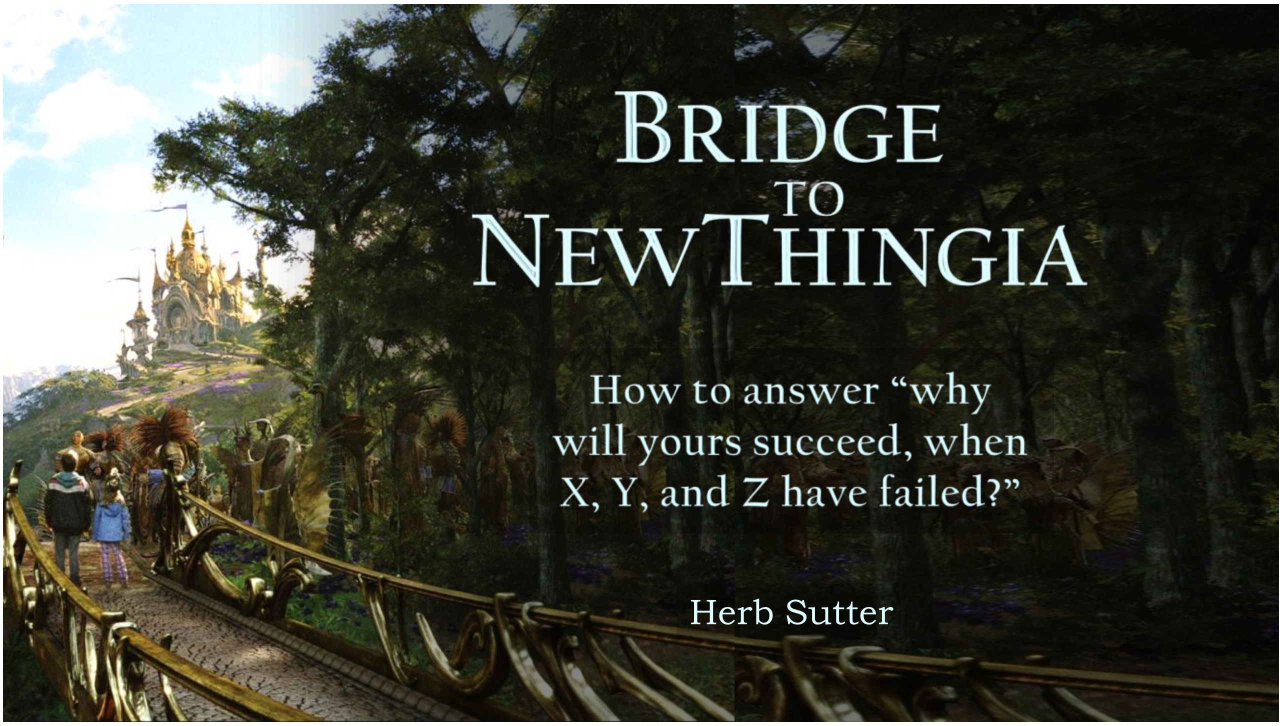


1

~~Out of scope~~
"Necessary but not sufficient"

Table stakes:
- funding
- team
- quality
- rollout
- marketing
- …

In scope
Strategy differentiators

What did SuccessfulThing **design for** so that it succeeded when X, Y, and Z didn't?

1. value
2. availability
3. **compatibility**

4

4

1. value
2. availability
3. compatibility

5

5

Definition: "NewThing"

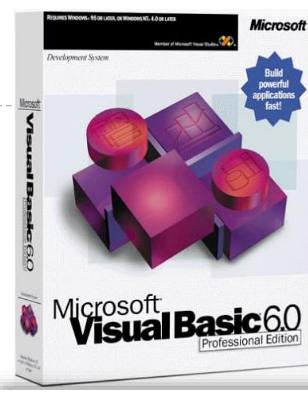
Can be a competing new product

- Examples: D, Go, Swift, Rust

Can be "vNext" of the same-brand product

- Same thing: For an established product, **vNext's biggest initial competitor is vPrev**
- Examples: C99, C++20, VB .NET, Python 3

"Same brand/name" doesn't mean users will accept as same and upgrade → **compatibility**



Visual Basic

From Wikipedia, the free encyclopedia

This article is about the Visual Basic language shipping with Microsoft Visual Studio 6.0 or earlier. For the Visual Basic language shipping with Microsoft Visual Studio .NET or later, see Visual Basic .NET.

6

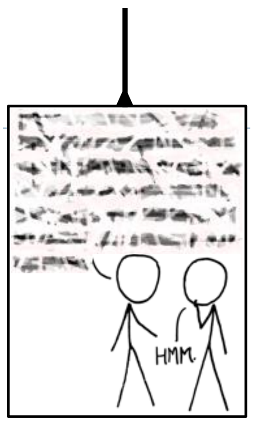
6

7

1. NewThing's **value**

Distinct articulable new value…

Clear and explainable in a 30-sec elevator pitch.



Adapted from imgs.xkcd.com/comics/freedom.png

8

8

1. NewThing's **value**

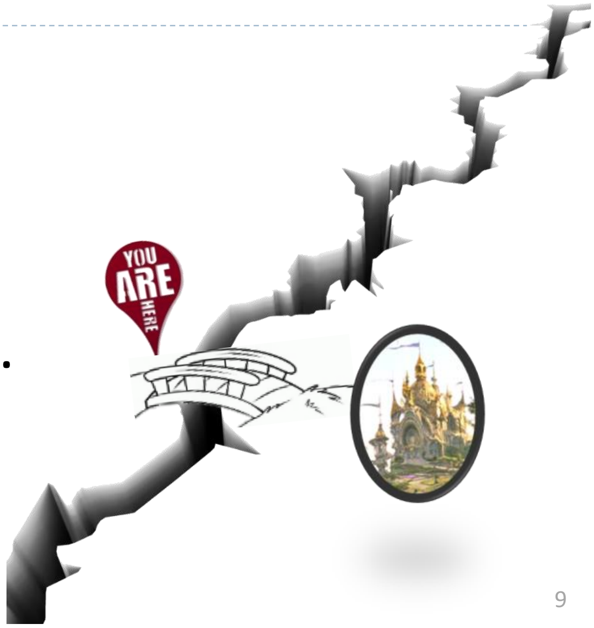
Distinct articulable new value…

Clear and explainable in a 30-sec elevator pitch.

… that solves **known pain points.**

Connection: To OldThing.

Orientation: "We aim to take you from here across this gap."



9

9

Example: JavaScript

Concise elevator pitch:

Well-known pain points with JavaScript

Dart

TS

10

10

Example: C++

Concise elevator pitch:

- Well-known pain points with C++

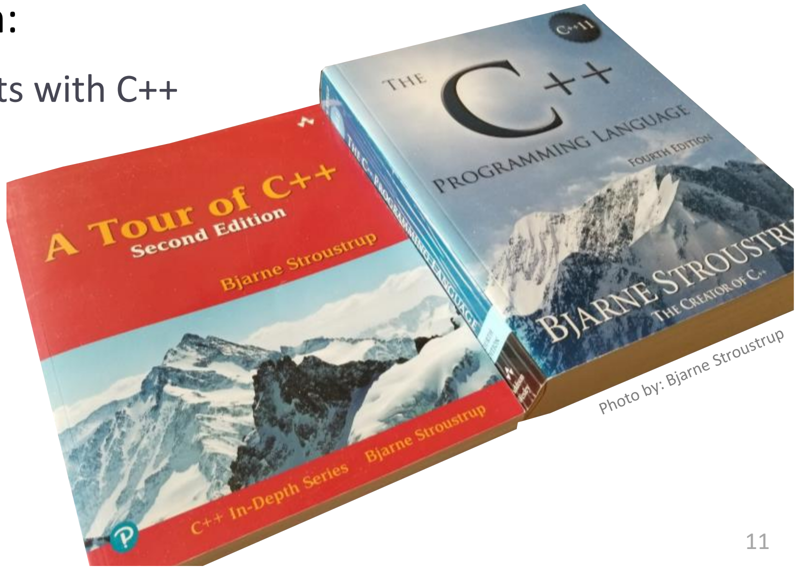


Photo by: Bjarne Stroustrup

11

11

12

13

2. NewThing **availability by design**

Requirement: Availability essentially anywhere OldThing is already used + easy to add to my project.

Grail: "Any OldThing project in any environment/platform can easily add NewThing."

14

14

2. NewThing **availability by design**

C: *Designed* to work on a wide range of hardware – "they said [portable performance] couldn't be done, and he did it."

Cfront: First C++ compiler *compiled to C*, usable "anywhere C is" – including the C optimizer (perf) and system linker (compat).

CPython: Reference compiler *written in C*, usable "anywhere C is."

TypeScript: Every TypeScript program *compiles to JavaScript*, runs on any JavaScript runtime.

Swift: Available to *every Xcode developer on every platform* that Objective-C supported (and now more).

Roslyn vNext C# compiler: Available to *every Visual Studio developer on every platform* Visual C# supported.

Cfront

CPython

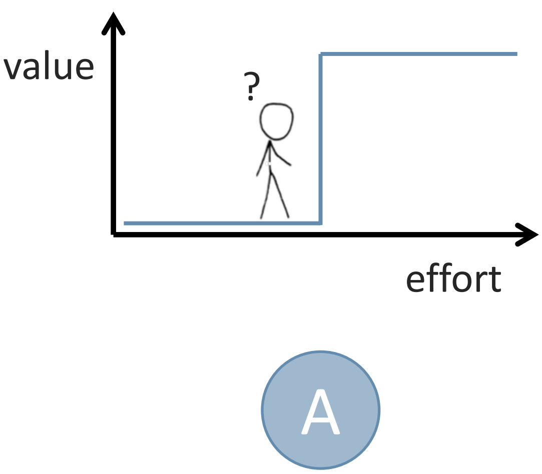
15

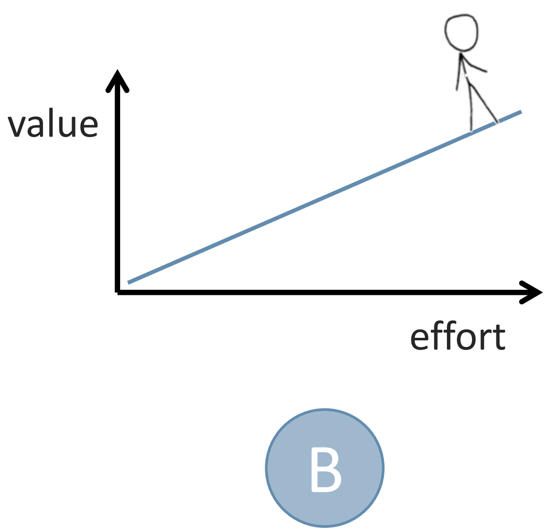
15

16

17

Which adoption function would you prefer?





18

18

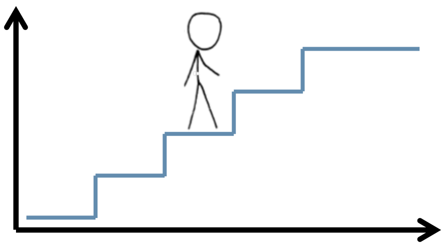
3. NewThing's **compatibility by design**

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."



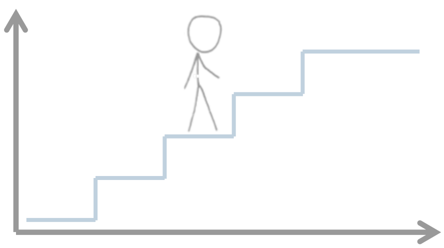
3. NewThing's **compatibility by design**

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."

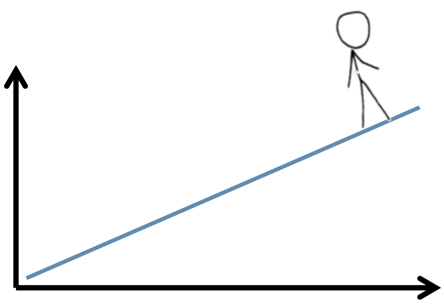


Grail: "An OldThing project can add NewThing **in one place** and start seeing benefit."

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.



21

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.

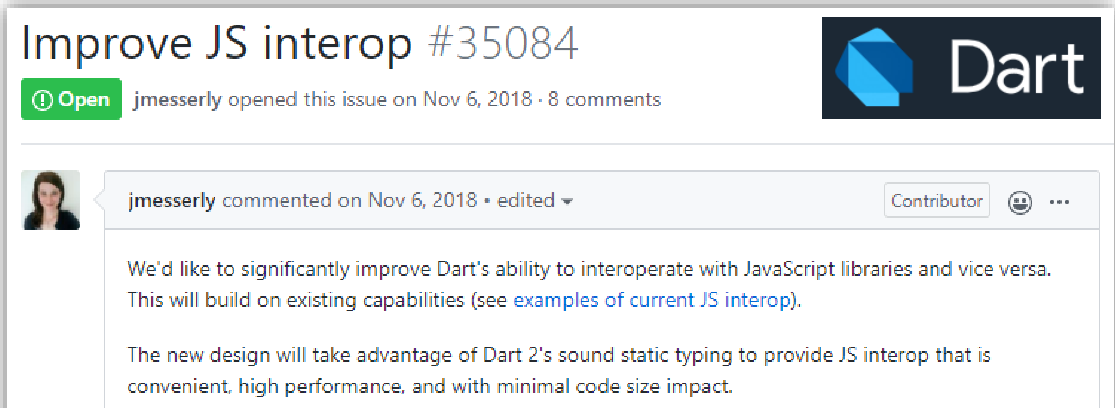
22

22

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

Compatibility requires strategic up-front design.

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


Improve JS interop #35084

Open jmesserly opened this issue on Nov 6, 2018 · 8 comments

jmesserly commented on Nov 6, 2018 • edited Contributor

We'd like to significantly improve Dart's ability to interoperate with JavaScript libraries and vice versa. This will build on existing capabilities (see examples of current JS interop).

The new design will take advantage of Dart 2's sound static typing to provide JS interop that is convenient, high performance, and with minimal code size impact.

23

23

Counterexample: Python

2008: Python 3

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

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

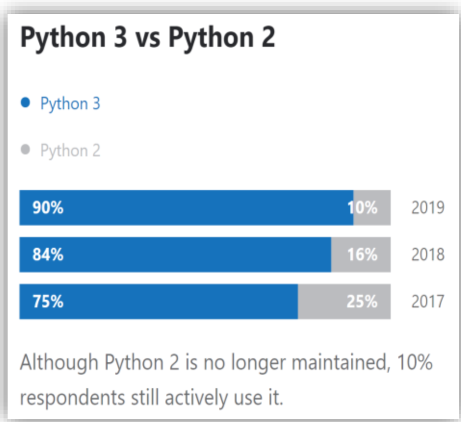
Manual migration + tools (2to3, Pylint, Futurize, Modernize, caniusepython3, tox, mypy)

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

2020: 2.x frozen and unsupported

~12-year transition

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



Source: JetBrains Python Developers Survey (Oct 2019)

24

24

Counterexample: C99

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)

25

25

Counterexample: C++11 string

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

- **Value** (efficient + machine-near) + **bridge** (compatibility)
- *"Stability is a feature."* – Bjarne Stroustrup

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

- ABI breaking change

GCC 5.1 (2015): First shipped a conforming `std::string`

- Then gradually adopted platform by platform (years)

GCC 8 on Red Hat Enterprise Linux 8 (2019):
First turned the conforming string on by default

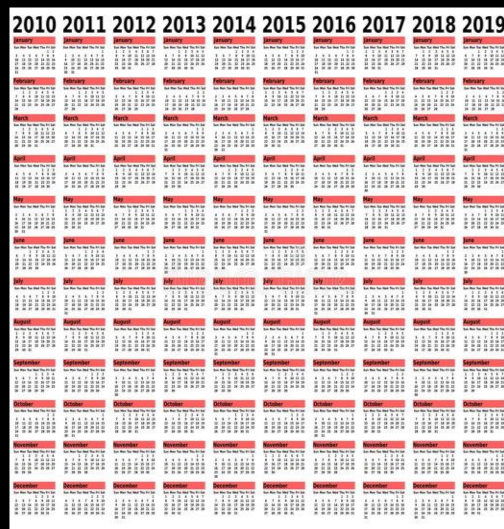
Red Hat

26

26

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

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



27

27

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

- C99 – ~12 years
  - Added _Complex and VLA in **1999**
  - Walked them back to “optional” in **2011**
- C++11 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)

28

28

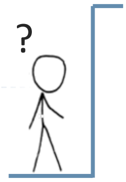

“Every time you take a sharp turn, some people fall off”
– Unknown
“Sometimes the truck falls over”
– Unknown2
29

29


A quick take on two common pitfalls…
30

30

Pit #1: Annotation requirements



Pitfall: Heavy annotation ⇒ **step function**

“Heavy” is often a low number, e.g., ~1 per KLOC

Esp. viral annotation (aka “color”) ⇒ **incompatible dialect**

Down: “A Red function can only call other Red functions”

Requires bottom-up annotation of the call tree

Lots of work for adopters + canonizes a Red dialect

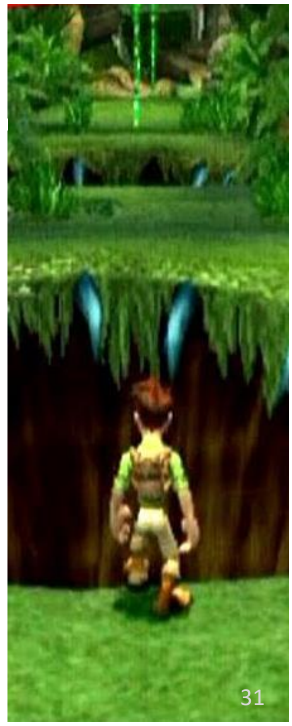
Example: `[[safe]]` functions

Up: “A Red function must be called by Red-aware functions”

Requires here-up annotation of the caller path

Example: Java checked exceptions
(common result: “`throws Exception`” opt-out)

Java



31

Counterexamples: “Safe C”

Many “Safe C” dialects focused on safe pointers (nullness, lifetime)

- 2001: **Cyclone** (Morrisett et al.)
  - 2006: v1.0
  - No longer supported
- 2002: **CCured** (Necula et al.)
  - Issues: **Annotations** + different representations
  - 2005 “Retrofitting”: infer annotations by whole program analysis + extend pointer type system
- 2015: **Checked C** (Tarditi et al.)
  - “Distinguished by its focus on **backward compatibility, incremental conversion**, …”

CYCLONE

CCured: Type-Safe Retrofitting of Legacy Code

Checked C

32

32

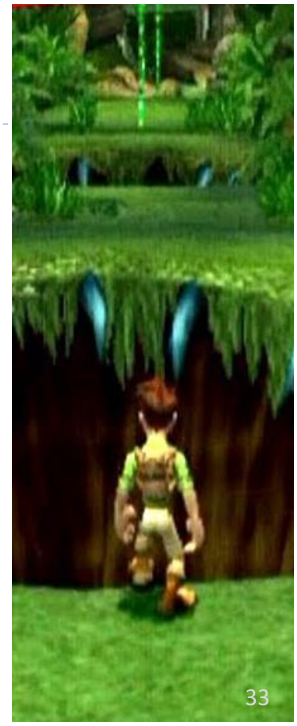
Pit #2: "False friend" incompatibility

3/2

Pitfall: "Same code means different things"
⇒ ambiguous, incompatible dialect

Poor choice 1: Allow both in the same source file
- Harder to **read** code – need to look at context, lose locality
- Harder for **tools** (e.g., refactoring)
- Example: `#pragma __new_syntax`, `version(2) {…}` block

Poor choice 2: Don't allow both in the same source file
- Can still allow both kinds of files in same project
- "Less harder" to read code and create tools
- Example: Different source extensions (e.g., `.lang2`)



33

So, then…

34

34

"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.

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."

35

35

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

"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.

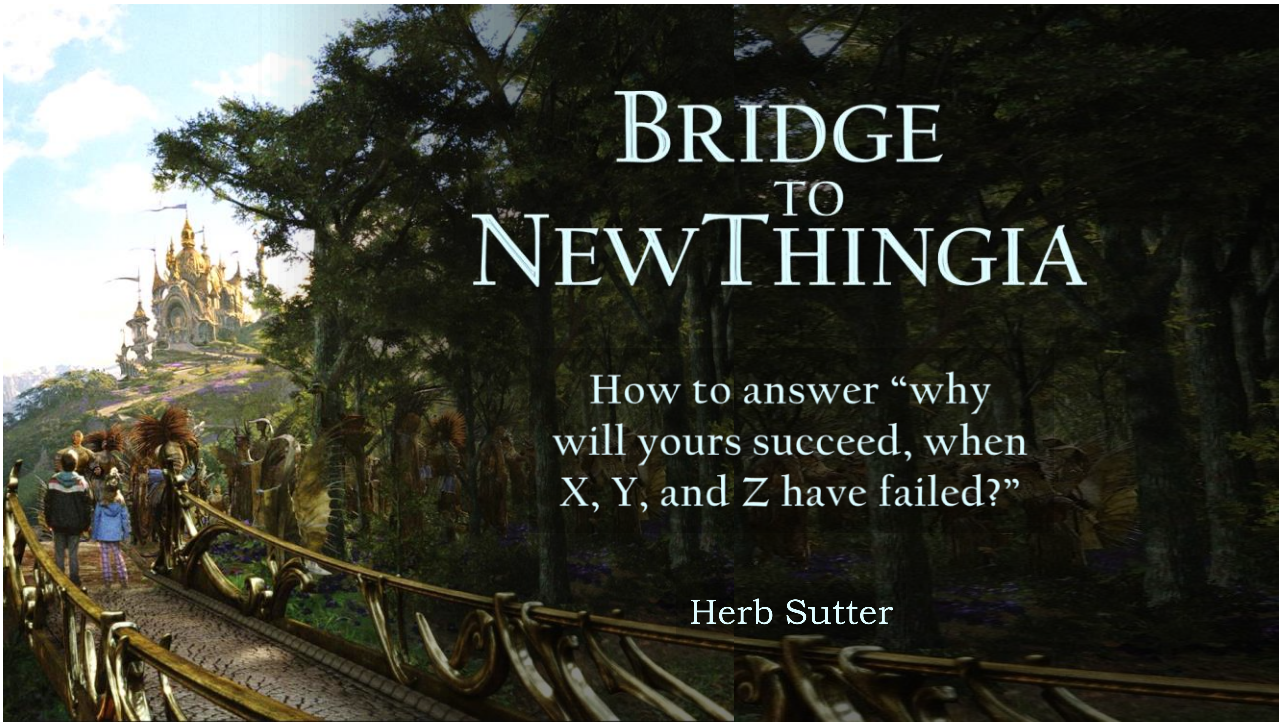
If you don't, expect to slow your adoption down by ~10 years.

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."

36

36


BRIDGE
TO
NEWTHINGIA
How to answer "why
will yours succeed, when
X, Y, and Z have failed?"
Herb Sutter

39