction and <=> std::strong_ordering
// comparison (each generated with memberwise semantics
// if the user didn't write their own, because "@value" explicitly
// opts in to ask for these functions), a public destructor, and
// no protected or virtual functions... the word "value" carries
// all that meaning as a convenient and readable opt-in, but
// without hardwiring "value" specially into the language
//
Point2D: @value type = {
    x: i32 = 0;  // data members (private by default)
    y: i32 = 0;  // with default values
    // ...
}
```

Common metafunctions, implemented from P0707

`interface`	An abstract class having only pure virtual functions
`polymorphic_base`	A pure polymorphic base type that is not copyable or movable, and whose destructor is either public+virtual or protected+nonvirtual
`ordered`	A totally ordered type with `operator<=>` that implements `std::strong_ordering` Also: `weakly_ordered`, `partially_ordered`
`copyable`	A type that has copy/move construction/assignment
`basic_value`	A `copyable` type that has public default construction and destruction (generated if not user-written) and no protected or virtual functions
`value`	An `ordered basic_value` Also: `weakly_ordered_value`, `partially_ordered_value`
`struct`	A `basic_value` with all public members, no virtual functions, and no user-written `operator=`

and a few more...

Enums: first-class (ordinary) types

C: Special `enum` language type

```
enum job_state {
    idle, running, paused
};
```

C++11: extended `enum` and added `enum class`

Can specify underlying type, enumerators are scoped, no implicit conversion

```
enum class job_state {
    idle, running, paused
};
```

Enums: first-class (ordinary) types

C: Special `enum` language type

```
enum job_state {
    idle, running, paused
};
```

C++11: extended `enum` and added `enum class`

- Can specify underlying type, enumerators are scoped, no implicit conversion

```
enum class job_state {
    idle, running, paused
};
```

Cpp2: `enum` metafunction on an ordinary `type`

- **Compile-time library, not baked-in → a more powerful language can be simpler**
- Metafunction can: apply defaults, enforce requirements, and generate code
- Same safeties as `enum class` + **only valid values**
- Automates what we already teach today

```
job_state: @enum type = {
    idle; running; paused;
}
```

Enums: first-class (ordinary) types

C: Special `enum` language type

```
enum job_state {
    idle = 10, running, paused
};
```

C++11: extended `enum` and added `enum class`

- Can specify underlying type, enumerators are scoped, no implicit conversion

```
enum class job_state : long long {
    idle = 10, running, paused
};
```

Cpp2: `enum` metafunction on an ordinary `type`

- **Compile-time library, not baked-in → a more powerful language can be simpler**
- Metafunction can: apply defaults, enforce requirements, and generate code
- Same safeties as `enum class` + **only valid values**
- Automates what we already teach today

```
job_state: @enum<u64> type = {
    idle := 10; running; paused;
}
```

How to use enums as flags in C++?

Asked 14 years ago Modified 2 months ago Viewed 230k times

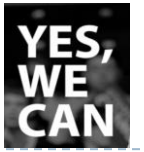
242

Treating `enum`s as flags works nicely in C# via the `[Flags]` attribute, but what's the best way to do this in C++?

For example, I'd like to write:

```
enum AnimalFlags
{
    HasClaws = 1,
    CanFly =2,
    EatsFish = 4,
    Endangered = 8
};

seahawk.flags = CanFly | EatsFish | Endangered;
```



How to use enums as flags in C++?

Asked 14 years ago Modified 2 months ago Viewed 230k times

242

Treating `enum`s as flags works nicely in C# via the `[Flags]` attribute, but what's the best way to do this in C++?

For example, I'd like to write:

```
enum AnimalFlags
{
    HasClaws = 1,
    CanFly =2,
    EatsFish = 4,
    Endangered = 8
};

seahawk.flags = CanFly | EatsFish | Endangered;
```

```
AnimalFlags: @flag_enum<u16> type = {
    HasClaws;    // 1
    CanFly;      // 2
    EatsFish;    // 4
    Endangered; // 8
    RareWaterbird := CanFly | EatsFish | Endangered;
}

main: () = std::cout << AnimalFlags::RareWaterbird;
```

```
(CanFly, EatsFish, Endangered, RareWaterbird)
```

From the `basic_enum` metafunction

```
(value: std::string = "-1")
for  t.get_members()
do   (m)
if   m.is_member_object()
{
    m.require( m.is_public() || m.is_default_access(),
               "an enumerator cannot be protected or private");

    mo := m.as_object();
```

```
    e: value_member_info = ( mo.name() as std::string, "", value );
    enumerators.push_back( e );

    mo.mark_for_removal_from_enclosing_type();
}

//  Compute the default underlying type, if it wasn't explicitly specified
```

```
t.remove_marked_members();

//  Generate all the common material: value and common functions
t.add_member( "    _value             : (underlying_type)$;");
t.add_member( "    private operator=  : (implicit out this, val: i64) == _value = cpp2::unsafe_narrow<
t.add_member( "    get_raw_value      : (this) -> (underlying_type)$ == _value;");
t.add_member( "    operator=          : (out this, that) == { }");
t.add_member( "    operator<=>        : (this, that) -> std::strong_ordering;");

//  Generate the bitwise operations and 'none' value, if appropriate
if bitwise {
    e: value_member_info = ( "none", "", "0");
    enumerators.push_back( e );

    t.add_member( "    operator|=: ( inout this, that )                == _value |= that._value;");
    t.add_member( "    operator&=: ( inout this, that )                == _value &= that._value;");
    t.add_member( "    operator^=: ( inout this, that )                == _value ^= that._value;");
    t.add_member( "    operator| : (       this, that ) -> (t.name())$  == _value |  that._value;");
    t.add_member( "    operator& : (       this, that ) -> (t.name())$  == _value &  that._value;");
    t.add_member( "    operator^ : (       this, that ) -> (t.name())$  == _value ^  that._value;");
    t.add_member( "    has       : ( inout this, that ) -> bool         == _value &  that._value;");
    t.add_member( "    set       : ( inout this, that )                == _value |= that._value;");
    t.add_member( "    clear     : ( inout this, that )                == _value &= that._value~;");
}
```

```
t.add_member( "    operator=        : (out this, that) == { }");
t.add_member( "    operator<=>      : (this, that) -> std::strong_ordering;");

//  Generate the bitwise operations and 'none' value, if appropriate
if bitwise {
    e: value_member_info = ( "none", "", "0");
    enumerators.push_back( e );

    t.add_member( "    operator|=: ( inout this, that )                == _value |= that._value;");
    t.add_member( "    operator&=: ( inout this, that )                == _value &= that._value;");
    t.add_member( "    operator^=: ( inout this, that )                == _value ^= that._value;");
    t.add_member( "    operator| : (       this, that ) -> (t.name())$  == _value |  that._value;");
    t.add_member( "    operator& : (       this, that ) -> (t.name())$  == _value &  that._value;");
    t.add_member( "    operator^ : (       this, that ) -> (t.name())$  == _value ^  that._value;");
    t.add_member( "    has       : ( inout this, that ) -> bool         == _value &  that._value;");
    t.add_member( "    set       : ( inout this, that )                == _value |= that._value;");
    t.add_member( "    clear     : ( inout this, that )                == _value &= that._value~;");
}

//  Add the enumerators
for enumerators do (e) {
    t.add_member( "    (e.name)$ : (t.name())$ == (e.value)$;");
}

//  Provide a 'to_string' function to print enumerator name(s)
(to_string: std::string = "    to_string: (this) -> std::string = { \n")
{
```

But this is all the user had to write, they could **directly express intent**

```
AnimalFlags: @flag_enum<u16> type = {
    HasClaws;    // 1
    CanFly;      // 2
    EatsFish;    // 4
    Endangered; // 8
    RareWaterbird := CanFly | EatsFish | Endangered;
}

main: () = std::cout << AnimalFlags::RareWaterbird;
```

```
(CanFly, EatsFish, Endangered, RareWaterbird)
```

Names are powerful:
To write a metafunction is to write a disciplined Word of Power for a set of defaults, requirements, and generated members

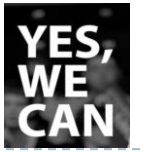
Applying one is always opt-in, we never need to “=delete” what it did

Names are powerful:
To write a metafunction is to write a disciplined Word of Power for a set of defaults, requirements, and generated members

Names are powerful:
To write a class is to write a disciplined Word of Power for a set of encapsulated state and interface behaviors

Names are powerful:
To write a function is to write a disciplined Word of Power for a set of encapsulated processing and interface signature

Names are powerful:
To write a variable is to write a disciplined Word of Power for a set of states and behaviors



Can a C++ enum class have methods?

Asked 9 years, 8 months ago Modified 9 months ago

Viewed 153k times

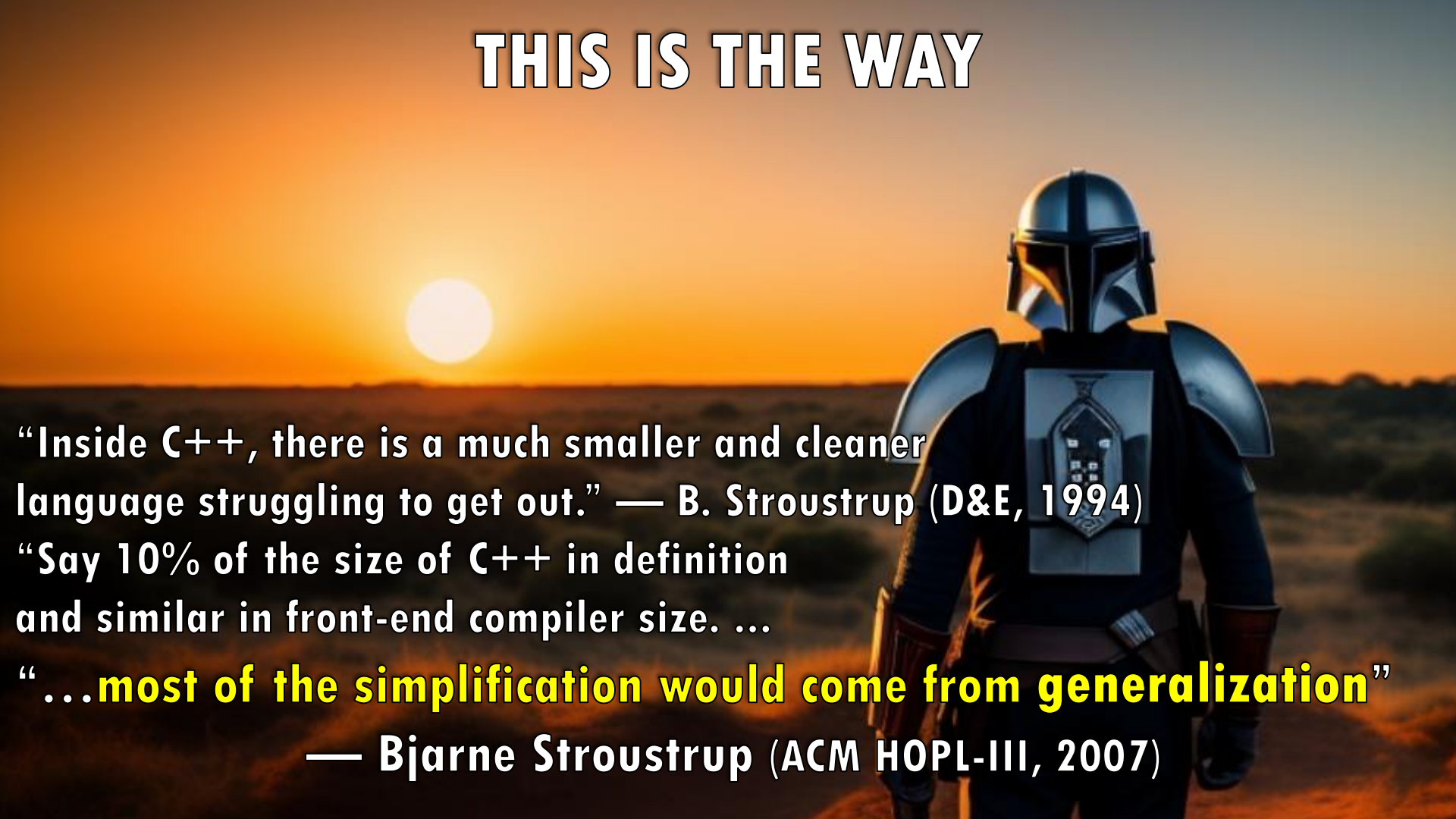
233

I have an enum class with two values, and I want to create a method which receives a value and returns the other one. I also want to maintain type safety(that's why I use enum class instead of enums).

```
janus: @enum type = {
    past;
    future;

    flip: (inout this) = {
        if this == past { this = future; }
        else { this = past; }
    }
}
```

THIS IS THE WAY



"Inside C++, there is a much smaller and cleaner language struggling to get out." — B. Stroustrup (D&E, 1994)

"Say 10% of the size of C++ in definition and similar in front-end compiler size. ...

"...most of the simplification would come from generalization"

— Bjarne Stroustrup (ACM HOPL-III, 2007)

(Safe) unions: first-class (ordinary) types

C: Special unsafe `union` language type

But: Everything is **unsafe**, roll your own enum tags.

```
enum S_tag { Char, Int, String };
union S {
    char c;
    int i;
    char const* s;
};

int main() {
    S     s;
    S_tag s_tag;
    s.i   = 42;
    s_tag = Int;
    std::cout << s.i;
}
```

(Safe) unions: first-class (ordinary) types

C: Special unsafe `union` language type

C++17: Safer `variant` library type

- But: Everything is **unnamed** → harder to use, and can't distinguish repeated types

```
struct S {
    enum { Char, Int, String } tag;
    std::variant<
        char,
        int,
        std::string
    > value;
};

int main() {
    S s;
    s.value = 42;
    std::cout << std::get<S::Int>(s.value);
}
```

(Safe) unions: first-class (ordinary) types

C: Special unsafe `union` language type

C++17: Safer `variant` library type

- But: Everything is unnamed → harder to use, and can't distinguish repeated types

Cpp2: `union` metafunction on an ordinary `type`

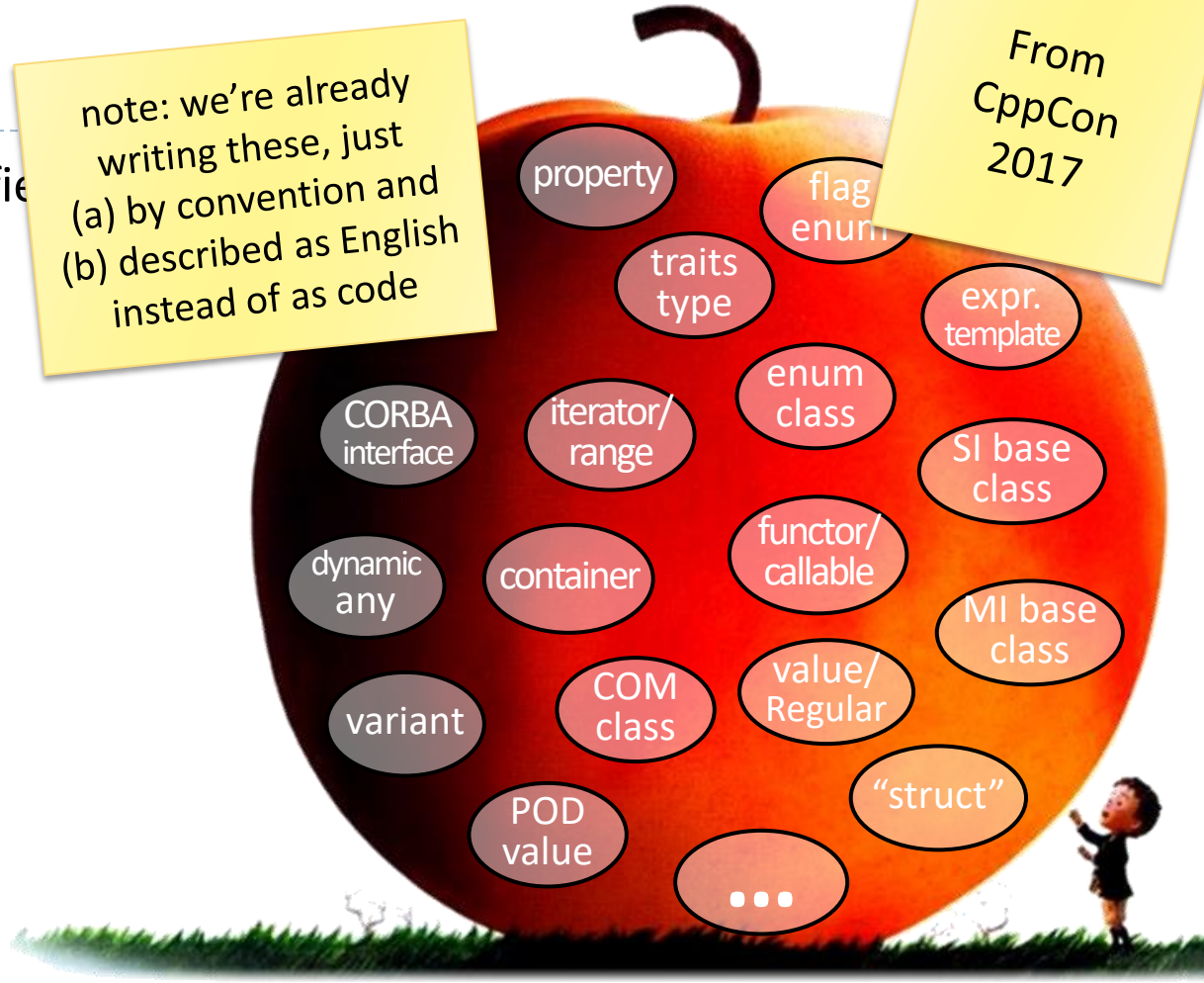
- **Compile-time library, not baked-in → a more powerful language can be simpler**
- Metafunction can: apply defaults, enforce requirements, and generate code
- **Same usability as `union`, same safety as `variant`**
- Automates what we already teach today

```
S: @union type = {
    c: char;
    i: int;
    s: std::string;
}

main: () = {
    s: S = ();
    s.set_i( 42 );
    std::cout << s.i();
}
```

Now Playing

The C++ type system is unifie



Roadmap

cppfront: Recap

- **Safety** for C++ — 50× — esp. guaranteed program-meaningful initialization
- **Simplicity** for C++ — 10× — esp. for programmers

cppfront: What's new

- Types, reflection, metafunctions, …
- Simplification through generalization

Compatibility for C++

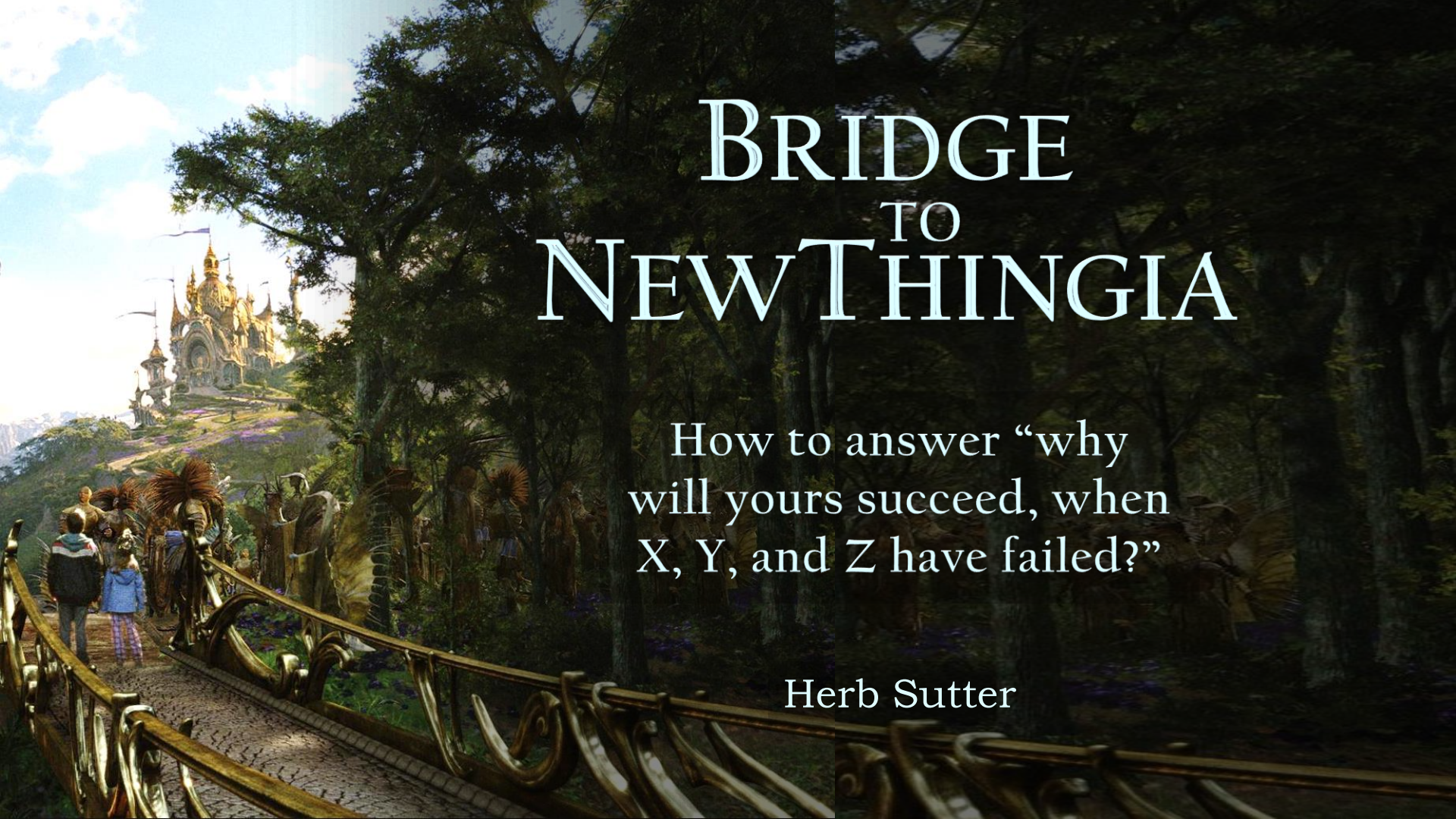
- Why • What kind • What plan

I do believe that there is real value in pursuing functional programming, but ***it would be irresponsible to exhort everyone to abandon their C++ compilers*** *and start coding in Lisp, Haskell, or, to be blunt, any other fringe language.*

To the eternal chagrin of language designers, there are plenty of externalities that can overwhelm the benefits of a language…

We have ***cross platform*** *issues, proprietary* ***tool chains****,* ***certification*** *gates,* ***licensed*** *technologies, and stringent* ***performance*** *requirements on top of the issues with* ***legacy*** *codebases and* ***workforce*** *availability that everyone faces. …*

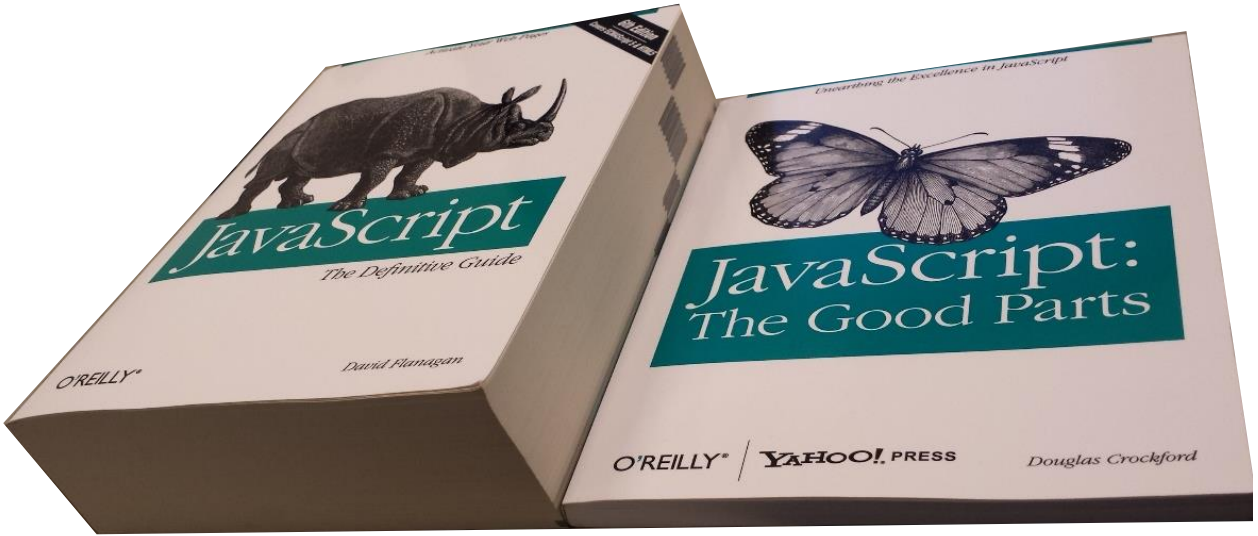
— John Carmack *[**emphasis** added]*


Bridge
to
NewThingia
How to answer “why will yours succeed, when X, Y, and Z have failed?”
Herb Sutter

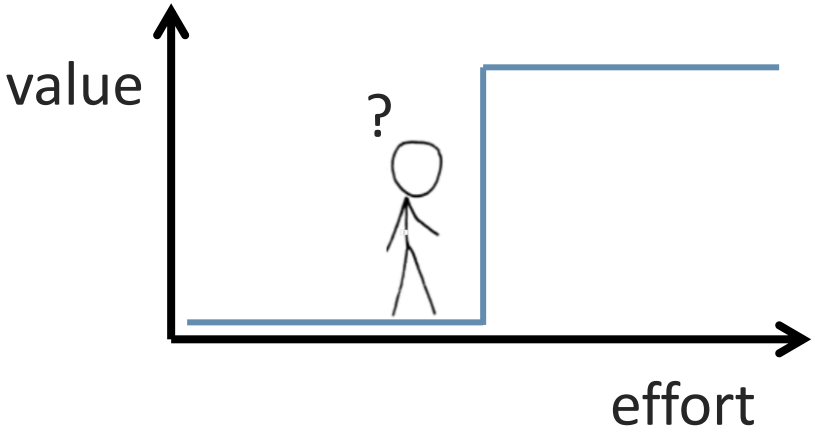
Example: JavaScript

Concise elevator pitch:

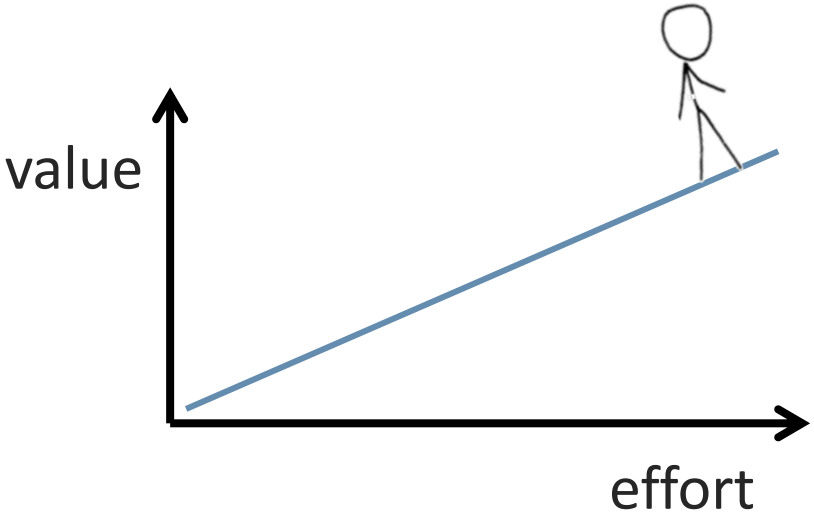
Well-known pain points with JavaScript



Which adoption function would you prefer?



A



B

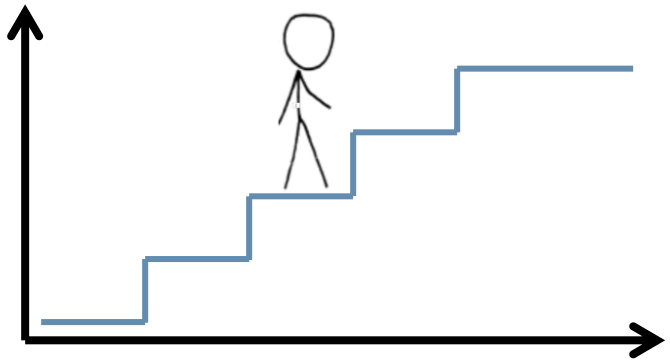
3. NewThing's **compatibility by design**

Basic requirement: **High fidelity interop.**

Min bar: NewThing can seamlessly use OldThing.

Good: "An OldThing project can add NewThing **side by side** and start seeing benefit."

Ex: "Add NewLang file and see benefit."



3. NewThing's **compatibility by design**

Basic requirement: **High fidelity interop.**

Min bar: NewThing can seamlessly use OldThing.

Good: "An OldThing project can add NewThing **side by side** and start seeing benefit."

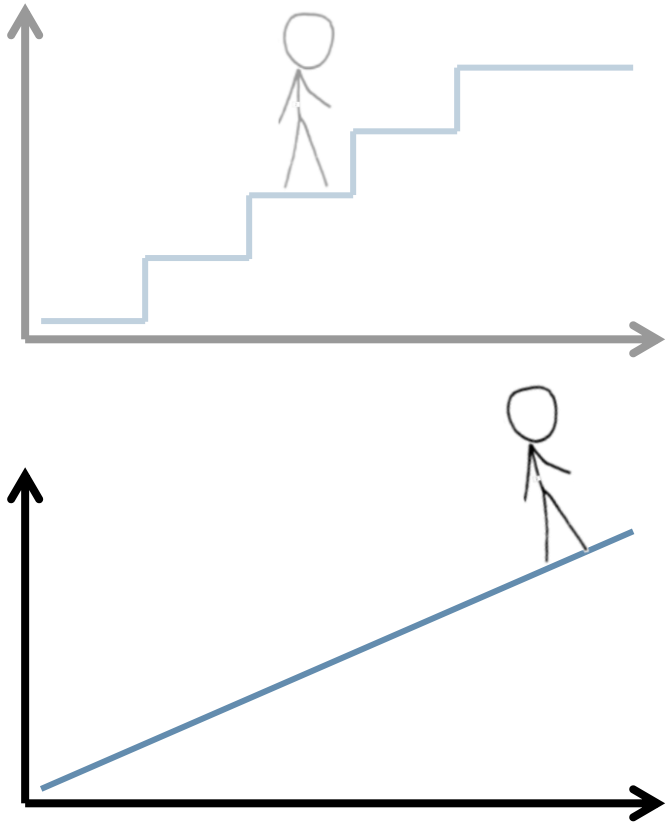
Ex: "Add NewLang file and see benefit."

Grail: "An OldThing project can add NewThing **in one place** and start seeing benefit."

Ex: "Write 1 line of NewLang and see benefit."

1980s: Rename `.c` to `.cpp`, add 1 class, benefit.

2010s: Rename `.js` to `.ts`, add 1 class, benefit.

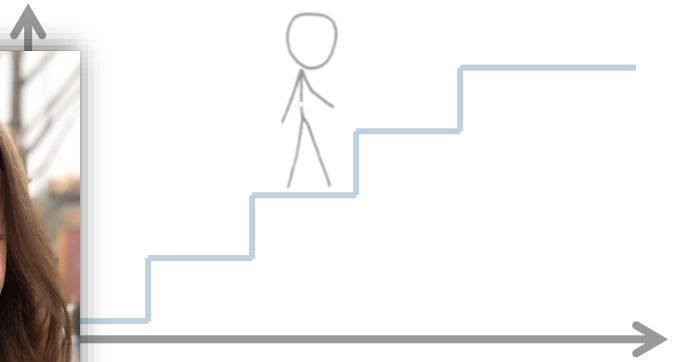


3. NewThing's **compatibility by design**

Basic requirement: High-fi[illegible]rop.

See: **Laura Savino**'s keynote yesterday

the importance of tactical abatement
→ tactical adoption/improvement

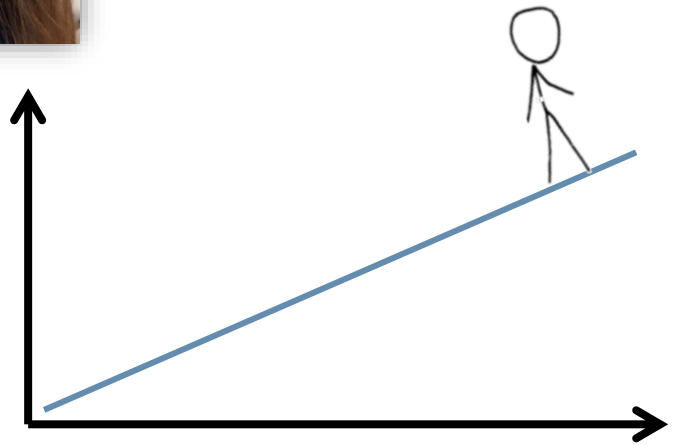


Grail: "An OldThing project can add NewThing **in one place** and start seeing benefit."

Ex: "Write 1 line of NewLang and see benefit."

1980s: Rename `.c` to `.cpp`, add 1 class, benefit.

2010s: Rename `.js` to `.ts`, add 1 class, benefit.




RAMPS ARE GREAT!
THEY'RE NOT JUST FOR OLD FOLKS

3. NewThing's **compatibility by design**

C++: Every C program is a C++ program (still mostly true) + any C++ code can seamlessly call any C + C optimizer+linker.

TypeScript: Every JS program is a TS program + any TS code can seamlessly call any JS code.

TypeScript: Every JS program is a TS program + any TS code can seamlessly call any JS code.

Compatible with JavaScript

Cooperates with JavaScript committee

Contributes proposals to JavaScript

TS features have become Standard JS

TypeScript: Every JS program is a TS program + any TS code can seamlessly call any JS code.

Compatible with JavaScript

Cooperates with JavaScript committee

Contributes proposals to JavaScript

TS features have become Standard JS

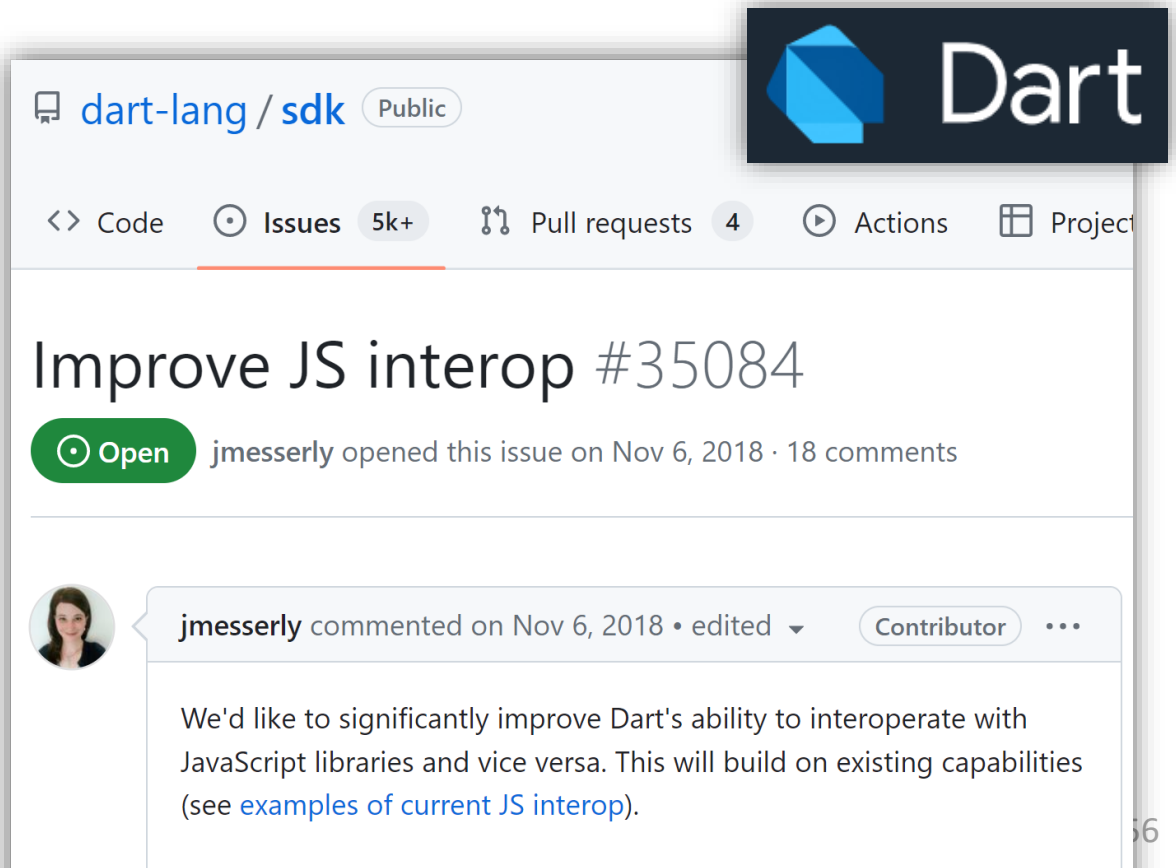
Observation

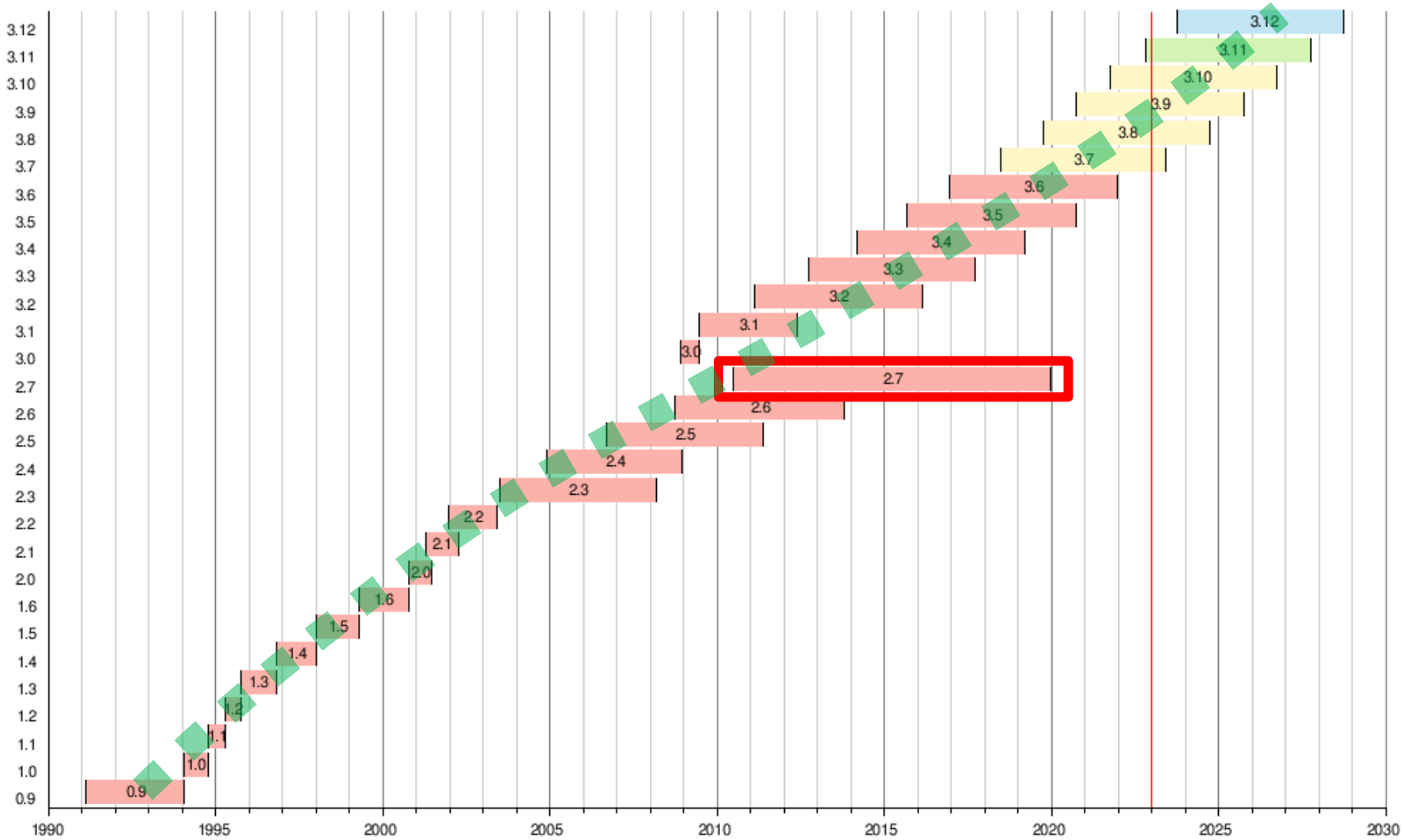
It doesn't seem to me that the TS designers view TS as a successor language

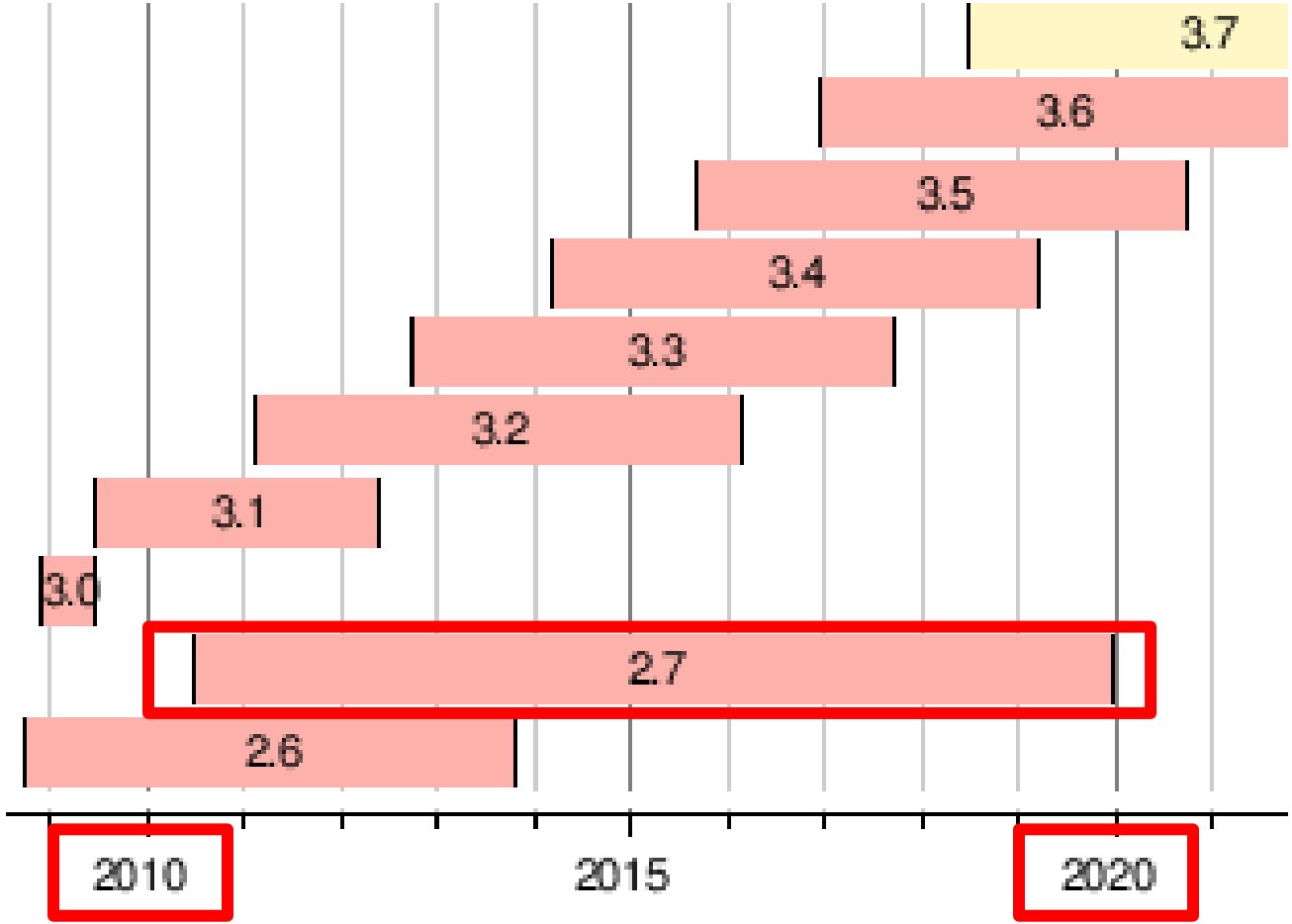
3. NewThing's **compatibility by design**

Compatibility requires strategic up-front design.

Often forgotten until it is too late. Often hard to retrofit.




3.12
3.11
3.10
3.9
3.8
3.7
3.6
3.5
3.4
3.3
3.2
3.1
3.0
2.7
2.6
2.5
2.4
2.3
2.2
2.1
2.0
1.6
1.5
1.4
1.3
1.2
1.1
1.0
0.9
1990
1995
2000
2005
2010
2015
2020
2025
2030


3.7
3.6
3.5
3.4
3.3
3.2
3.1
3.0
2.7
2.6
2010
2015
2020

Python 3

2008: Python 3

Source breaking change (can't compile 2 as 3)

	Python 2	Python 3
`x = 3/2`	`x == 2`	`x == 1.5`

Manual migration + tools (2to3, Pylint, Futurize, Modernize, caniusepython3, tox, mypy)

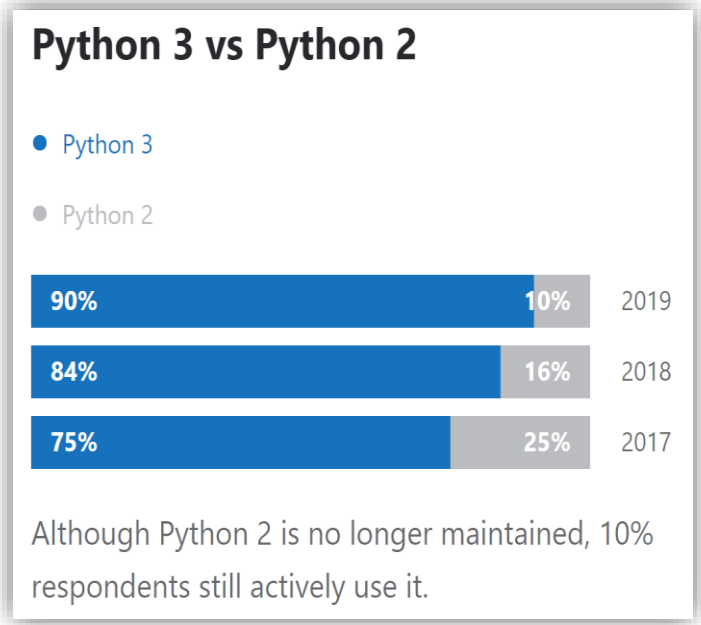
2017: Most Python code still written in "2∩3"

2020: 2.x frozen and unsupported

2023: Still used, CVE backport requests

~12-year transition

vs. 8 years per major version for 1→2→3 (1994→2000→2008)



Source: JetBrains Python Developers Survey (Oct 2019)

Python 3

2008: Python 3

Source breaking change (can't compile 2 as 3)

	Python 2	Python 3
`x = 3/2`	`x == 2`	`x == 1.5`

Manual migration + tools (2to3, Pylint, Futurize, Modernize, caniusepython3, tox, mypy)

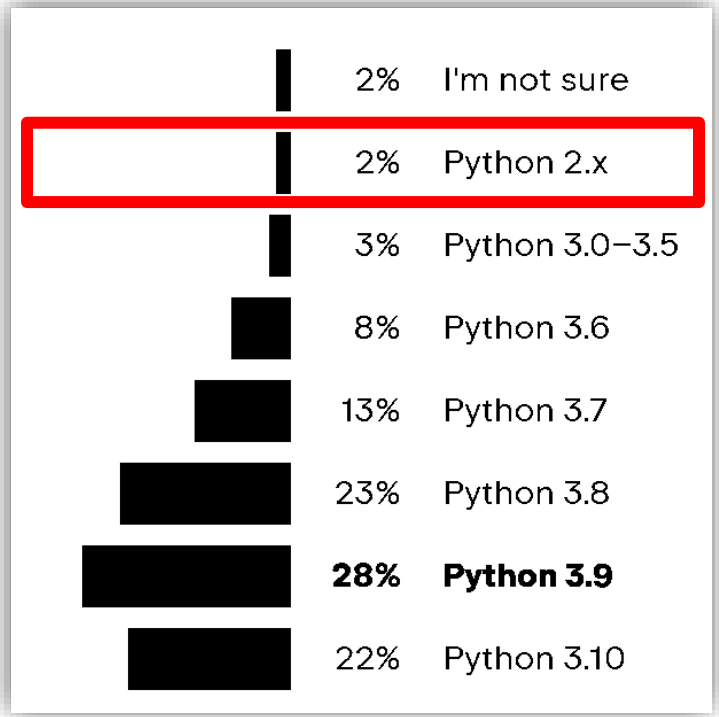
2017: Most Python code still written in "2∩3"

2020: 2.x frozen and unsupported

2023: Still used, CVE backport requests

~12-year transition

vs. 8 years per major version for 1→2→3 (1994→2000→2008)



Source: JetBrains Python Developers Survey (2022)

Python 3

2008: Python 3

Source breaking change (can't compile 2 as 3)

	Python 2	Python 3
`x = 3/2`	`x == 2`	`x == 1.5`

Manual migration + tools (2to3, Pylint, Futurize, Modernize, caniusepython3, tox, mypy)

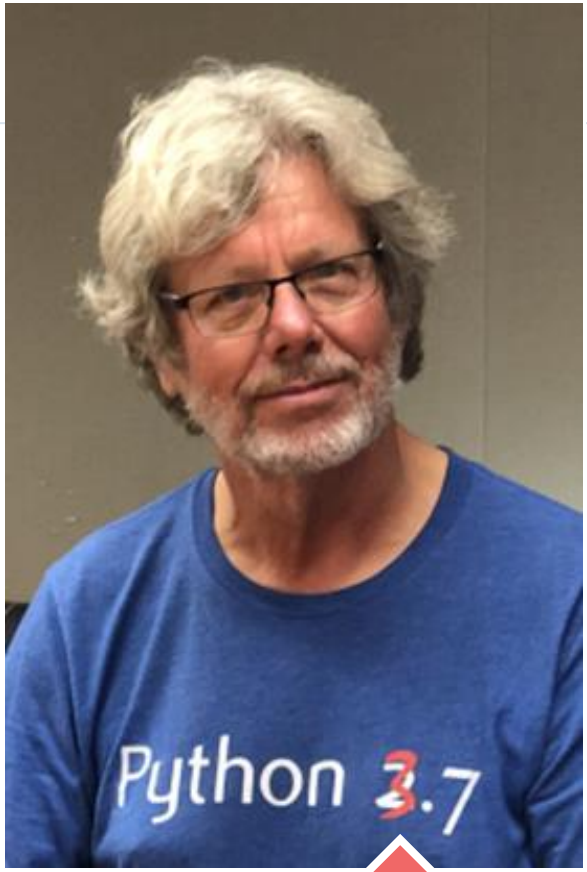
2017: Most Python code still written in "2∩3"

2020: 2.x frozen and unsupported

2023: Still used, CVE backport requests

~12-year transition

vs. 8 years per major version for 1→2→3 (1994→2000→2008)

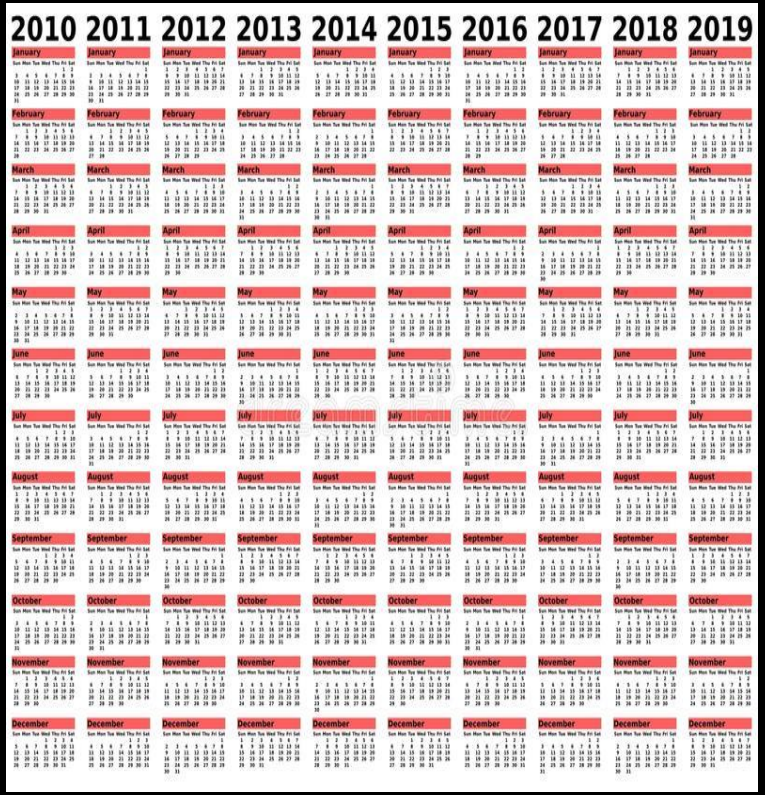


“In hindsight, what would you have done differently?”

“Everything!”

Quick recap: A “**lost decade**” pattern

a visual to illustrate
“a decade is a long time”



Quick recap: A "**lost decade**" pattern

MSVC 6 — ~12 years
- Shipped in **1998**
- "10 is the new 6" fanfare in **2010**

C99 `_Complex` and VLAs — ~12 years
- Added in **1999**
- Walked them back to "optional" in **2011**

C++11 `std::string` — ~11 years
- Banned RC for std::string in **2008/2010**
- Major Linux distro enabled it in **2019**

Python 3 — ~12 years
- Shipped 3.0 in **2008**
- 10% still using 2.x as of early **2020**

If you don't build a strong backward compatibility bridge, expect to slow your adoption down by

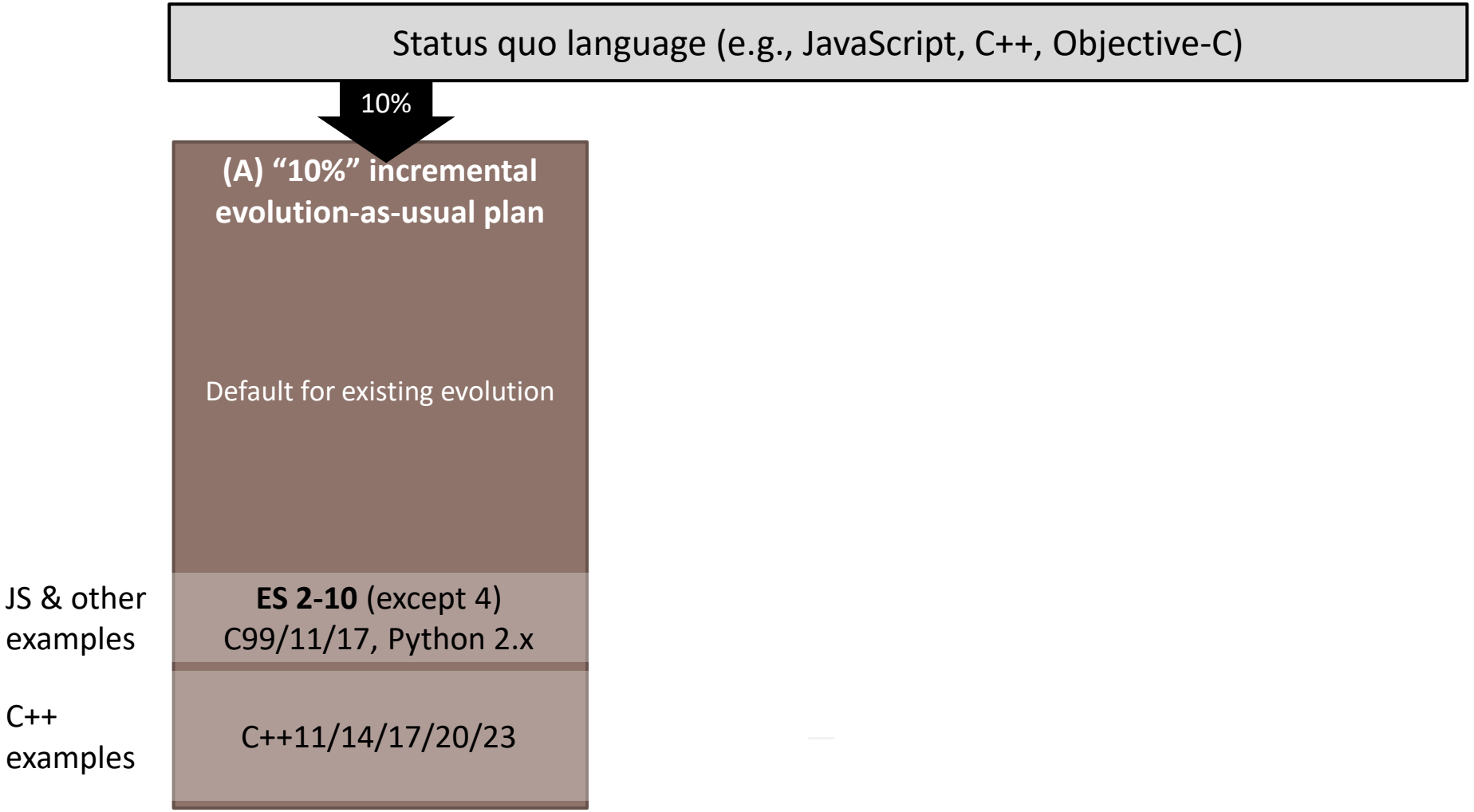
~10 years

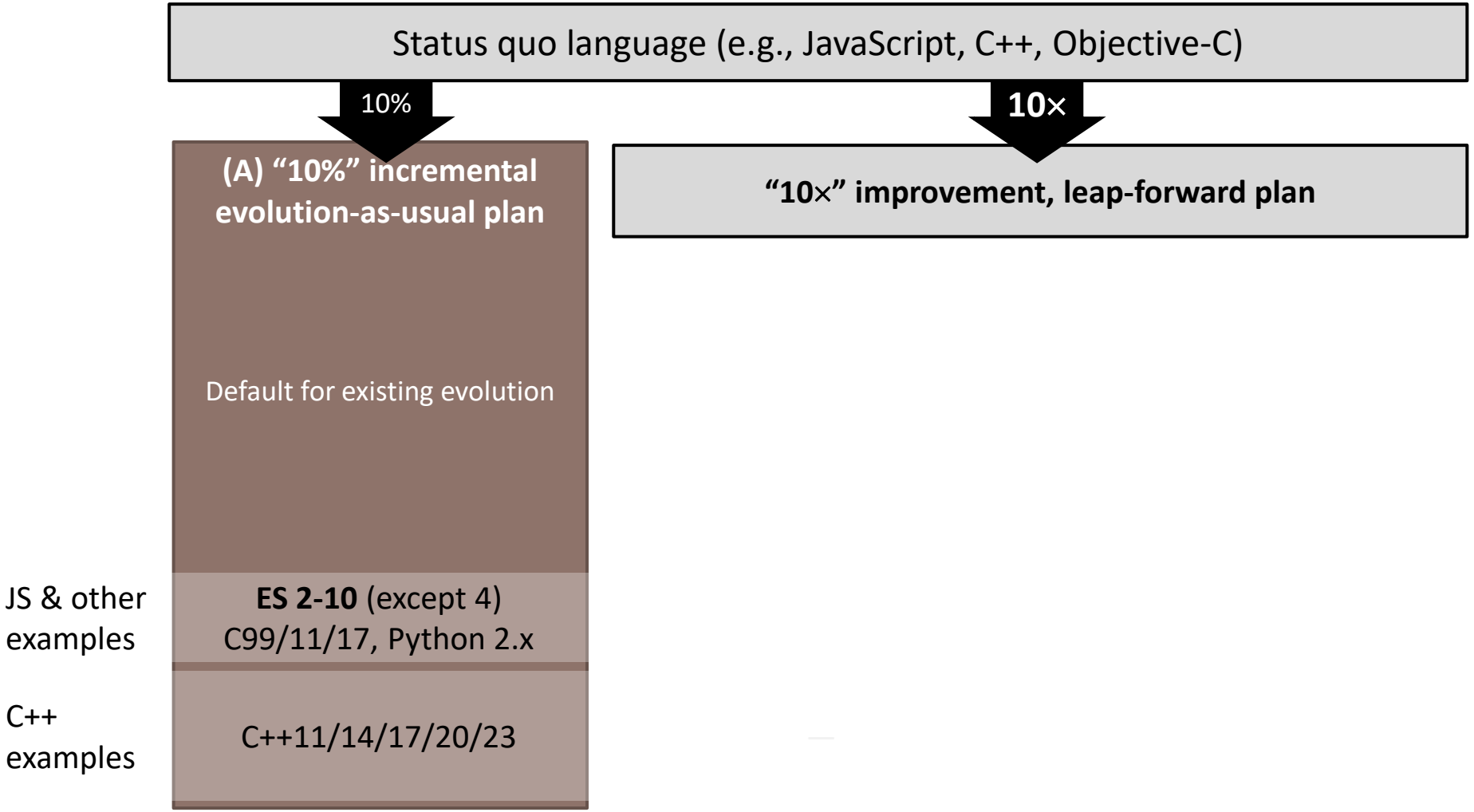
(absent other forces)

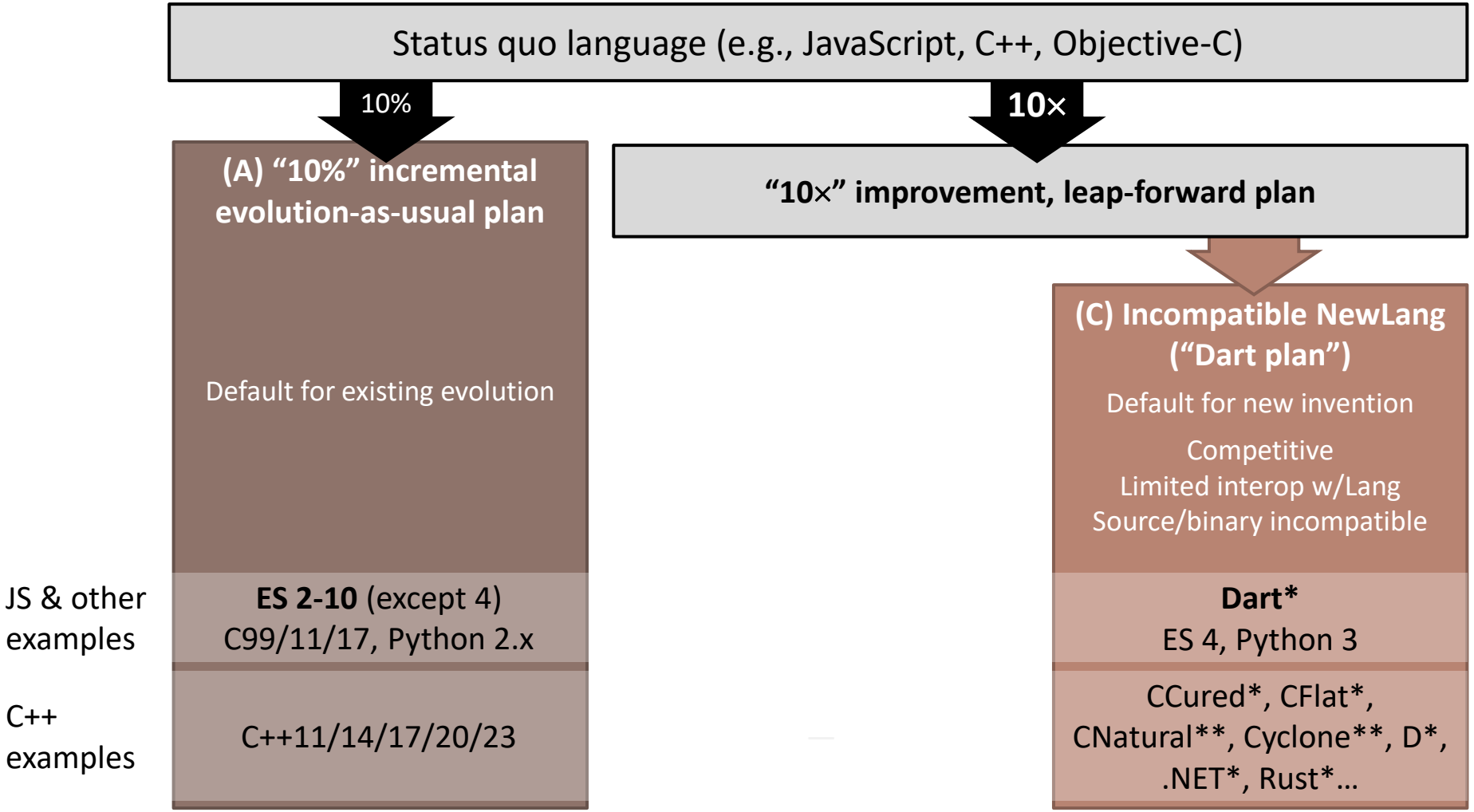
Status quo language (e.g., JavaScript, C++, Objective-C)

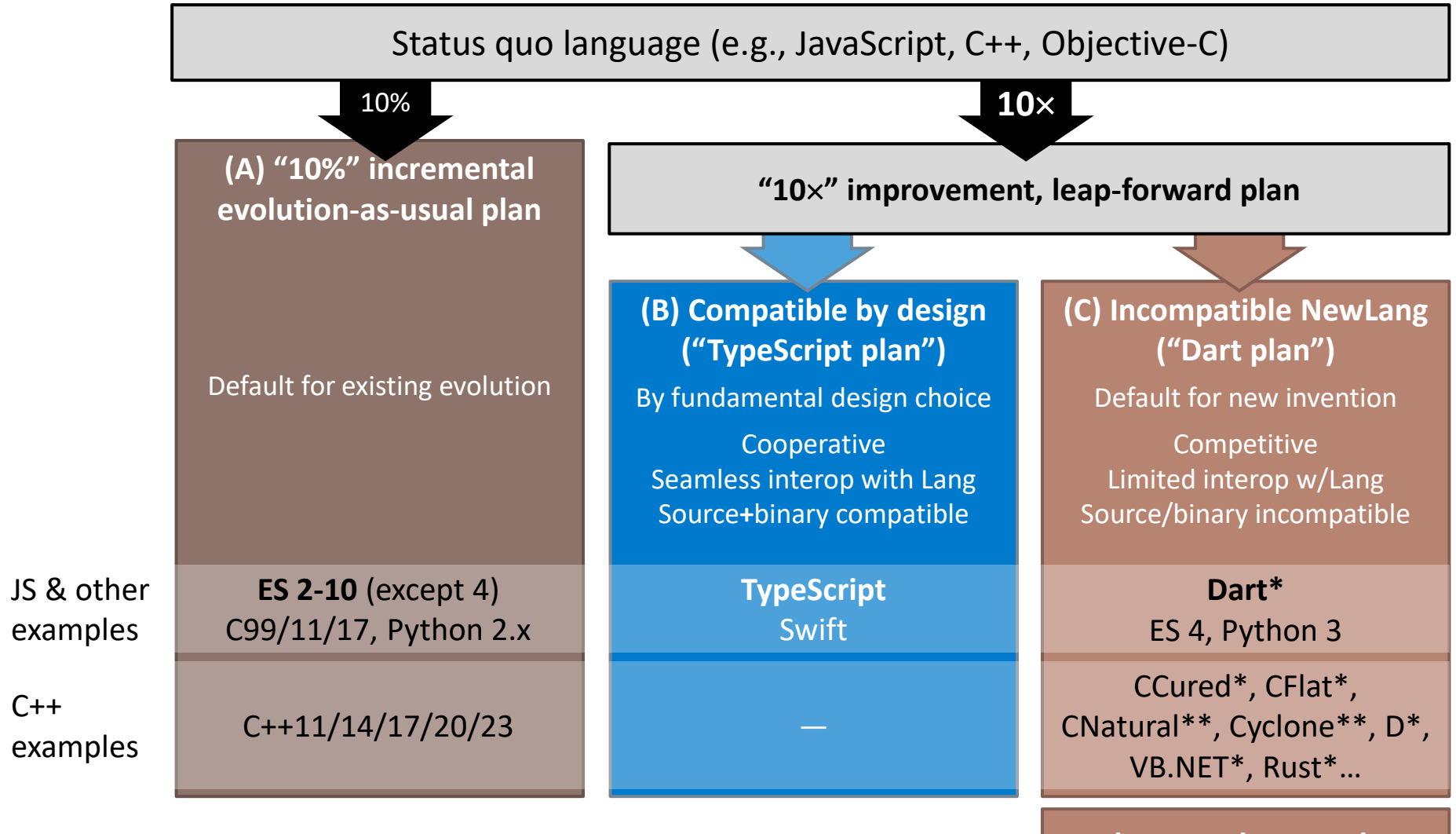
JS & other examples

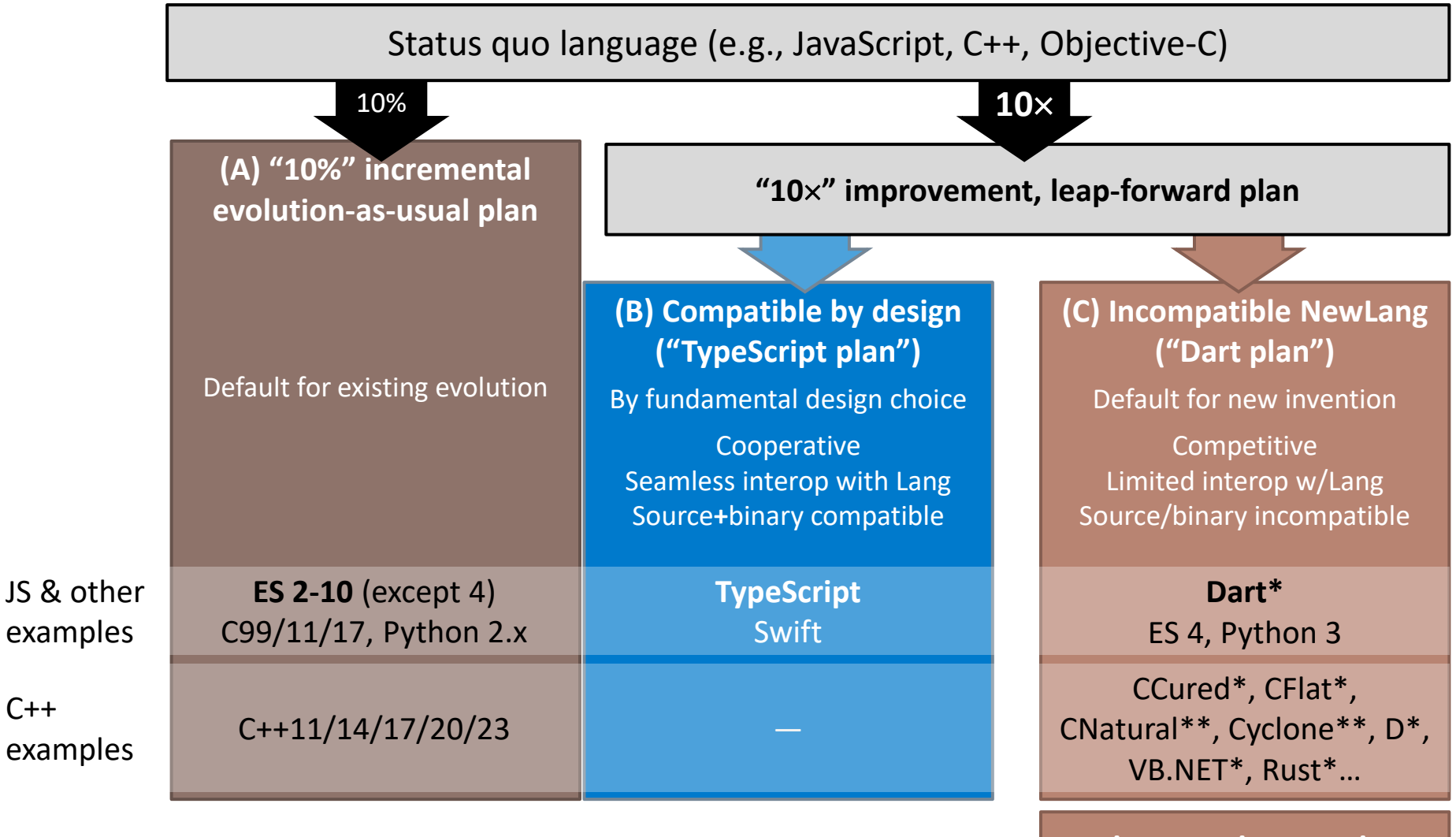
C++ examples


Status quo language (e.g., JavaScript, C++, Objective-C)
10%
(A) “10%” incremental evolution-as-usual plan
Default for existing evolution
JS & other examples
ES 2-10 (except 4)
C99/11/17, Python 2.x
C++ examples
C++11/14/17/20/23


Status quo language (e.g., JavaScript, C++, Objective-C)
10%
10×
(A) “10%” incremental evolution-as-usual plan
“10×” improvement, leap-forward plan
Default for existing evolution
JS & other examples
ES 2-10 (except 4)
C99/11/17, Python 2.x
C++ examples
C++11/14/17/20/23


Status quo language (e.g., JavaScript, C++, Objective-C)
10%
10×
(A) "10%" incremental evolution-as-usual plan
Default for existing evolution
"10×" improvement, leap-forward plan
(C) Incompatible NewLang ("Dart plan")
Default for new invention
Competitive
Limited interop w/Lang
Source/binary incompatible
JS & other examples
ES 2-10 (except 4)
C99/11/17, Python 2.x
Dart*
ES 4, Python 3
C++ examples
C++11/14/17/20/23
CCured*, CFlat*, CNatural**, Cyclone**, D*, .NET*, Rust*…


Status quo language (e.g., JavaScript, C++, Objective-C)
10%
10×
(A) "10%" incremental evolution-as-usual plan
Default for existing evolution
"10×" improvement, leap-forward plan
(B) Compatible by design ("TypeScript plan")
By fundamental design choice
Cooperative
Seamless interop with Lang
Source+binary compatible
(C) Incompatible NewLang ("Dart plan")
Default for new invention
Competitive
Limited interop w/Lang
Source/binary incompatible
JS & other examples
ES 2-10 (except 4)
C99/11/17, Python 2.x
TypeScript
Swift
Dart*
ES 4, Python 3
C++ examples
C++11/14/17/20/23
—
CCured*, CFlat*, CNatural**, Cyclone**, D*, VB.NET*, Rust*…


Status quo language (e.g., JavaScript, C++, Objective-C)
10%
10×
(A) "10%" incremental evolution-as-usual plan
Default for existing evolution
"10×" improvement, leap-forward plan
(B) Compatible by design ("TypeScript plan")
By fundamental design choice
Cooperative
Seamless interop with Lang
Source+binary compatible
(C) Incompatible NewLang ("Dart plan")
Default for new invention
Competitive
Limited interop w/Lang
Source/binary incompatible
JS & other examples
ES 2-10 (except 4)
C99/11/17, Python 2.x
TypeScript
Swift
Dart*
ES 4, Python 3
C++ examples
C++11/14/17/20/23
—
CCured*, CFlat*, CNatural**, Cyclone**, D*, VB.NET*, Rust*…

		Cooperative Seamless interop with Lang Source+binary compatible	Competitive Limited interop w/Lang Source/binary incompatible
JS & other examples	**ES 2-10** (except 4) C99/11/17, Python 2.x	**TypeScript** Swift	**Dart*** ES 4, Python 3
C++ examples	C++11/14/17/20/23	—	CCured*, CFlat*, CNatural**, Cyclone**, D*, VB.NET*, Rust*…
Extra costs		**+ cooperation** **Cooperate and participate** with Lang continued evolution **Contribute** evolution proposals to Lang evolution	**+ interop via wrapping** (*) plus “wrapper” strategies for interop to JS/C++: - some library/tool based (C++/WinRT, package:js, …) - some with compiler support (Objective-C++, C++/CLI, C++/CX) (**) no interop to JS/C++ and/or defunct language

Last 8 years

2015-16: Basic language design

- “Refactor C++” into fewer, simpler, composable, general features

2016 - : Try individual parts as standalone proposals for Syntax 1

- Flesh each out in more detail
- Validate it’s a problem the committee wants to solve for C++
- Validate it’s a solution direction programmers might like for C++

Lifetime	gc_arena	<=>	Reflection & metaclasses	Value-based exceptions	Parameter passing	Patmat using `is` and `as`
P1179		P0515	P0707	P0709	d0708	P2392
CppCon 2015/18	CppCon 2016	CppCon 2017	CppCon 2017/18	CppCon 2019	CppCon 2020	CppCon 2021

Last week on SG21 (Contracts) mailing list…

`RE: Searching for a keyword alternative for "assert"`

~100 replies in the thread

This was just the **most recent** discussion!

Timur Doumler (SG21 vice-chair)

"I ran [codesearch.isocpp.org] on all candidate keywords that have been suggested so far. Here are all of them sorted by how often they appear as user-defined identifiers in that particular dataset: …"

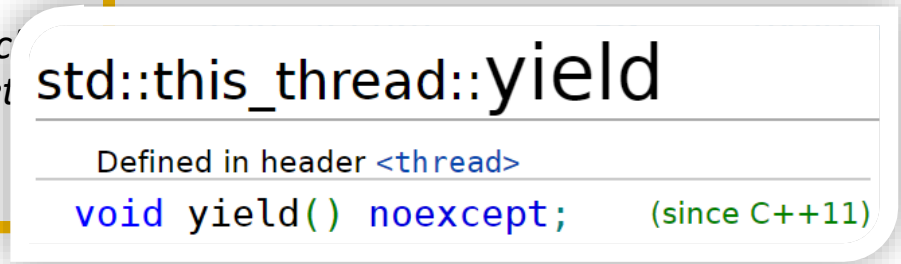
assertexpr	0
ccassert	0
co_assert	0
contract_assert	0
contractassert	0
cppassert	0
dynamic_assert	0
std_assert	0
stdassert	0
assrt	2
runtime_assert	5
cpp_assert	59
affirm	65
assess	163
asrt	256
cassert	380
aver	427
posit	617
claim	2800
ass	4145
must	4263
confirm	4341
assertion	5433
ensure	8149
verify	20727
expect	43725
check	147315

Timur Doumler (SG21 vice-chair)

"I ran [codesearch.isocpp.org] on all candidate keywords that have been suggested so far. Here are all of them sorted by how often they appear as user-defined identifiers in that particular dataset: …"

assertexpr	0	asrt	256
ccassert	0	cassert	380
co_assert	0	aver	427
contract_assert	0	posit	617
contractassert	0	claim	2800
cppassert	0	ass	4145
dynamic_assert	0	must	4263
std_assert	0	confirm	4341
stdassert	0	assertion	5433
assrt	2	ensure	8149
runtime_assert	5	verify	20727
			43725
			147315

*"… we made `requires` a full keyword in C++20, which has 3734 matches … but we also did *not* make `yield` a full keyword, which has 9533 matches in that dataset and chose `co_yield` instead… Where is the bar?"*



Facts of life

(1) Naming is hard

(2) Compatibility is a major naming constraint

All good names are taken, by the standard or by users

(cf. `std::byte`)

Compatibility

Consider this whole program…

```
thing      : @struct type = { x:int; y:int; z:int; }
state      :   @enum type = { idle; running; paused; }
name_or_num:  @union type = { name: std::string; num: i32; }

main: () = {
    mything := new<thing>( 1, 2, 3 );
    [[assert: mything.get() != nullptr]]
}
```

Q: Do you notice anything interesting about this code?

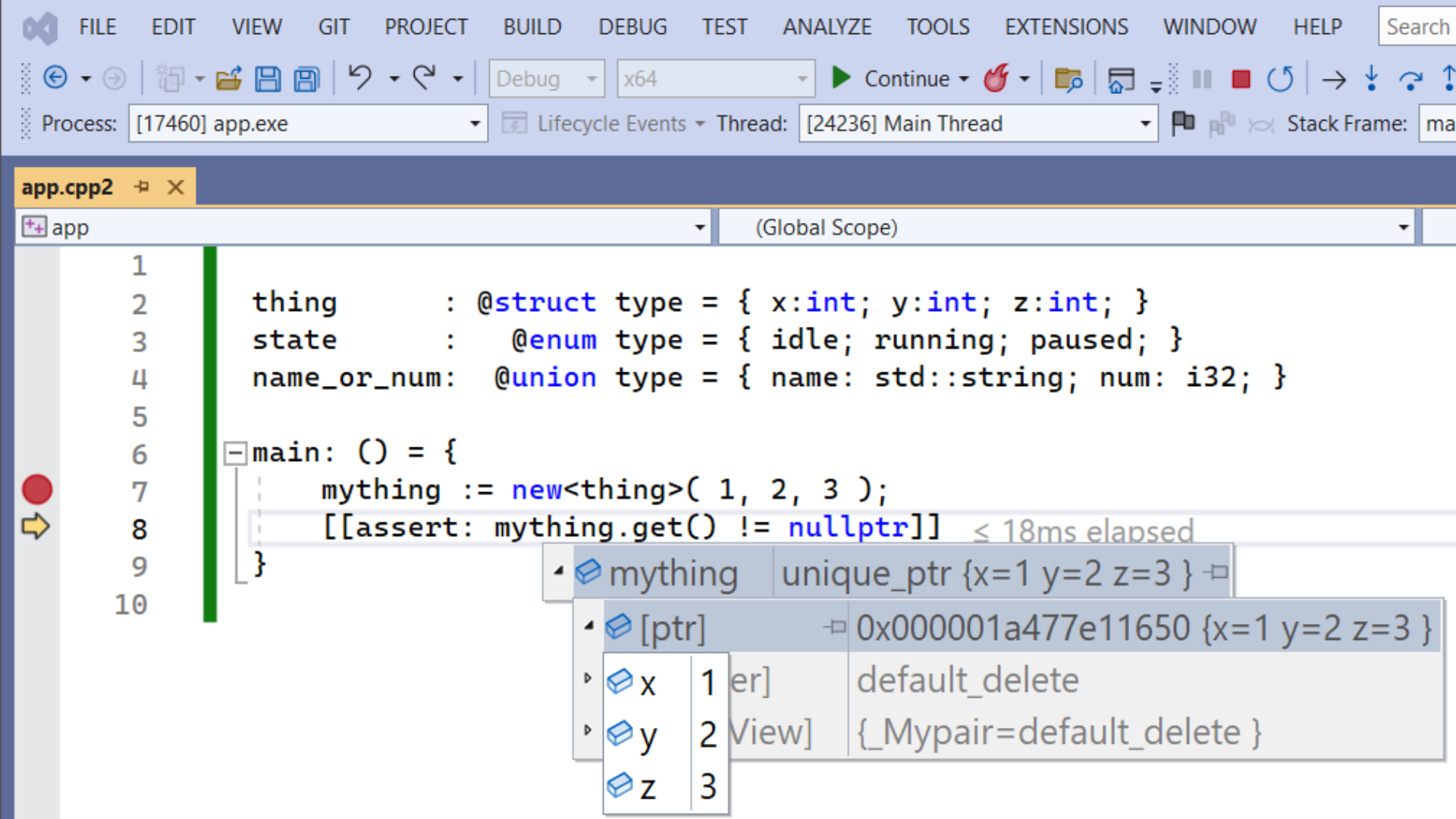
A: Reserved words? Macro names!? … Wait, what??

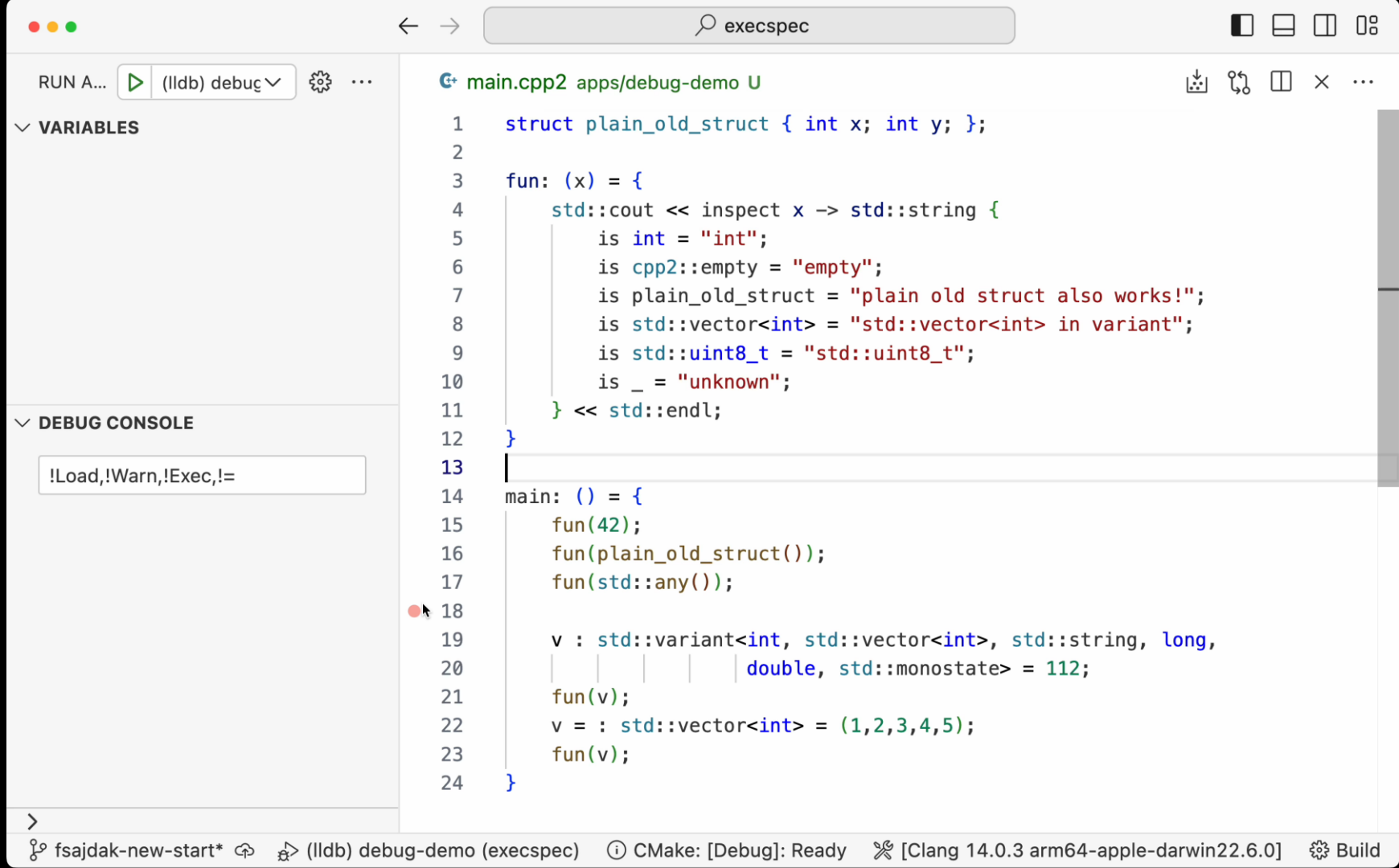
Inherent power of an alternate "syntax 2": We can change any default, incl. make reserved names, even macros, mean the right/safe thing, with no breaking change

Not just an implementation detail… but yes, cppfront runs *before* the preprocessor

Yesterday in the hallway outside this room:

“Are you emitting the Cpp2 code into the PDB?”


FILE EDIT VIEW GIT PROJECT BUILD DEBUG TEST ANALYZE TOOLS EXTENSIONS WINDOW HELP
Search
Debug
x64
Continue
Process: [17460] app.exe
Lifecycle Events
Thread: [24236] Main Thread
Stack Frame:
app.cpp2
app
(Global Scope)
1
2 thing : @struct type = { x:int; y:int; z:int; }
3 state : @enum type = { idle; running; paused; }
4 name_or_num: @union type = { name: std::string; num: i32; }
5
6 main: () = {
7 mything := new<thing>(1, 2, 3);
8 [[assert: mything.get() != nullptr]]
9 }
10
≤ 18ms elapsed
mything unique_ptr {x=1 y=2 z=3 }
[ptr] 0x000001a477e11650 {x=1 y=2 z=3 }
x 1
y 2
z 3
er] default_delete
View] {_Mypair=default_delete }


execspec
RUN A...
(lldb) debug
VARIABLES
DEBUG CONSOLE
!Load,!Warn,!Exec,!=
main.cpp2 apps/debug-demo U
1 struct plain_old_struct { int x; int y; };
2
3 fun: (x) = {
4 std::cout << inspect x -> std::string {
5 is int = "int";
6 is cpp2::empty = "empty";
7 is plain_old_struct = "plain old struct also works!";
8 is std::vector<int> = "std::vector<int> in variant";
9 is std::uint8_t = "std::uint8_t";
10 is _ = "unknown";
11 } << std::endl;
12 }
13
14 main: () = {
15 fun(42);
16 fun(plain_old_struct());
17 fun(std::any());
18
19 v : std::variant<int, std::vector<int>, std::string, long,
20 double, std::monostate> = 112;
21 fun(v);
22 v = : std::vector<int> = (1,2,3,4,5);
23 fun(v);
24 }
fsajdak-new-start*
(lldb) debug-demo (execspec)
CMake: [Debug]: Ready
[Clang 14.0.3 arm64-apple-darwin22.6.0]
Build

This is what compatibility looks like

This is what “it’s still 100% pure C++” looks like

100% seamless **interop** with all existing code is important

But it’s also **toolsets, build systems, debuggers, visualizers, profilers, linkers, custom in-house tools, test infrastructure, contract handling infrastructure**, …

When you have a compiler, groovy!
That’s step 1 of ~100 to real world industrial usage

*I do believe that there is real value in pursuing functional programming, but **it would be irresponsible to exhort everyone to abandon their C++ compilers** and start coding in Lisp, Haskell, or, to be blunt, any other fringe language.*

To the eternal chagrin of language designers, there are plenty of externalities that can overwhelm the benefits of a language…

*We have **cross platform** issues, proprietary **tool chains**, **certification** gates, **licensed** technologies, and stringent **performance** requirements on top of the issues with **legacy** codebases and **workforce** availability that everyone faces. …*

— John Carmack *[**emphasis** added]*

A tale of two plans — both valid and well understood

“Dart plan” — Competitive/successor 10× improvement

- New creation
- Limited interop, relies on wrapping/marshaling/thunking…
- Competes with standard (e.g., different modules, different generic constraints)
- **Evolves independently of standards committee — far fewer design constraints**
- **Needs to bootstrap a new ecosystem**

“TypeScript plan” — Cooperative/compatible 10× improvement

- Every `.js` file is a valid `.ts` file, add 1 class and see benefit
- Lowers to standard `.js`, 100% seamless compat with all JS libraries
- Cooperates with the standards committee (ECMAScript)
- **Brings evolution proposals to standards committee**
- **Leverages entire existing ecosystem — works with all JS implementations & tools**

Theme: 100% pure C++… just nicer

What if we could do “C++11 feels like a new language” again, this time for the whole language:

- as an intentional directed evolution
- to be **10× simpler**
- to be **50× safer**
- to evolve more freely again for another 30 years?

Cooperative C++ Evolution

Toward a Typescript for C++

HERB SUTTER