2023

Fill in the blank: _____ for C++

Herb Sutter

C++ now

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, …
- Example: Initialization → unifying SMFs/conversions

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?

Caveats

From CppCon 2022

My personal experiment

(learn some things, prove out some concepts, share some ideas)

Hilariously incomplete

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

> 290 issues

> 130 pull requests

> 100 contributors

Abhinav00, Robert Adam, Adam, Aaron Albers, Alex, Graham Asher, Peter Barnett, Sean Baxter, Jan Bielak, Simon Buchan, Michael Clausen, ct-clmsn, Joshua Dahl, Denis, Matthew Deweese, dmicsa, dobkeratops, Deven Dranaga, Konstantin F, Igor Ferreira, Stefano Fiorentino, fknauf, Robert Fry, Artie Fuffkin, Gabriel Gerlero, Matt Godbolt, William Gooch, ILoveGoulash, Víctor M. González, Terence J. Grant, GrigorenkoPV, HALL9kv0,

Morten Hattesen, Neil Henderson, Michael Hermier, h-vetinari, Stefan Isak, Tim Keitt, Vanya Khodor, Hugo Lindström, Ferenc Nandor Janky, jarzec, jgarvin, Dominik Kaszewski, kelbon, Marek Knápek, Emilia Kond, Vladimir Kraus, Ahmed Al Lakani, Junyoung Lee, Greg Marr, megajocke, Thomas Neumann, Niel, Jim Northrup, Daniel Oberhoff, Jussi Pakkanen, PaTiToMaSteR, Johel Ernesto Guerrero Peña, Bastien Penavayre, Daniel Pfeifer, Piotr, Davide Pomi, Andrei Rabusov, rconde01, realgdman, Alex Reinking, Pierre Renaux, Alexey Rochev, RPeschke, Sadeq, Filip Sajdak, satu, Wolf Seifert, Tor Shepherd, Luke Shore, Zenja Solaja, Francis Grizzly Smit, Sören Sprößig, Benjamin Summerton, Hypatia of Sva, SwitchBlade, Ramy Tarchichy, tkielan, Marzo Sette Torres Junior, Nick Treleaven, Jan Tusil, userxfce, 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, …
- Example: Initialization → unifying SMFs/conversions

Compatibility for C++

- Why • What kind • What plan

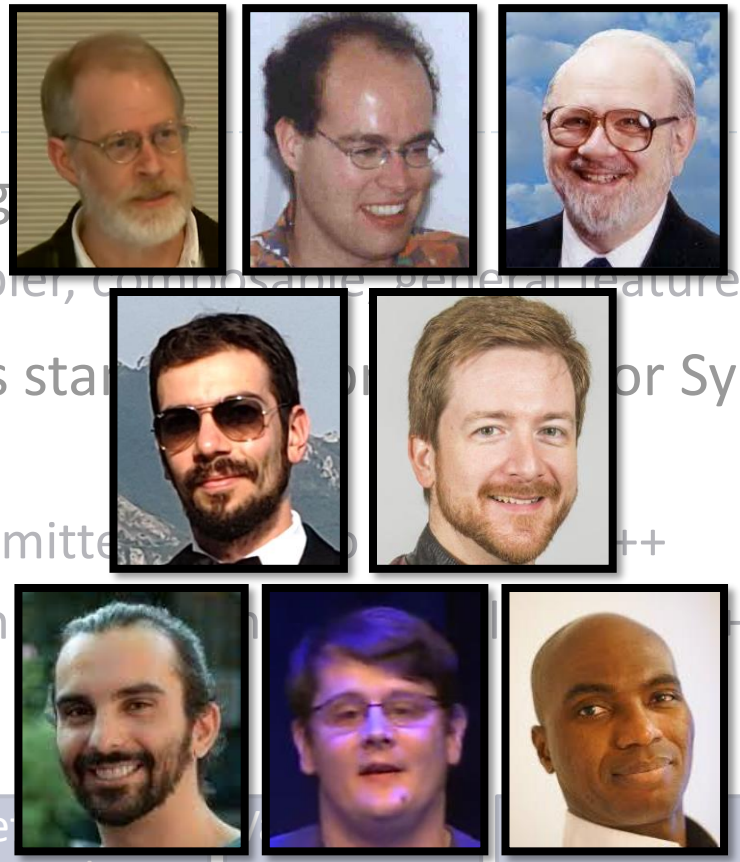
Last 8 years

2015-16: Basic language desig

- “Refactor C++” into fewer, simpler, [illegible] features

2016 - : Try individual parts as sta[illegible]or Syntax 1

- Flesh each out in more detail
- Validate it’s a problem the committe[illegible]++
- Validate it’s a solution direction[illegible]



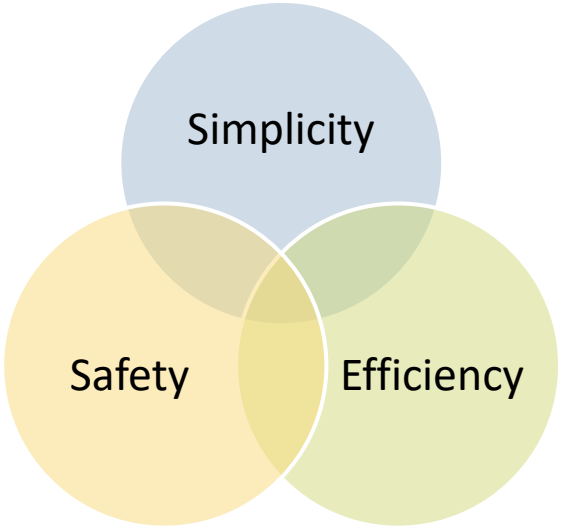
Lifetime	gc_arena	<=>	Re[illegible] metaclasses	exceptions	passing	Patmat using is and as
P1179		P0515	P0707	P0709	d0708	P2392
CppCon 2015/18	CppCon 2016	CppCon 2017	CppCon 2017/18	CppCon 2019	CppCon 2020	CppCon 2021

Sometimes you can have it all

These are not always in tension

Judicious abstraction

⇒ **directly express intent**

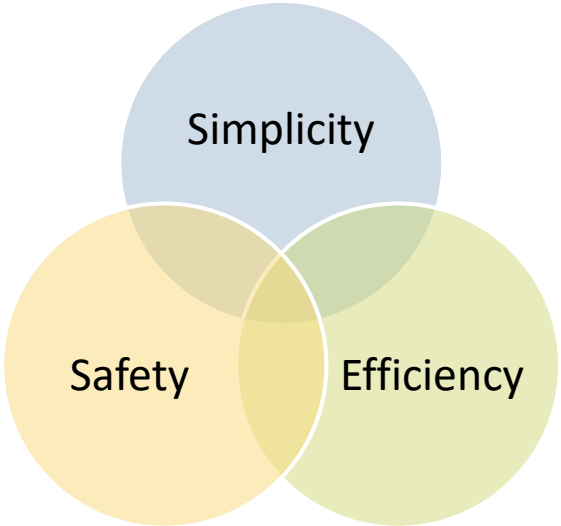


Sometimes you can have it all

These are not always in tension

Judicious abstraction

⇒ **directly express intent**

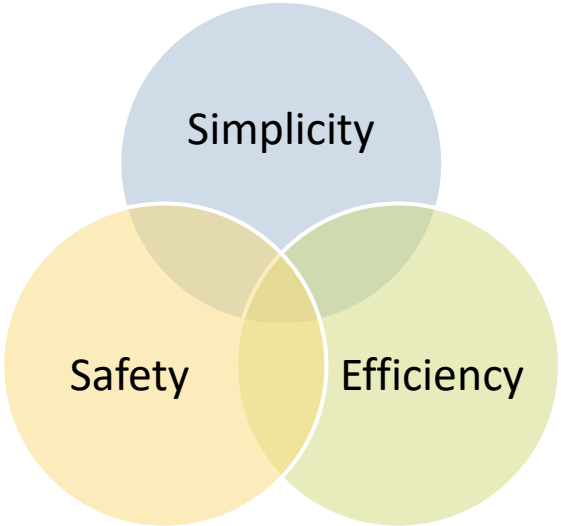


Sometimes you can have it all

These are not always in tension

Judicious abstraction

⇒ **directly express intent**




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

Chained comparisons

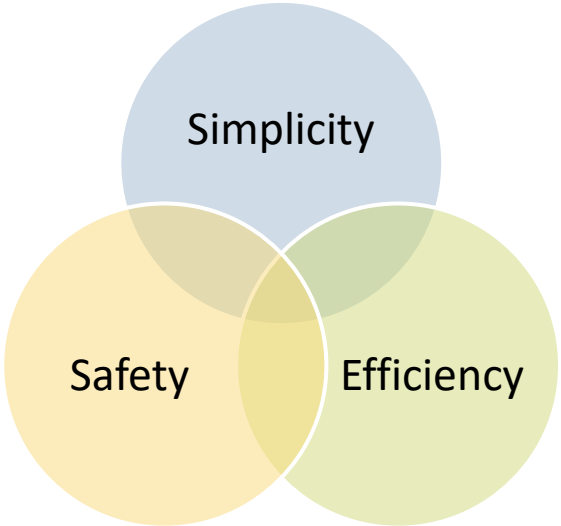
Simple	DRY & say what you mean
Safe	Cpp2 allows mathematically sound (transitive) chains, but not `a >= b < c` or `d != e != f`
Efficient	Single evaluation for all terms

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

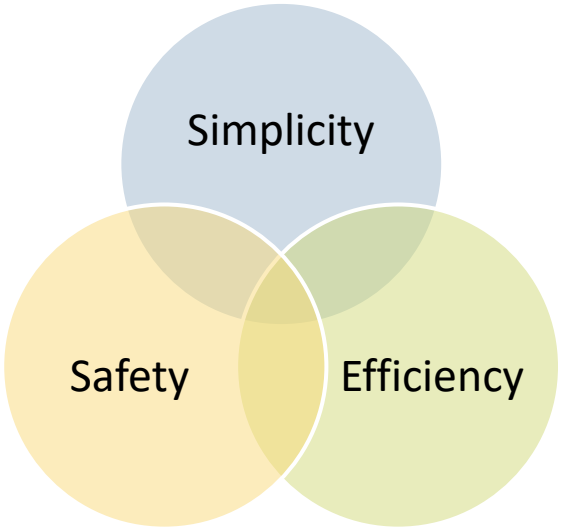
Simple	No extra var, directly expressed
Safe	Structured, reduces demand for (even a structured) `goto`
Efficient	Can't make it faster by hand

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`)

type: User-defined types

```
myclass : type = {
    data: i32 = 42 * 12;
    more: std::string = "xyzzy";

    print: (this) = {
        std::cout << "data: (data)$, more: (more)$\n";
    }

    add_to_data: (inout this, x: i32) = {
        data += x;       // "this.data += x;"
    }

    nested : type = {
        g: () = std::cout << "N::myclass::nested::g
    }
}
```

name : type = value
left-to-right declaration

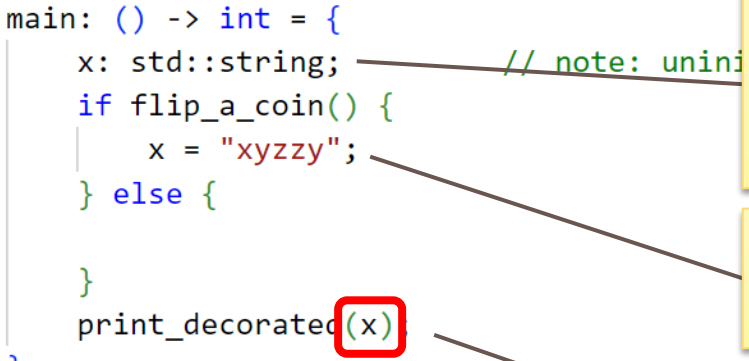
explicit **this**
alias for this object (not pointer)
normal parameter passing

safe and simple defaults:

functions & types are **public**

objects are **private**

only generated functions are **dtor** (if not written) and **default ctor** (if no ctors written)



no = ⇒ **uninitialized**

allocate name & storage, even if not yet ready to construct

unified = ⇒ **value-set**

can construct

guaranteed init-before-use
with program-meaningful value

```
demo.cpp2(6,5): error: local variable x must be initialized on both branches or n
either branch
demo.cpp2(7,5): error: "if" initializes x on:
  branch starting at line 7
but not on:
  branch starting at line 9
  ==> program violates initialization safety guarantee - see previous errors
```

```
main: () -> int = {
    x: std::string;                    // note: uninitialized!
    if flip_a_coin() {
        x = "xyzzy";
    } else {
        fill( out x, "plugh", 3 );  // not
    }
    print_decorated(x);
}

fill: (out x: std::string,
       value: std::string,
       count: int)
    [[pre: value.ssize() >= count,
           "value must contain at least co
= {
    x = value.substr(0, count);
}
```

From

out arguments/parameters
⇒ **composable initialization**

every function with an **out** parameter is effectively a (delegating) constructor

= ⇒ **initialized**

"=" is a generalized value-set syntax, used for both constructing and assigning
⇒ **can express both at once**

```
demo.cpp2... ok (mixed Cpp1/Cpp2, Cpp2 code passes safety checks)
```

operator=: Unified ctor/assign/dtor/conversion

As pointed out in paper d0708:

`out this`	is naturally a construct/assign operation
`/*in*/ that`	is naturally a copy/move operation

Therefore the four combinations:

`( out this, that)`	{construct,assign} x {copy,move}
`( out this, move that)`	{construct,assign} move
`(inout this, that)`	assign {copy,move}
`(inout this, move that)`	assign move

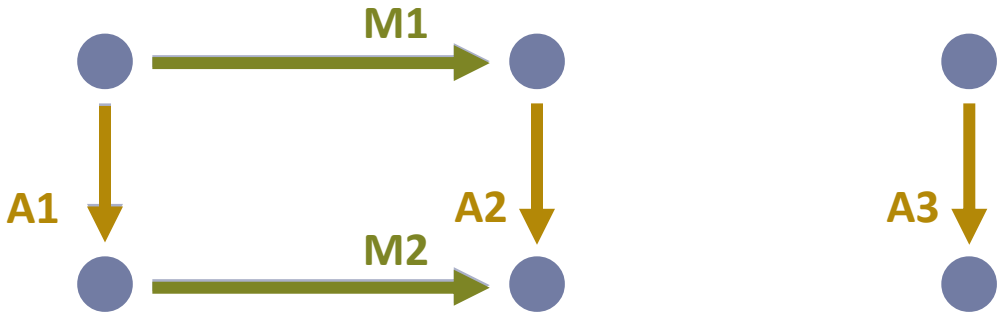
And:

`( out this, val: other_type)`	{construct,assign} convert
`(inout this, val: other_type)`	assign convert
`( move this)`	destroy

operator=, this & that

is usable as (generates) →

	in that *copy*	move that *move*	other (not that) *conversion*
out this *construction*		M1	
inout this *assignment*	A1	A2 M2	A3



operator=

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

    //  {copy, move} x {construction, assignment}
    operator=: (out this, that) = {
        // name = that.name;
        // addr = that.addr;
        std::cout << "general operator=";
    }

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

data members,
private by default

that is like this,
another object of this type

unified special member function(s)

always defaults to **memberwise**
(even when customized)

unified conversion function(s)

explicit by default

always defaults to **memberwise**
(even when customized)

operator=

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

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

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

operator=

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

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

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

```
class myclass {
    private: std::string name {"Henry"};
    private: std::string addr {"123 Ford Dr."};

    public: myclass(myclass const& that);
    public: auto operator=(myclass const& that) -> myclass& ;
    public: myclass(myclass&& that);
    public: auto operator=(myclass&& that) -> myclass& ;

    public: explicit myclass(cpp2::in<std::string> x);
    public: auto operator=(cpp2::in<std::string> x) -> myclass& ;
};

myclass::myclass(myclass const& that)
    : name{ that.name }
    , addr{ that.addr }
{
    std::cout << "general operator=";
```

operator=

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

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

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

```
myclass::myclass(myclass const& that)
    : name{ that.name }
    , addr{ that.addr }
{
    std::cout << "general operator=";
}
auto myclass::operator=(myclass const& that) -> myclass& {
    name = that.name;
    addr = that.addr;
    std::cout << "general operator=";
    return *this;
}
myclass::myclass(myclass&& that)
    : name{ std::move(that).name }
    , addr{ std::move(that).addr }
{
    std::cout << "general operator=";
}
auto myclass::operator=(myclass&& that) -> myclass& {
    name = std::move(that).name;
    addr = std::move(that).addr;
    std::cout << "general operator=";
```

operator=

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

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

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

```
    , addr{ std::move(that).addr }
{
    std::cout << "general operator=";
}
auto myclass::operator=(myclass&& that) -> myclass& {
    name = std::move(that).name;
    addr = std::move(that).addr;
    std::cout << "general operator=";
    return *this;
}

myclass::myclass(cpp2::in<std::string> x)
    : name{ x }
{
    std::cout << "conversion - from string";
}
auto myclass::operator=(cpp2::in<std::string> x) -> myclass& {
    name = x;
    addr = "123 Ford Dr.";
    std::cout << "conversion - from string";
    return *this;
}
```

operator=

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

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

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

```
    , addr{ std::move(that).addr }
{
    std::cout << "general operator=";
}
auto myclass::operator=(myclass&& that) -> myclass& {
    name = std::move(that).name;
    addr = std::move(that).addr;
    std::cout << "general operator=";
    return *this;
}

myclass::myclass(cpp2::in<std::string> x)
    : name{ x }
{
    std::cout << "conversion - from string";
}
auto myclass::operator=(cpp2::in<std::string> x) -> myclass& {
    name = x;
    addr = "123 Ford Dr.";
    std::cout << "conversion - from string";
    return *this;
}
```

operator=

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

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

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

```
class myclass {
    private: std::string name {"Henry"};
    private: std::string addr {"123 Ford Dr."};

    public: myclass(myclass const& that);
    public: auto operator=(myclass const& that) -> myclass& ;
    public: myclass(myclass&& that);
    public: auto operator=(myclass&& that) -> myclass& ;

    public: explicit myclass(cpp2::in<std::string> x);
    public: auto operator=(cpp2::in<std::string> x) -> myclass& ;
};

myclass::myclass(myclass const& that)
    : name{ that.name }
    , addr{ that.addr }
{
    std::cout << "general operator=";
}
auto myclass::operator=(myclass const& that) -> myclass& {
    name = that.name;
    addr = that.addr;
    std::cout << "general operator=";
    return *this;
}
myclass::myclass(myclass&& that)
    : name{ std::move(that).name }
    , addr{ std::move(that).addr }
{
    std::cout << "general operator=";
}
auto myclass::operator=(myclass&& that) -> myclass& {
    name = std::move(that).name;
    addr = std::move(that).addr;
    std::cout << "general operator=";
    return *this;
}

myclass::myclass(cpp2::in<std::string> x)
    : name{ x }
{
    std::cout << "conversion - from string";
}
auto myclass::operator=(cpp2::in<std::string> x) -> myclass& {
    name = x;
    addr = "123 Ford Dr.";
    std::cout << "conversion - from string";
    return *this;
}
```

operator=

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

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

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

```
    , addr{ std::move(that).addr }
{
    std::cout << "general operator=";
}
auto myclass::operator=(myclass&& that) -> myclass& {
    name = std::move(that).name;
    addr = std::move(that).addr;
    std::cout << "general operator=";
    return *this;
}

myclass::myclass(cpp2::in<std::string> x)
    : name{ x }
{
    std::cout << "conversion - from string";
}
auto myclass::operator=(cpp2::in<std::string> x) -> myclass& {
    name = x;
    addr = "123 Ford Dr.";
    std::cout << "conversion - from string";
    return *this;
}
```

operator=

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

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

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

operator=

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

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

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

Virtuality is all about `this`

Virtual function

- `this` parameter is `virtual`, `override`, or `final` (exactly one)
- Example: `speak: (virtual this) = /*...body...*/`

Base class

- Declared like a data member named `this`
- Example: `this: Shape = /*...default value...*/`

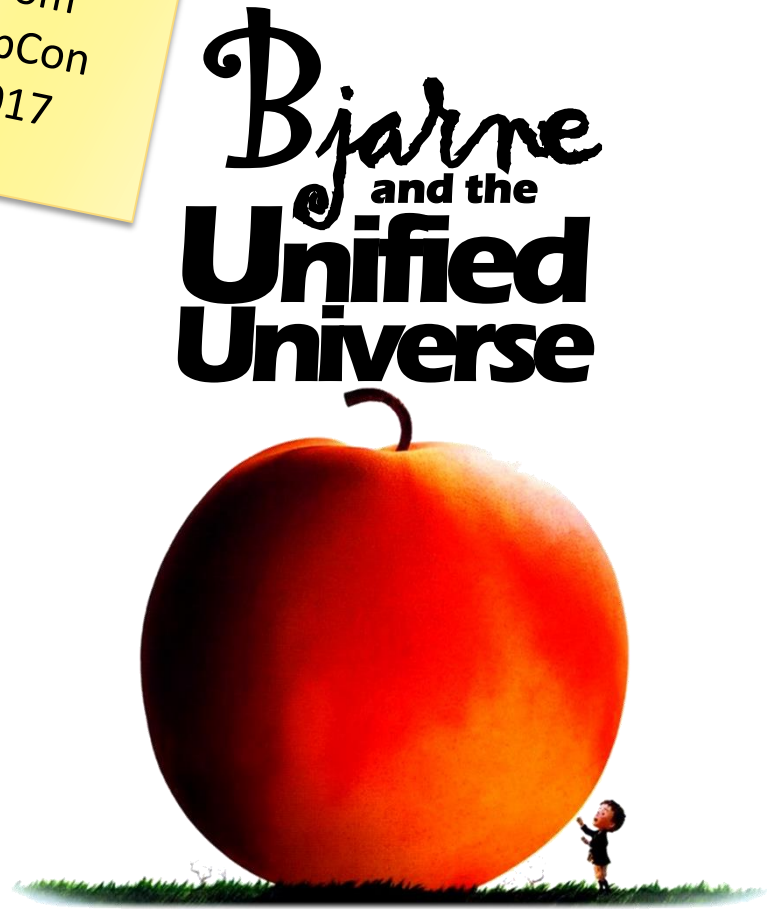
Demo: Inheritance

Now Playing

From CppCon 2017

Now Playing

From CppCon 2017



Now Playing

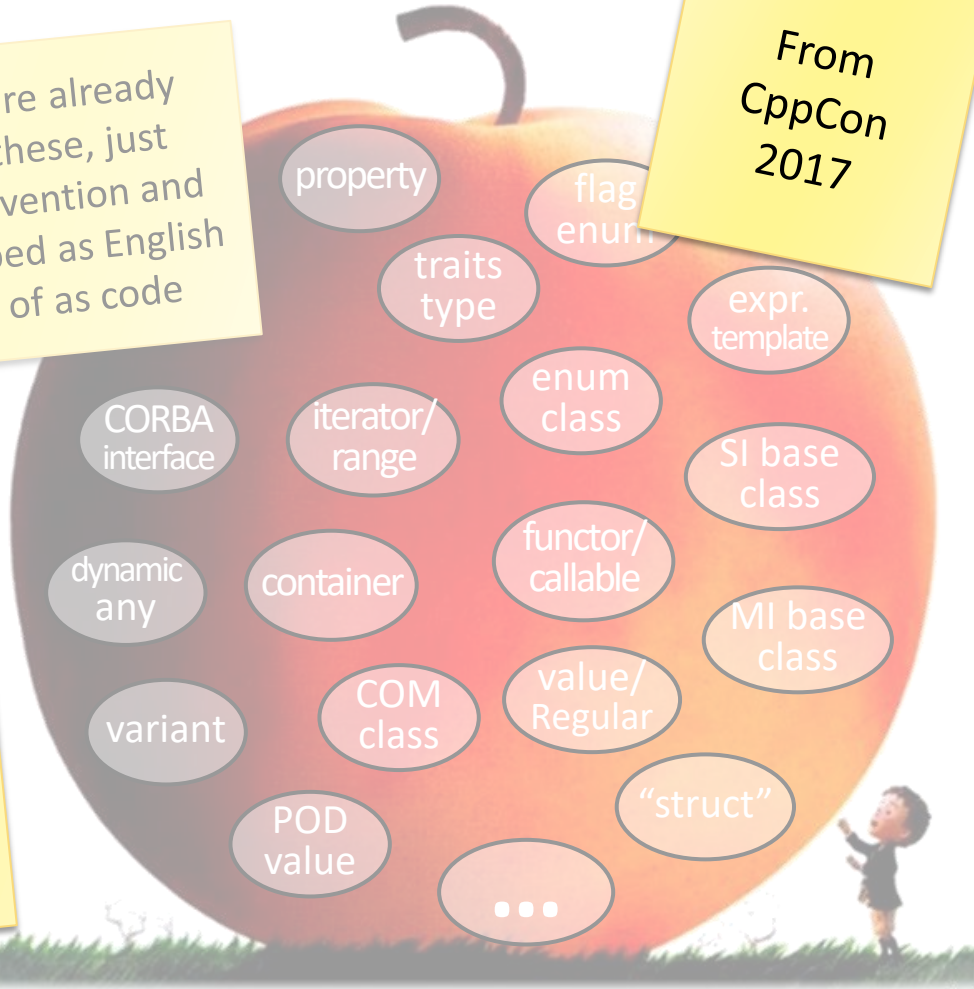
- The C++ type system is unified!

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=(const Point&&) =default;
};
```

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

The languag

Source code

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

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

nothing too crazy!

just participating in interpreting the meaning of definitions

From CppCon 2017

not making the language grammar mutable

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

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=(const Point&&) =default;
};
```

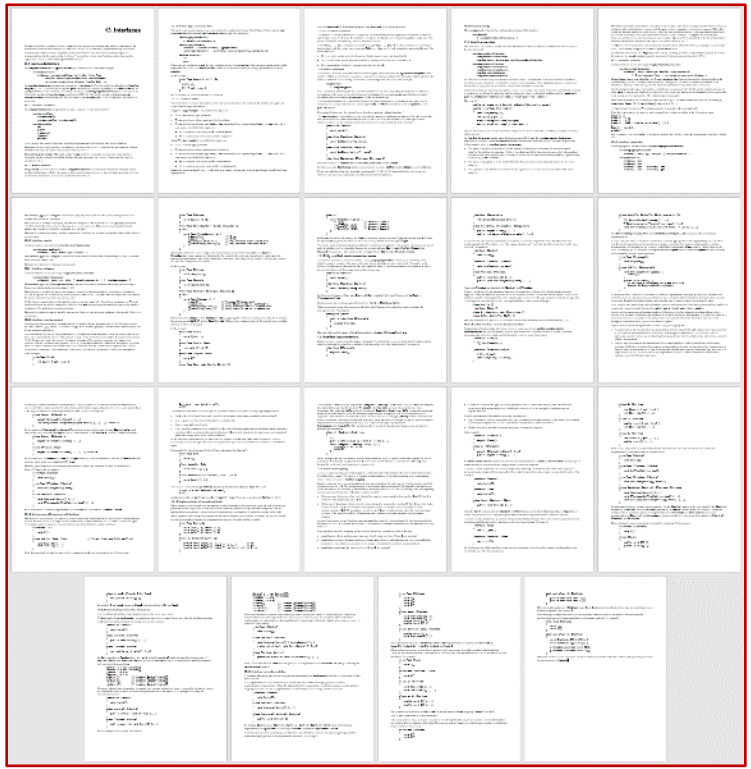
not making definitions mutable after the fact

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

interface (implementation)

From ACCU 2017 and CppCon 2017

C# language: ~18pg, English



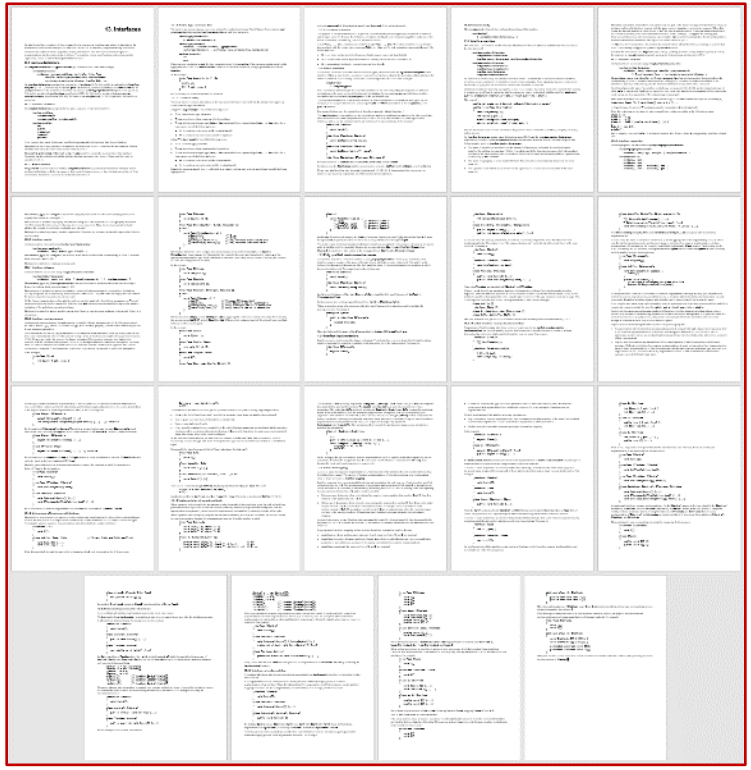
Proposed C++: ~10 lines, testa

```
$class interface {
   ~interface() noexcept { }
   constexpr {
      compiler.require($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)

From a week ago (Cpp1 syntax)

C# language: ~18pg, English



cppfront last week

```
auto interface(meta::type_declaration& t) -> void {
    auto has_dtor = false;
    for (auto m : t.get_members()) {
        m.require( !m.is_object(),
                   "interfaces may not contain data objects")
        if (m.is_function()) {
            auto mf = m.as_function();
            mf.require( !mf.is_copy_or_move(),
                        "interfaces may not copy or move; cor
            mf.require( !mf.has_initializer(),
                        "interface functions must not have a
            mf.require( mf.make_public(),
                        "interface functions must be public")
            mf.make_virtual();
            has_dtor |= mf.is_destructor();
        }
    }
    if (!has_dtor) {
        t.require( t.add_member( "operator=: (virtual move th
                   "could not add virtual destructor");
    }
}
```

interface (implementation)

From a week ago (Cpp1 syntax)

cppfront last week

```
auto interface(meta::type_declaration& t) -> void {
    auto has_dtor = false;
    for (auto m : t.get_members()) {
        m.require( !m.is_object(),
                   "interfaces may not contain data objects");
        if (m.is_function()) {
            auto 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.make_virtual();
            has_dtor |= mf.is_destructor();
        }
    }
    if (!has_dtor) {
        t.require( t.add_member( "operator=: (virtual move this) = { }"),
                   "could not add virtual destructor");
    }
}
```

- Clang Format with Cpp2 support. By Johel Ernesto Guerrero Peña. A Clang Format fork with support for Cpp2 syntax.
- Meson support for cppfront. By Jussi Pakkanen, creator of Meson.
- /JohelEGP/jegp.cmake_modules/#jegpcpp2. By Johel Ernesto Guerrero Peña. A CMake module to build Cpp2 source files.
- Conan recipe for cppfront. By Fernando Pelliccioni. A cppfront Conan recipe/package in the Conan Center Index (the official package index for

Commits on May 6, 2023

Start self-hosting: half of `reflect.h` (all the metafunctions)

committed 17 hours ago

```
C:\GitHub\cppfront\source>cppfront.exe reflect.h -verb
reflect.h2... ok (mixed Cpp1/Cpp2, Cpp2 code passes safety checks)
   Cpp1: 173 lines
   Cpp2: 599 lines (77%)
```

interface (implementation)

From two days ago (Cpp2 syntax)

cppfront last weekend

```
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.make_virtual();
            has_dtor |= mf.is_destructor();
        }
    }
    if !has_dtor {
        t.require( t.add_member( "operator=: (virtual move this) = { }"),
                   "could not add virtual destructor");
    }
}
```

Example from last week's 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 last week's 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
    // ...
}
```

Metafunctions implemented so far

`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`	(not a reserved word in Cpp2) A `basic_value` with all public members, no virtual functions, and no user-written `operator=`

Demo: Type metafunctions

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, …
- Example: Initialization → unifying SMFs/conversions

Compatibility for C++

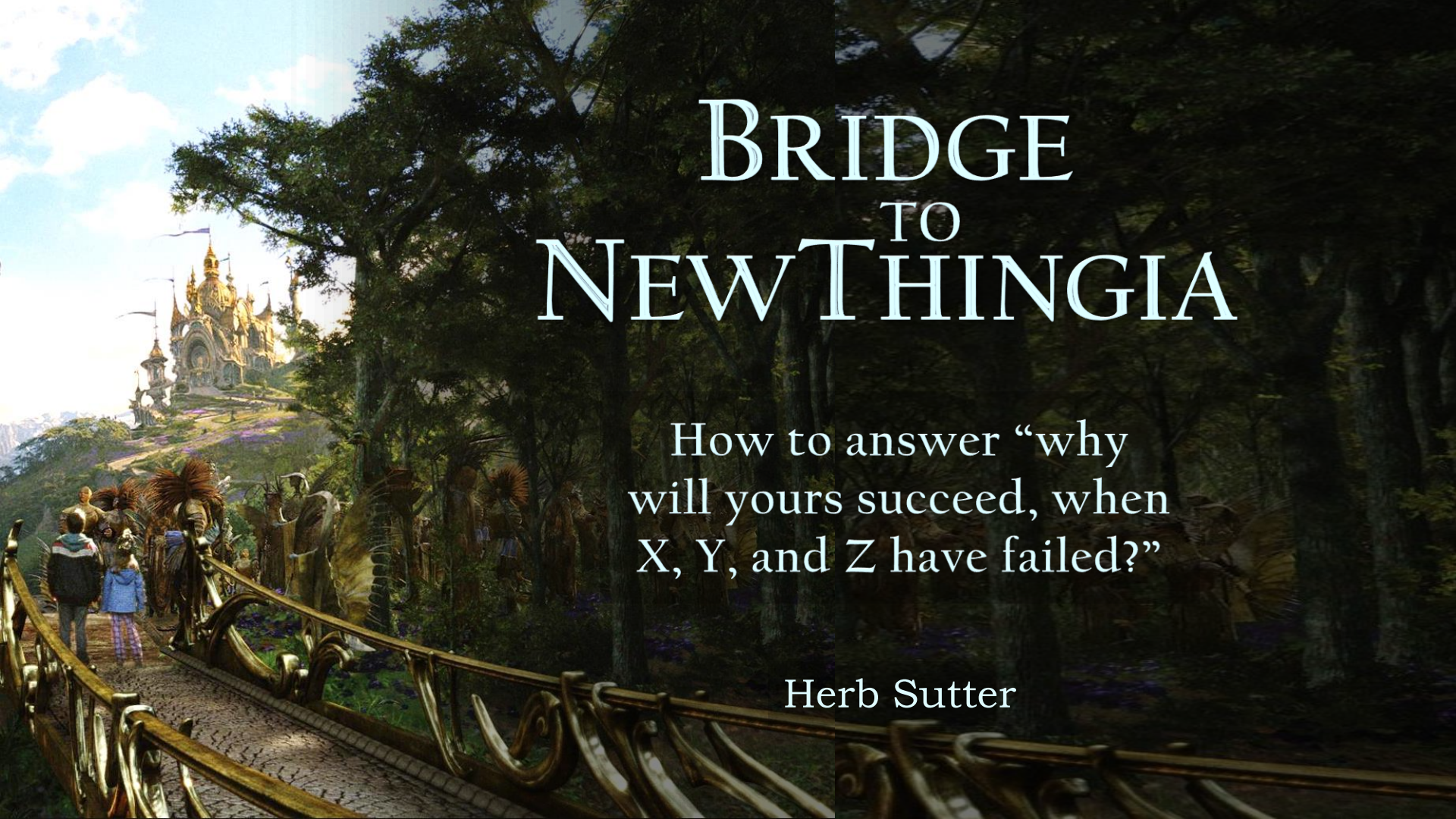
- Why • What kind • What plan

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. …*

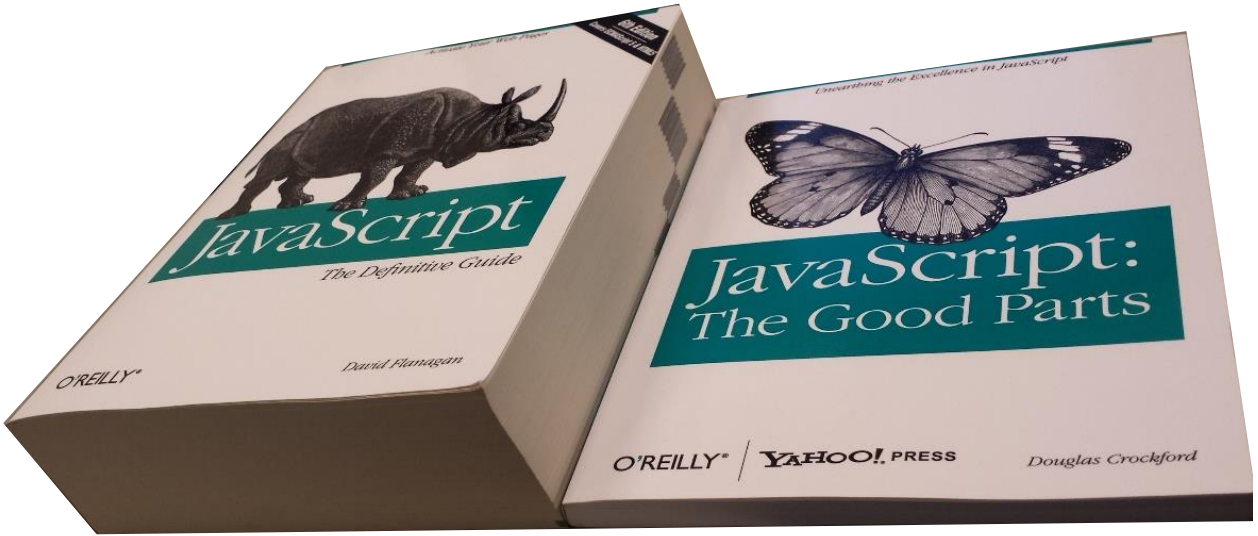
— John Carmack *[**emphasis** added]*


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

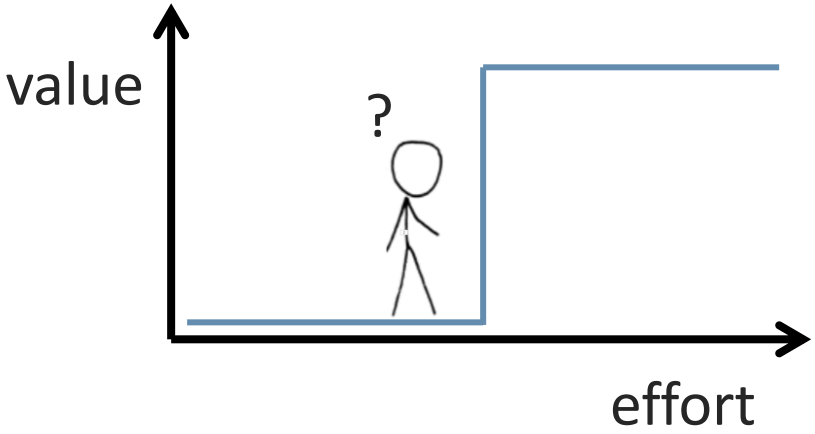
Example: JavaScript

Concise elevator pitch:

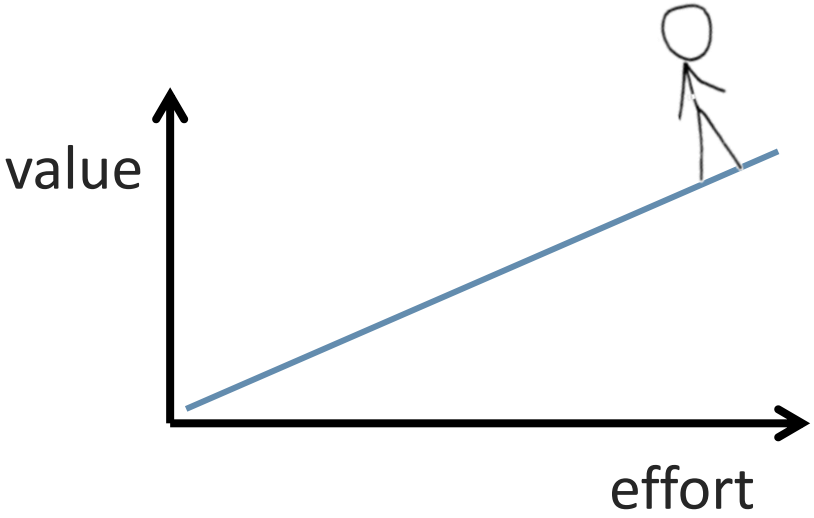
Well-known pain points with JavaScript



Which adoption function would you prefer?



A



B

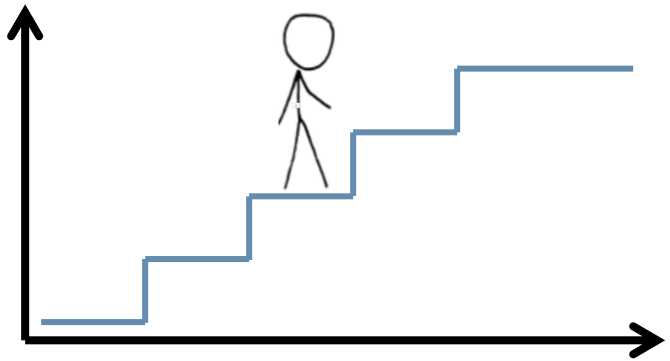
3. NewThing's **compatibility by design**

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."

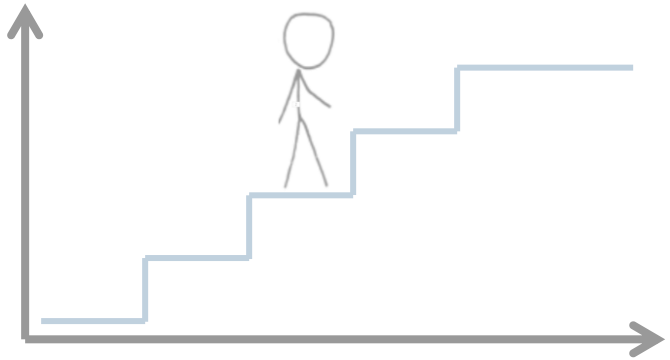
Ex: "Add NewLang file and see benefit."

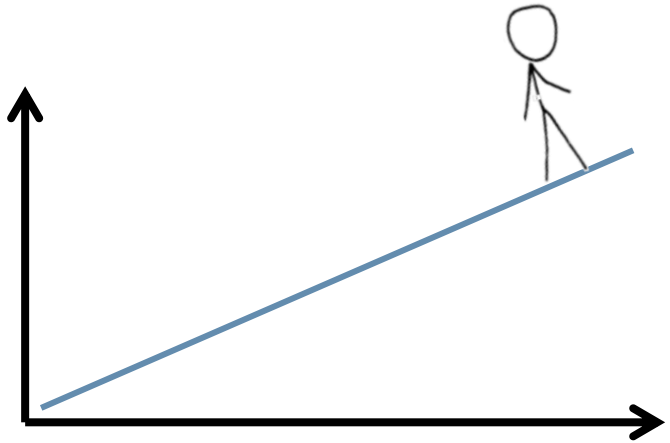
Grail: "An OldThing project can add NewThing **in one place** and start seeing benefit."

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.






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.

Swift: Bidirectional (Swift calls ObjC, ObjC calls Swift), ObjC-friendly object and lifetime models *(ObjC ARC + modules designed for Swift)*, automatic bridging header generation, tool support to view ObjC as if written in Swift.

Roslyn next-gen C# compiler: Strict compatibility requirements, adhered to rigorously via compat tests.

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

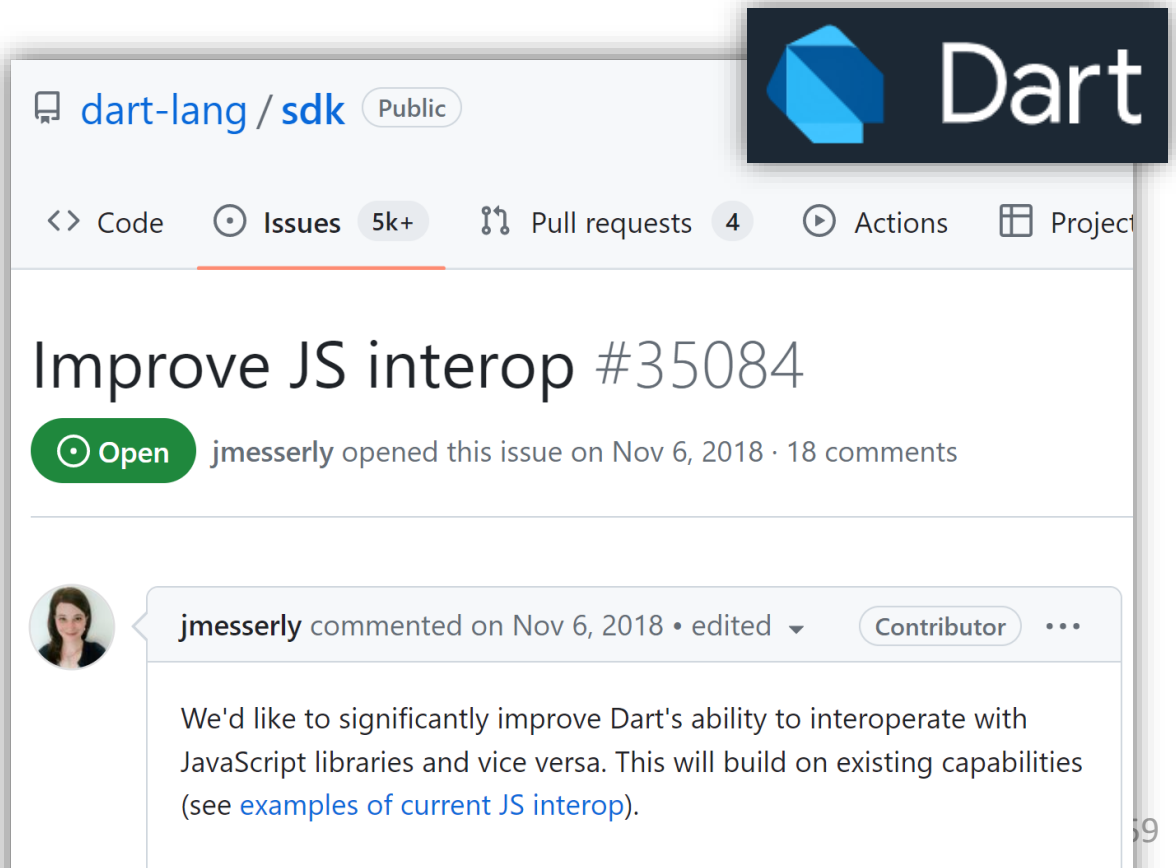
Observation

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.



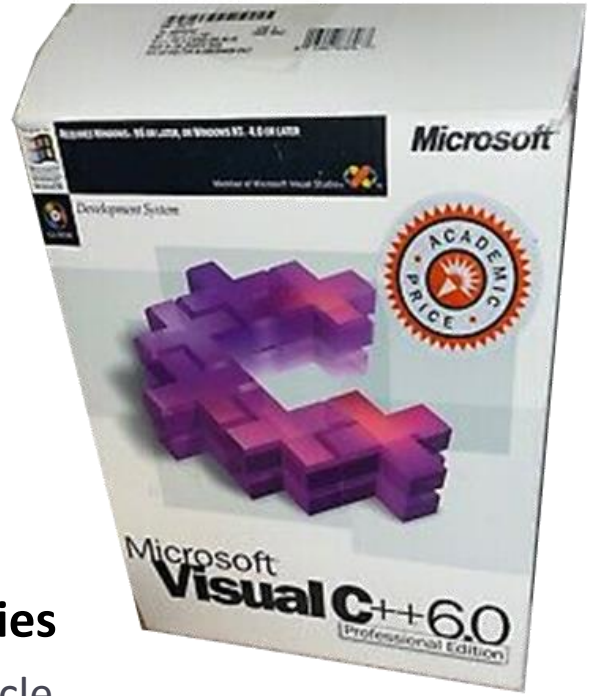
Visual C++ 6 (1998)

1998: VC++ 6.0 released

2008: Lots of customers still using v6

- IDE added many features, but slower & cumbersome
- **Usability regression is a "UX" incompatibility**
- Customers complained they could write and debug an MFC program better and faster in v6

2010: VC++ 2010 slogan "10 is the new 6"



Contributing factors: **Binary and source incompatibilities**

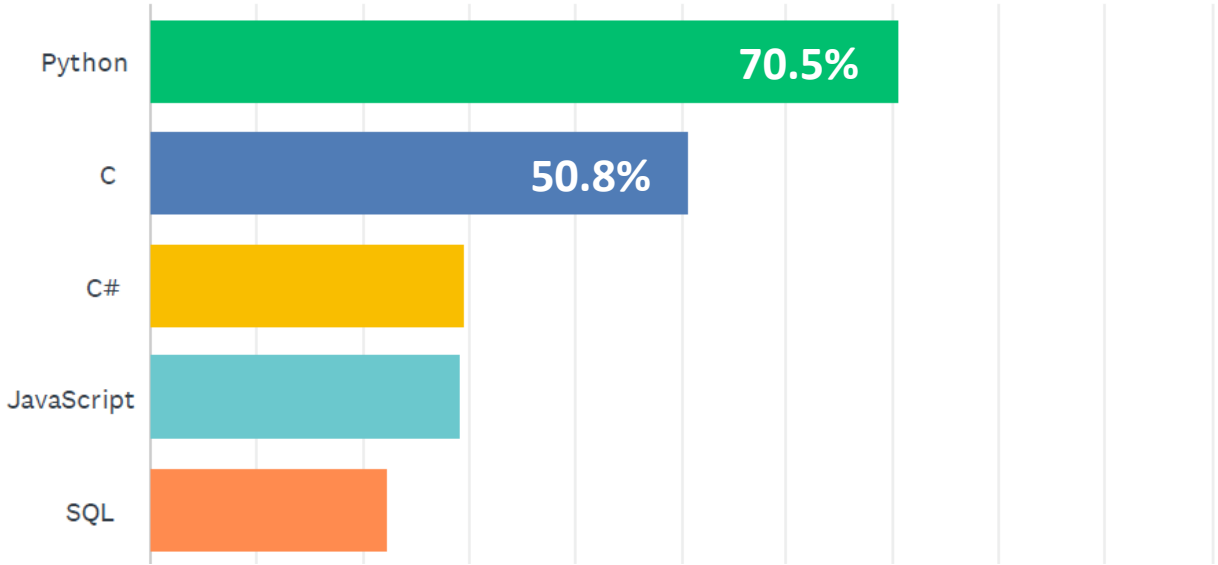
- Both happened simultaneously ⇒ opposite of virtuous cycle

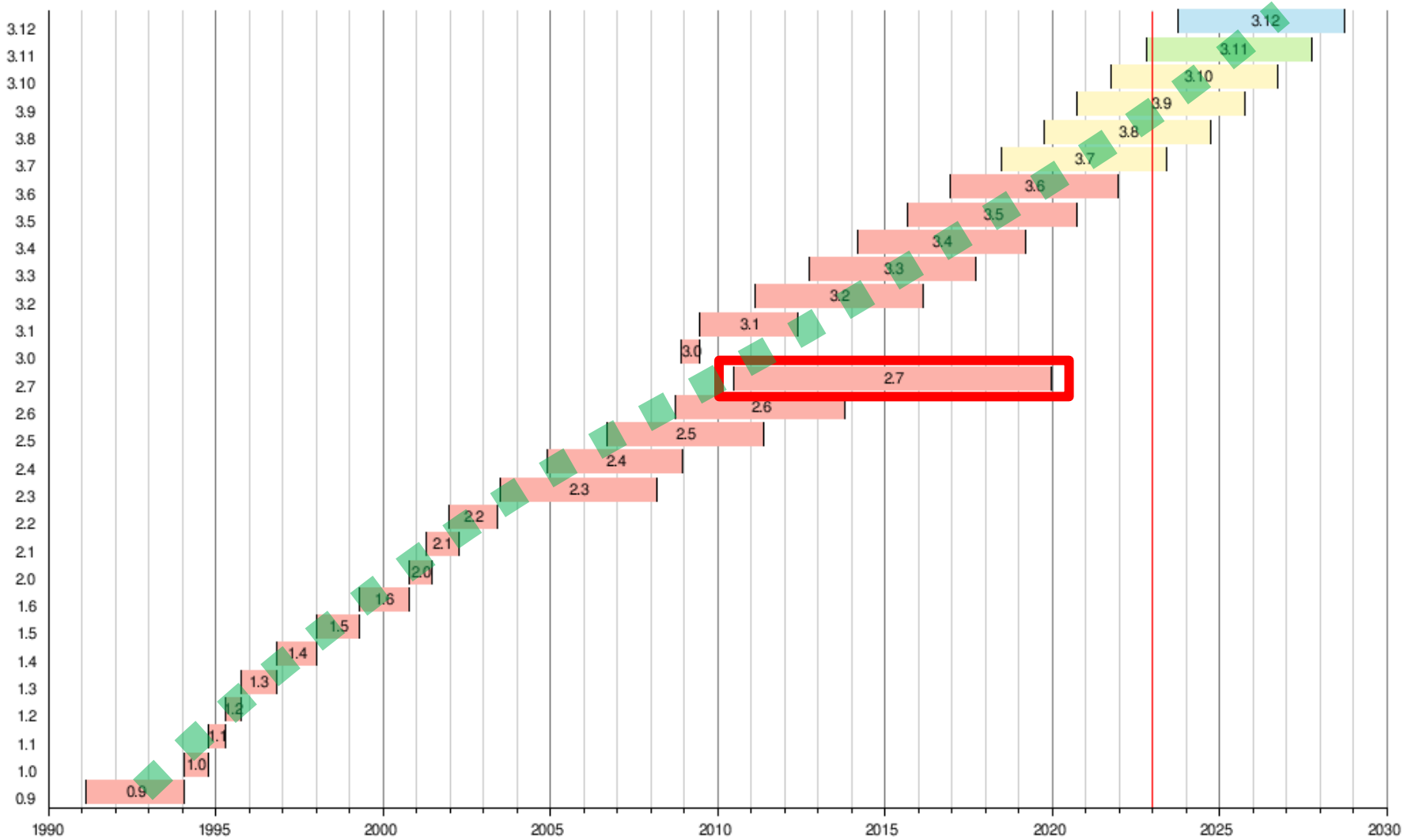
Q15 Besides C++, what programming languages/environments do you use in your current and recent projects? (select all that apply)

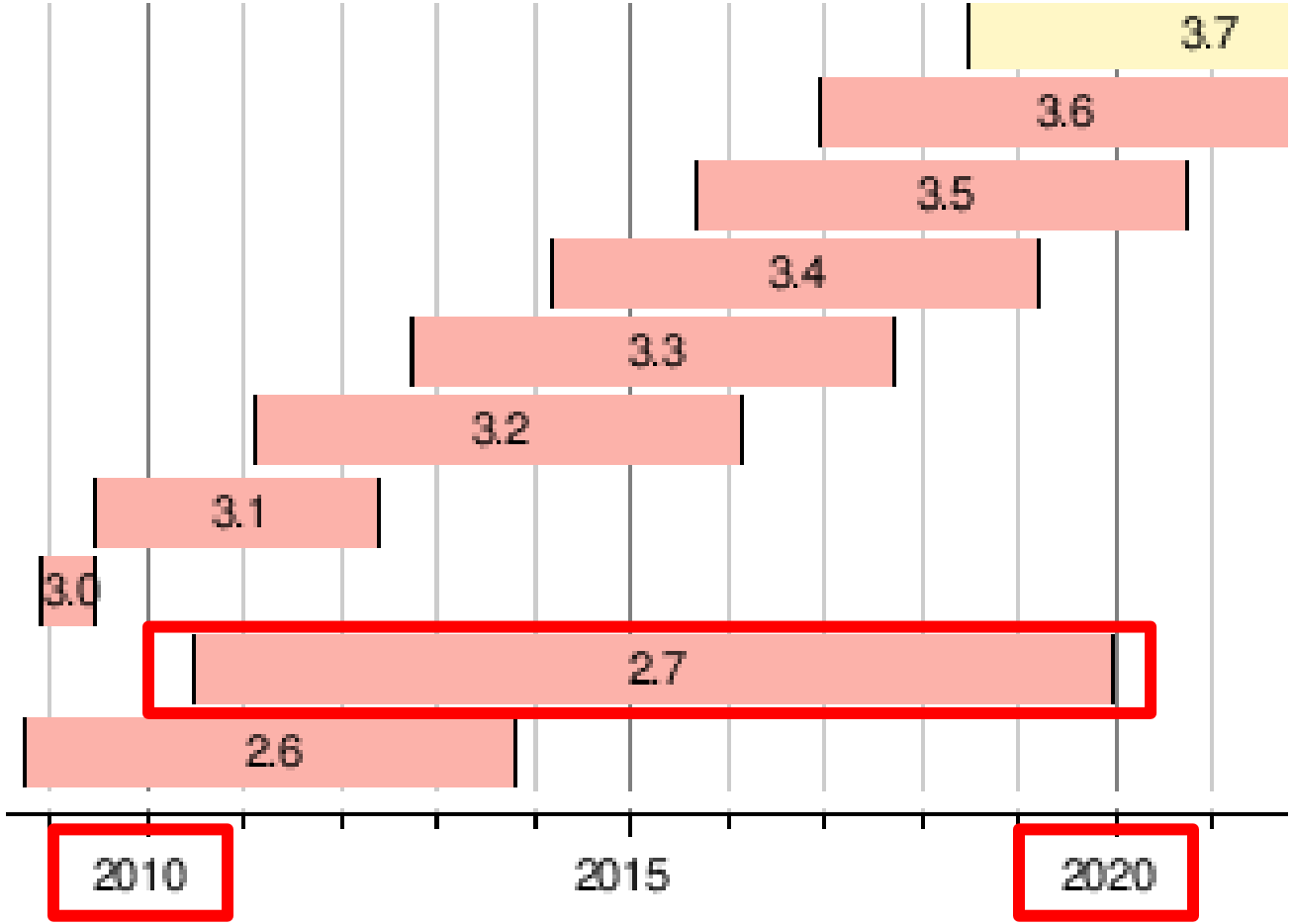
Answered: 1,676 Skipped: 50

Q15 Besides C++, what programming languages/environments do you use in your current and recent projects? (select all that apply)

Answered: 1,676 Skipped: 50




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)

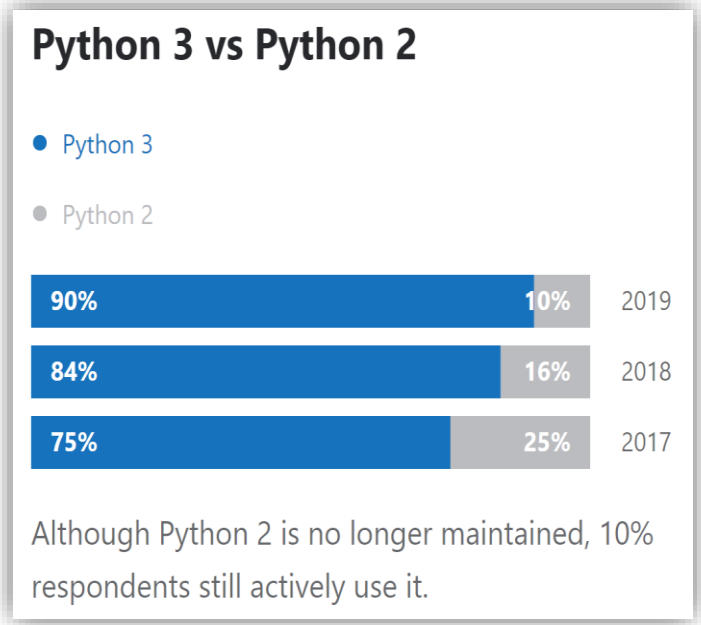
2017: Most Python code still written in "2∩3"

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)

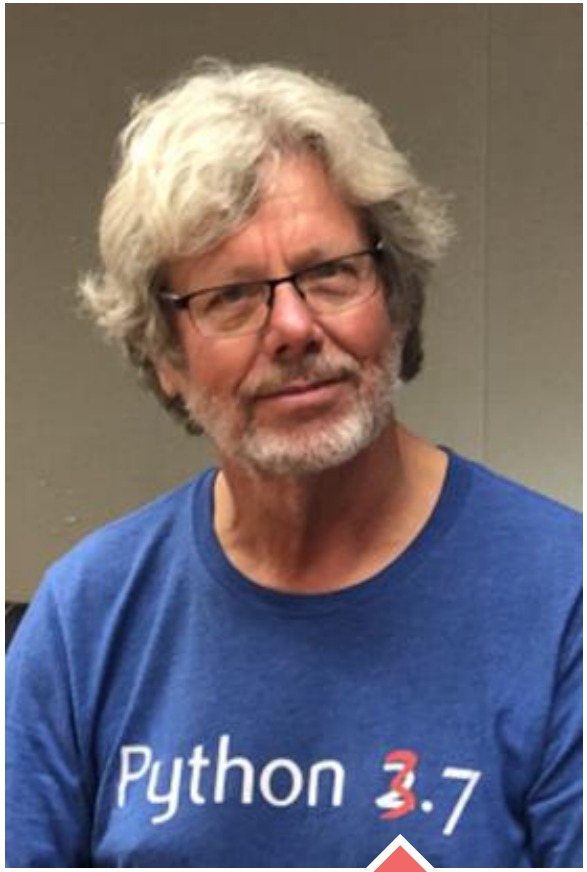
2017: Most Python code still written in "2∩3"

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)



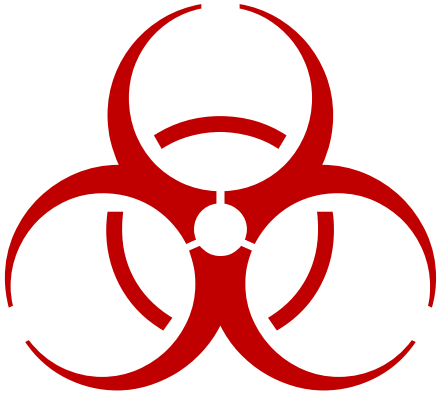
Version	Latest micro version	Release date	End of full support	End of security fixes
0.9	0.9.9[2]	1991-02-20[2]	1993-07-29[a][2]	
1.0	1.0.4[2]	1994-01-26[2]	1994-02-15[a][2]	
1.1	1.1.1[2]	1994-10-11[2]	1994-11-10[a][2]	
1.2		1995-04-13[2]	Unsupported	
1.3		1995-10-13[2]	Unsupported	
1.4		1996-10-25[2]	Unsupported	
1.5	1.5.2[42]	1998-01-03[2]	1999-04-13[a][2]	
1.6	1.6.1[42]	2000-09-05[43]	2000-09[a][42]	
2.0	2.0.1[44]	2000-10-16[45]	2001-06-22[a][44]	
2.1	2.1.3[44]	2001-04-15[46]	2002-04-09[a][44]	
2.2	2.2.3[44]	2001-12-21[47]	2003-05-30[a][44]	
2.3	2.3.7[44]	2003-06-29[48]	2008-03-11[a][44]	
2.4	2.4.6[44]	2004-11-30[49]	2008-12-19[a][44]	
2.5	2.5.6[44]	2006-09-19[50]	2011-05-26[a][44]	
2.6	2.6.9[27]	2008-10-01[27]	2010-08-24[b][27]	2013-10-29[27]
2.7	2.7.18[32]	2010-07-03[32]	2020-01-01[c][32]	
3.0	3.0.1[44]	2008-12-03[27]	2009-06-27[51]	

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

“Everything!”

Version	Latest micro version	Release date	End of full support	End of security fixes
2.6	2.6.9[27]	2008-10-01[27]	2010-08-24[b][27]	2013-10-29[27]
2.7	2.7.18[32]	2010-07-03[32]	2020-01-01[c][32]	
3.0	3.0.1[44]	2008-12-03[27]	2009-06-27[51]	
3.1	3.1.5[52]	2009-06-27[52]	2011-06-12[53]	2012-04-06[52]
3.2	3.2.6[54]	2011-02-20[54]	2013-05-13[b][54]	2016-02-20[54]
3.3	3.3.7[55]	2012-09-29[55]	2014-03-08[b][55]	2017-09-29[55]
3.4	3.4.10[56]	2014-03-16[56]	2017-08-09[57]	2019-03-18[a][56]
3.5	3.5.10[58]	2015-09-13[58]	2017-08-08[59]	2020-09-30[58]
3.6	3.6.15[60]	2016-12-23[60]	2018-12-24[b][60]	2021-12-23[60]
3.7	*3.7.16*[61]	2018-06-27[61]	2020-06-27[b][61]	2023-06-27[61]
3.8	*3.8.16*[62]	2019-10-14[62]	2021-05-03[b][62]	2024-10[62]
3.9	*3.9.16*[63]	2020-10-05[63]	2022-05-17[b][63]	2025-10[63][64]
3.10	3.10.11[65]	2021-10-04[65]	2023-05[65]	2026-10[65]
3.11	3.11.3[66]	2022-10-24[66]	2024-05[66]	2027-10[66]
3.12	3.12.0[67]	2023-10-02[67]	2025-05[67]	2028-10[67]

Version	Latest micro version	Release date	End of full support	End of security fixes
2.6	2.6.9[27]	2008-10-01[27]	2010-08-24[b][27]	2013-10-29[27]
2.7	2.7.18[32]	2010-07-03[32]	2020-01-01[c][32]	



(1) lab leak?

(2) animal spillover?

... (3) Python 2 end of security fixes?

C99 `_Complex` and VLAs

C99 (1999)

Some additions were controversial and resisted

Fun fact: CPython only allows selected C99 features because of lack of portable compiler support

C11 (2011): Made `_Complex` and VLAs "conditionally supported"
⇒ **optional**, not required for conformance

This third edition cancels and replaces the second edition, ISO/IEC 9899:1999, as corrected by ISO/IEC 9899:1999/Cor 1:2001, ISO/IEC 9899:1999/Cor 2:2004, and ISO/IEC 9899:1999/Cor 3:2007. Major changes from the previous edition include:

— conditional (optional) features (including some that were previously mandatory)

C++: API (ISO) and ABI (vendor) compat

C++ is highly source & binary compatible (with C & C++prev)

"Stability is a feature." – Bjarne Stroustrup

C++11 (2008,11): Banned reference-counted `std::string`

→ ABI breaking change

2015, GCC 5.1: First shipped a conforming `std::string`

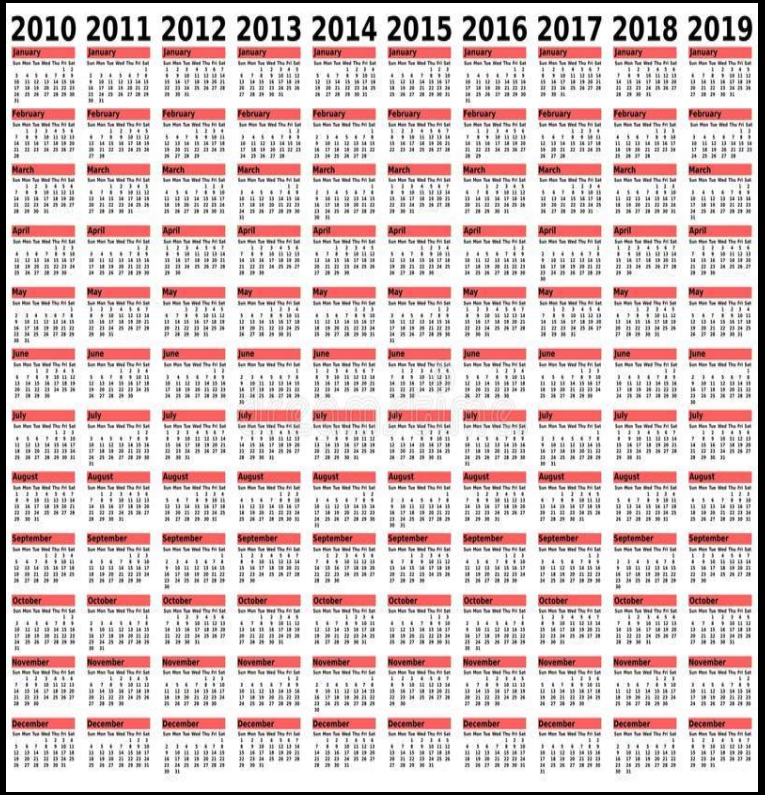
Then gradually adopted platform by platform (years)

2019, GCC 8 on Red Hat Enterprise Linux 8:

First turned the conforming string on by default

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)

"Every time you take a sharp turn, some people fall off"

– Unknown

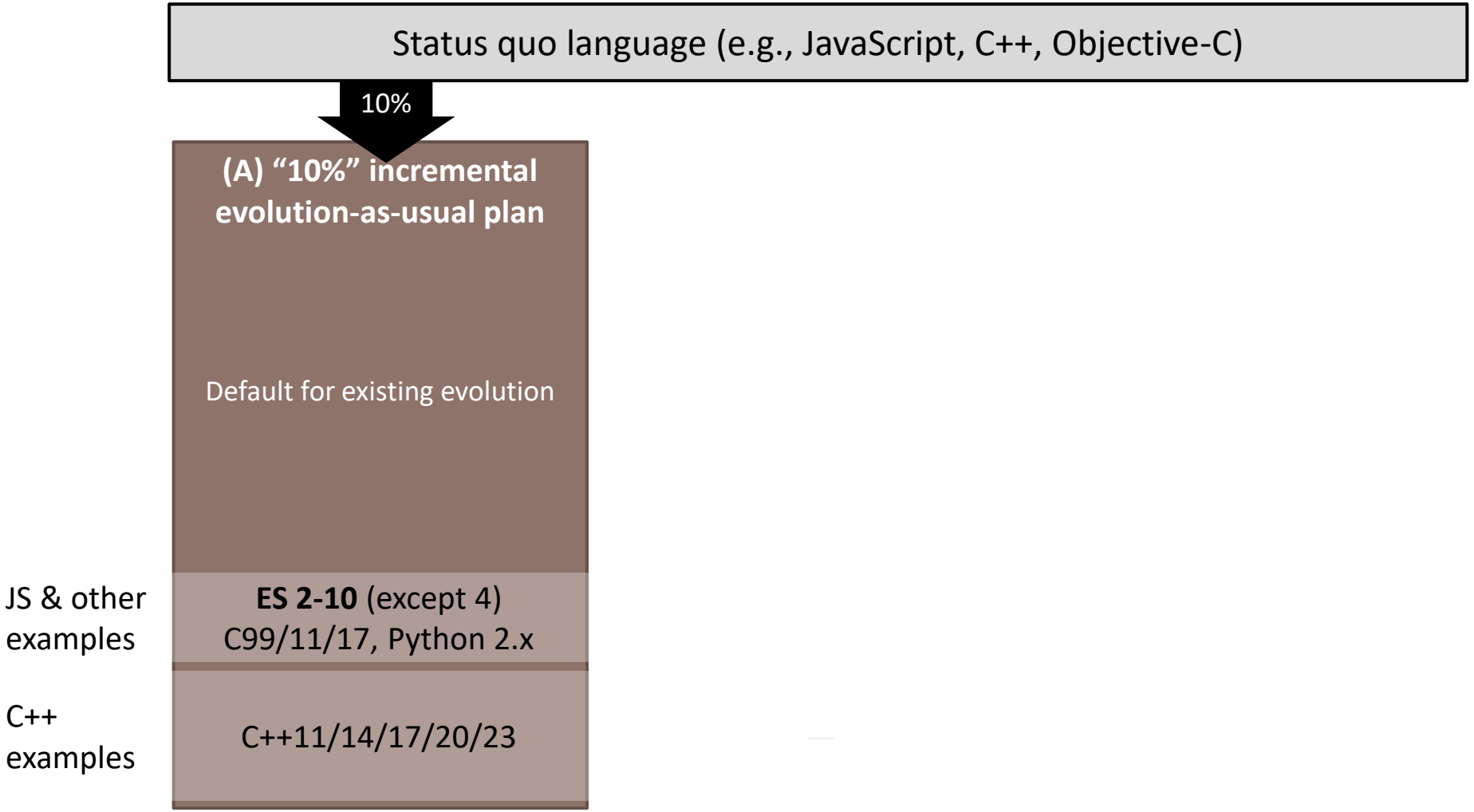
"Sometimes the truck falls over"

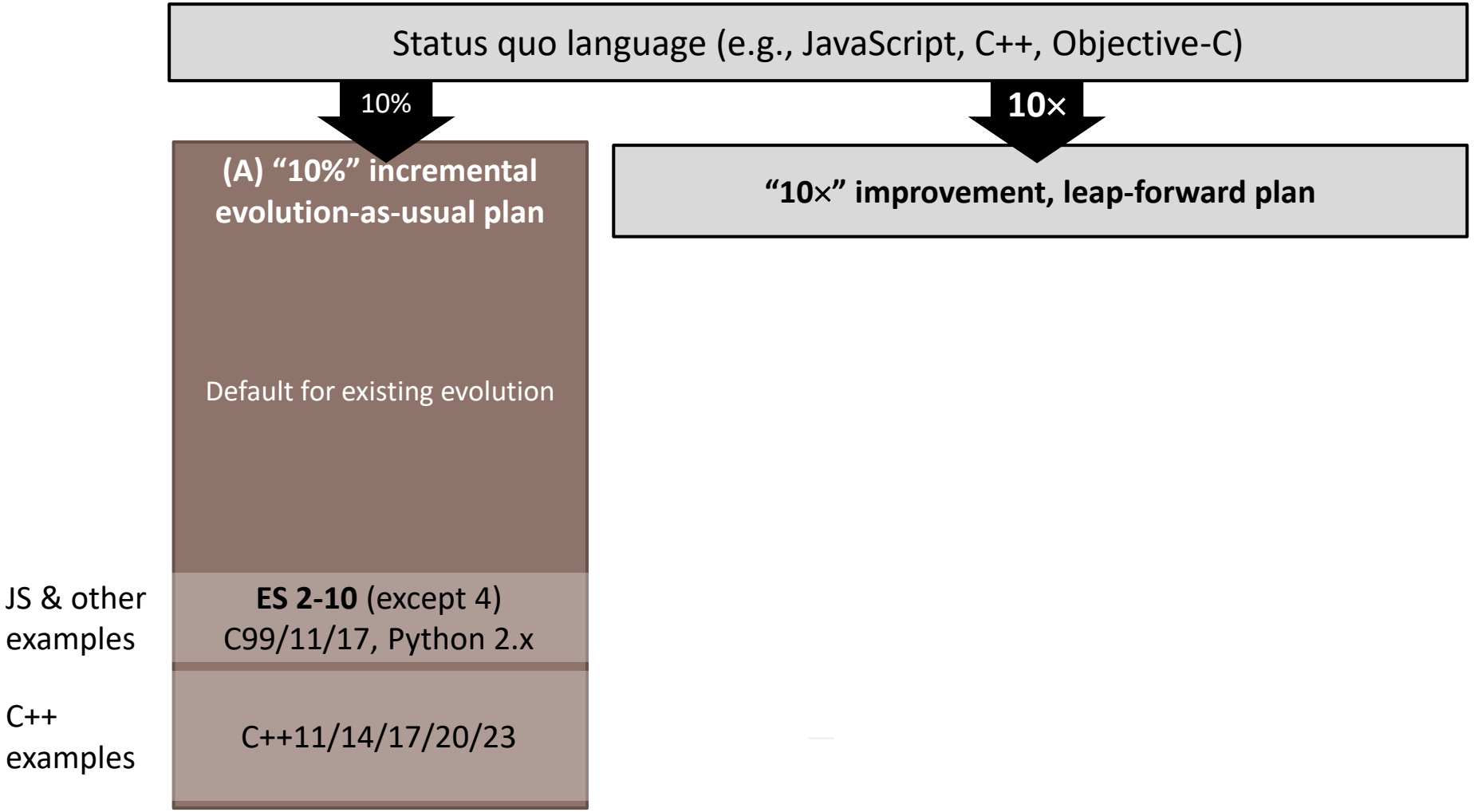
– Unknown2

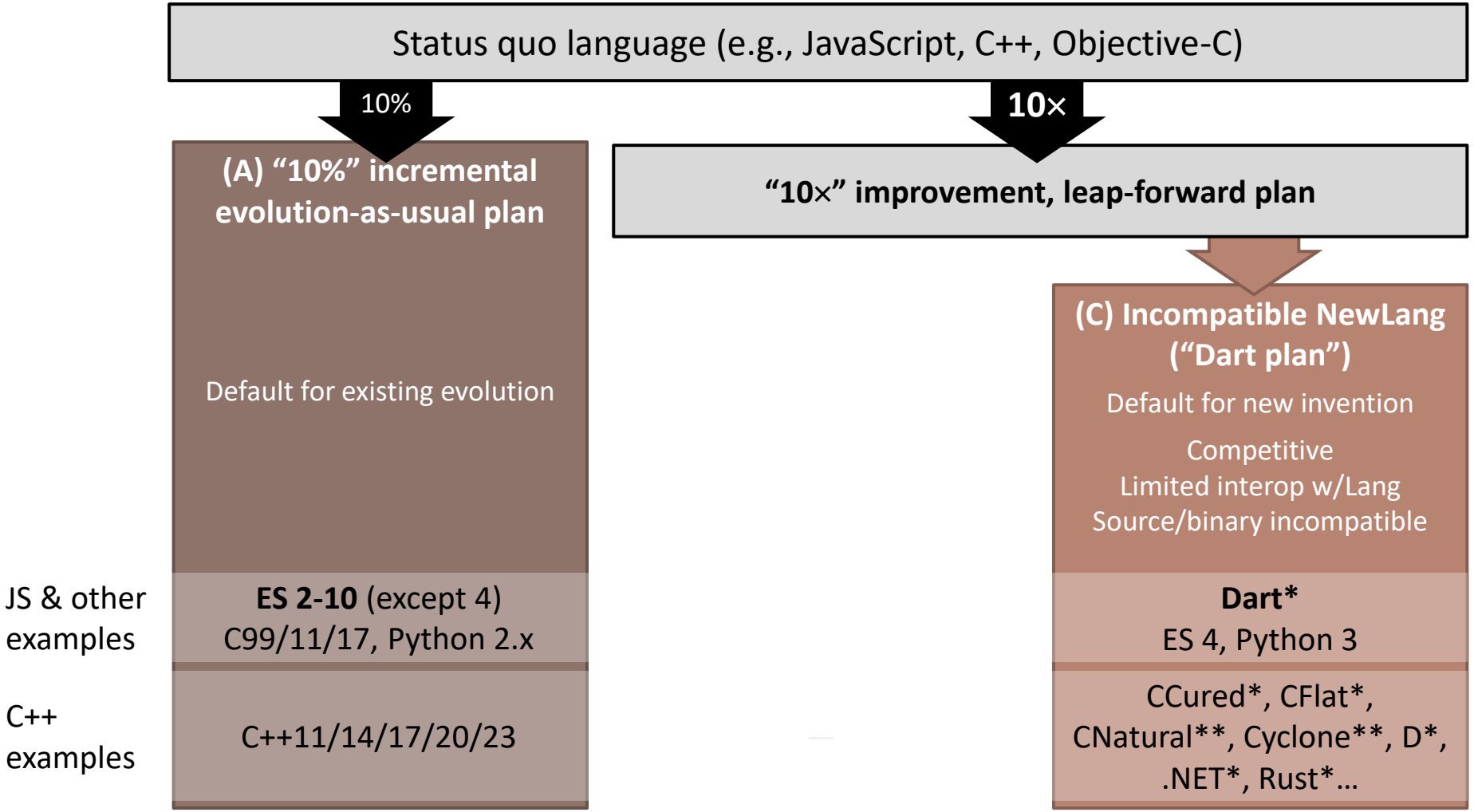
Status quo language (e.g., JavaScript, C++, Objective-C)

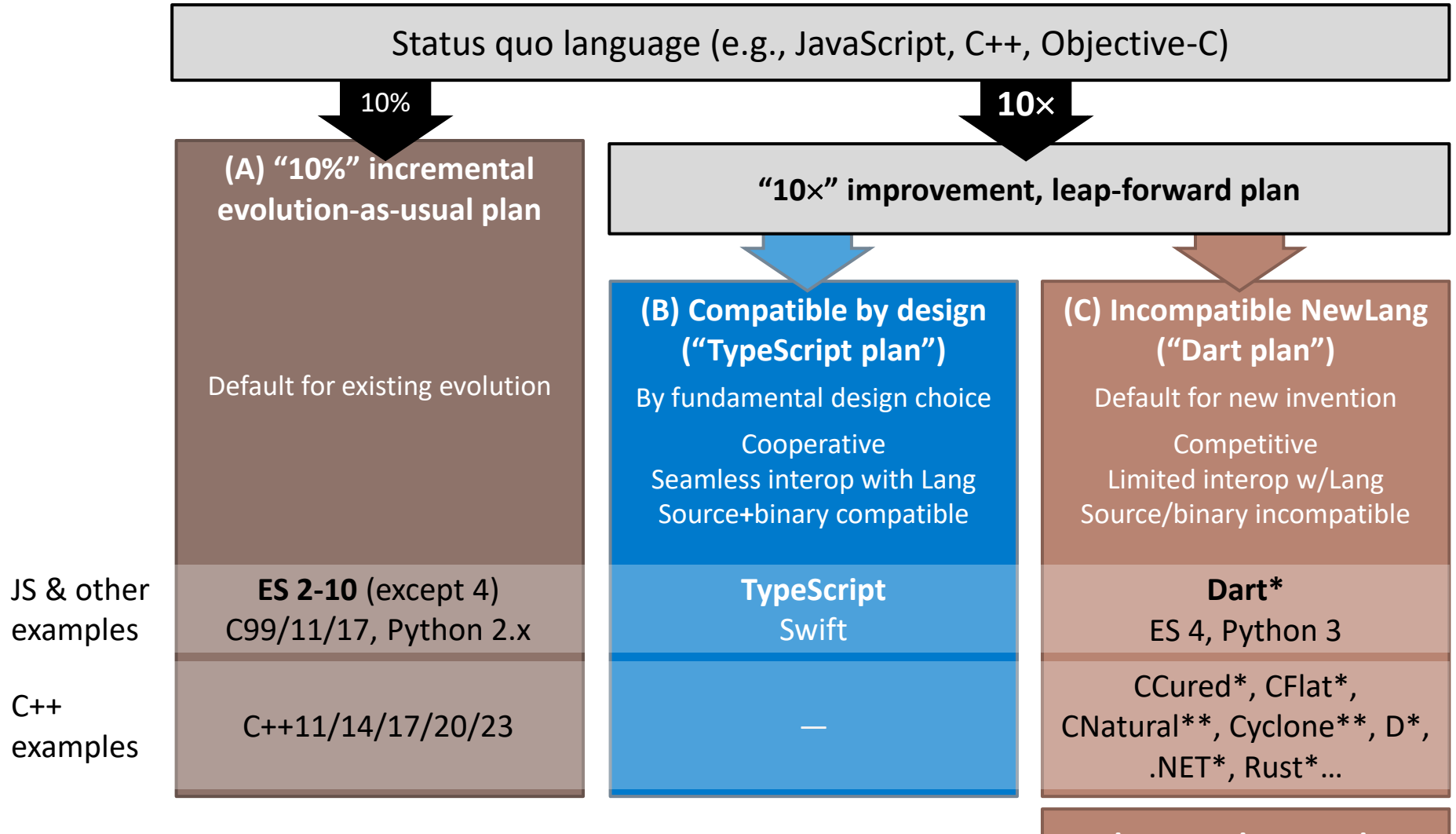
JS & other examples

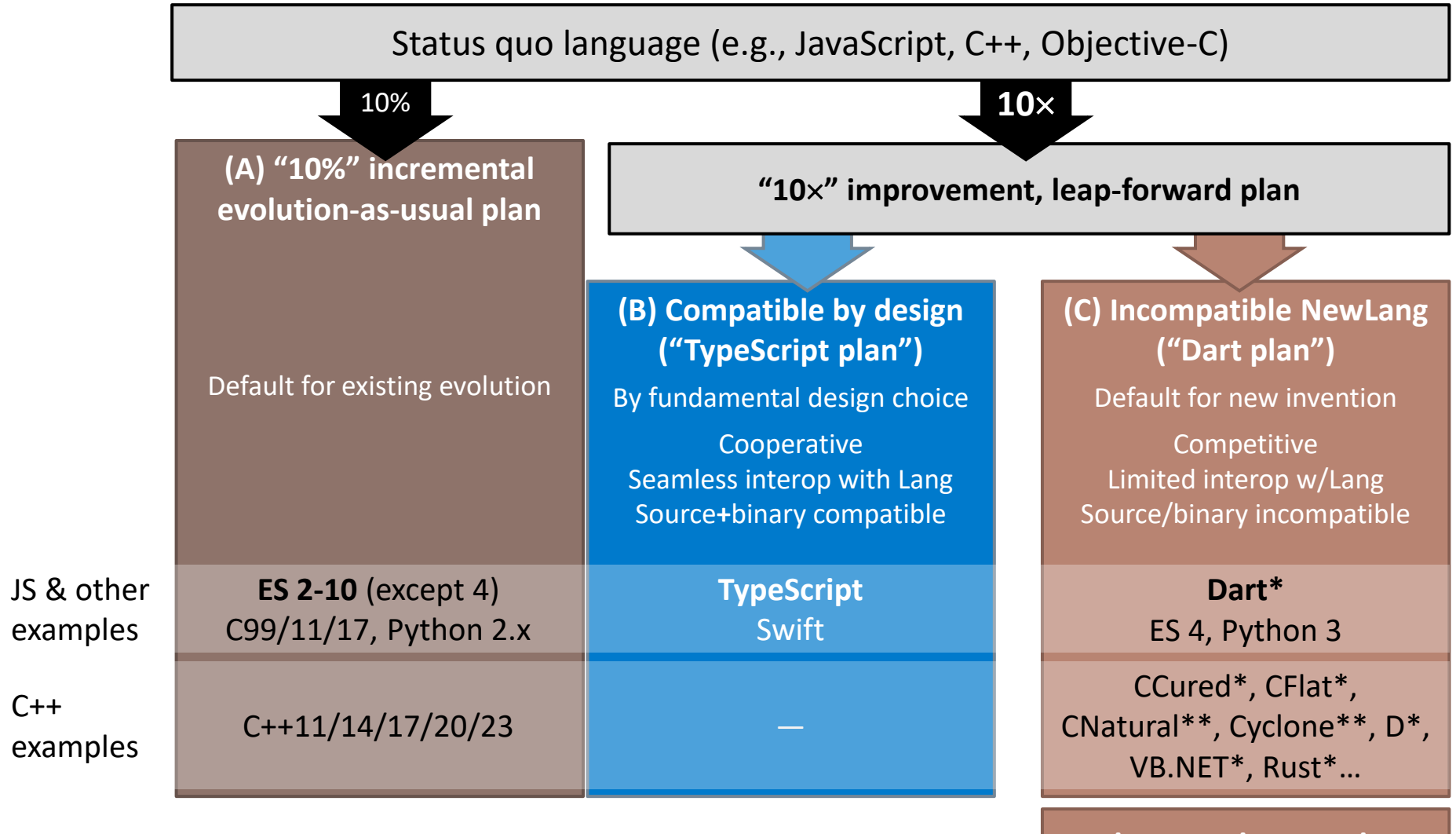
C++ examples


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

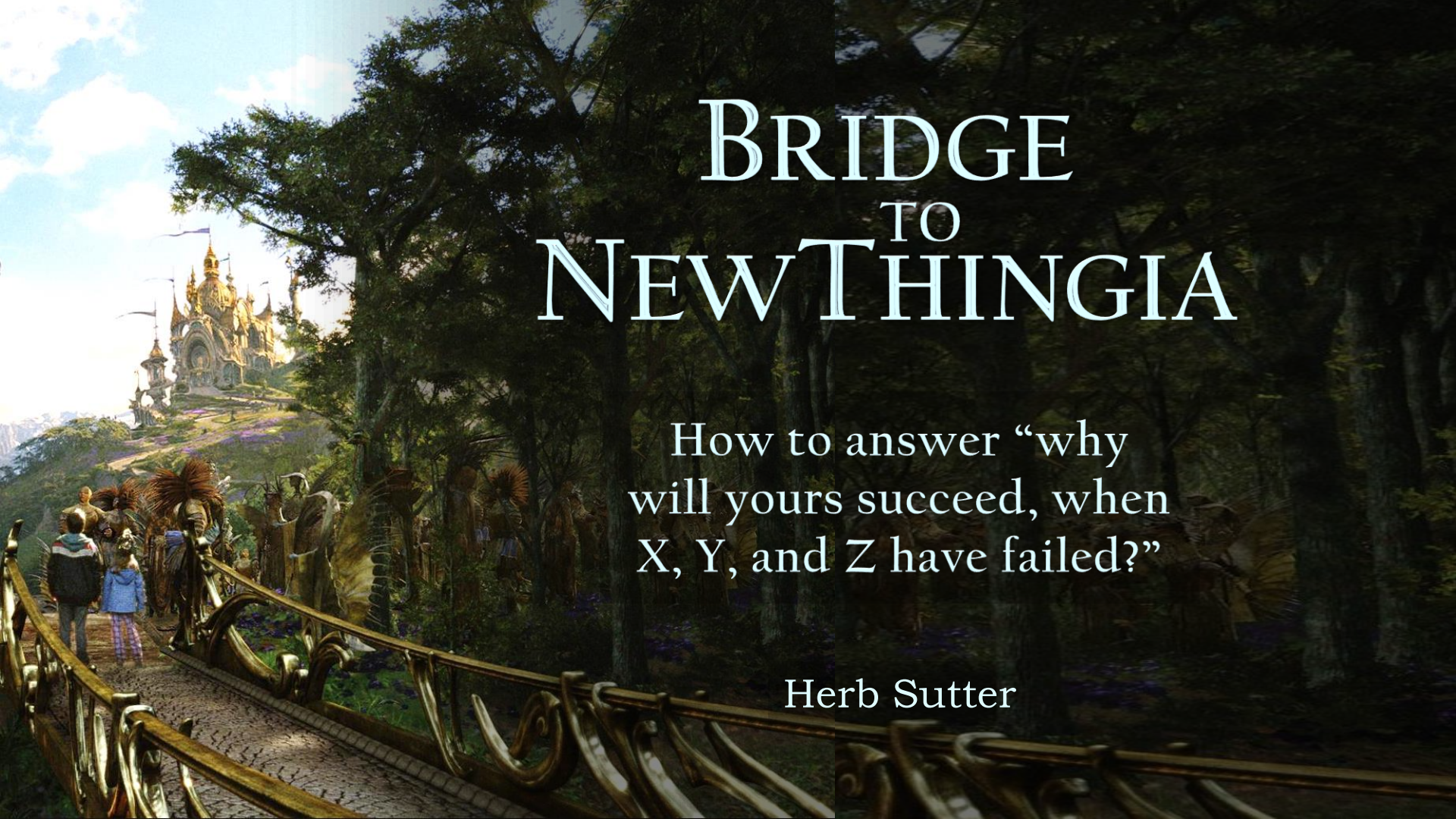

Status quo language (e.g., JavaScript, C++, Objective-C)
10%
10×
(A) “10%” incremental evolution-as-usual plan
“10×” improvement, leap-forward plan
Default for existing evolution
JS & other examples
ES 2-10 (except 4)
C99/11/17, Python 2.x
C++ examples
C++11/14/17/20/23


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*, .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


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

“Why will yours succeed, when X, Y, and Z have failed?”

1. **Value** to address known OldThing pain (and know OldThing’s value).

   Real pain needs little explanation. Existing value is often underestimated.

2. **Availability** wherever OldThing is used.

   Explicit design goal from the start, but can grow into it.

3. **Compatibility bridge.** Seamless backward interop with OldThing.

   Explicit design goal from the start. Hard to back into later.

   If you don’t, expect to slow your adoption down by ~10 years.

   Good: “I can use NewThing **side by side** in an OldThing project.”

   Grail: “I can **write 1 line** of NewThing inside OldThing and see benefit.”

From “Bridge to NewThingia”

Example: Successor to C++20? (C++23 or NewLang)

1. Value (solving what C++ pain?) + **2. Availability** (wherever (

From "Bridge to NewThingia"

"Why will your C++20 successor succeed when <*many*> haven't?"

Here's a differentiator that only C++next has tried … because it's legit hard …

3. Compatibility bridge. Seamless backward interop with C++.

Explicit design goal from the start. Hard to back into later.

If you don't, expect to slow your adoption down by ~10 years.

Good: "I can use NewLang **side by side** in a C++ project."

Grail: "I can **write 1 line** in NewLang inside a C++ file and see benefit."

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 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

2023

Fill in the blank: _____ for C++

Herb Sutter

C++ now

2023

Fill in the blank: ___Safety___ for C++

Herb Sutter

C++ now

2023

Fill in the blank: __Simplicity__ for C++

Herb Sutter

C++ now

2023

Fill in the blank: Compatibility for C++

Herb Sutter

C++ now

2023

Fill in the blank:
A TypeScript for C++

Herb Sutter

C++ now