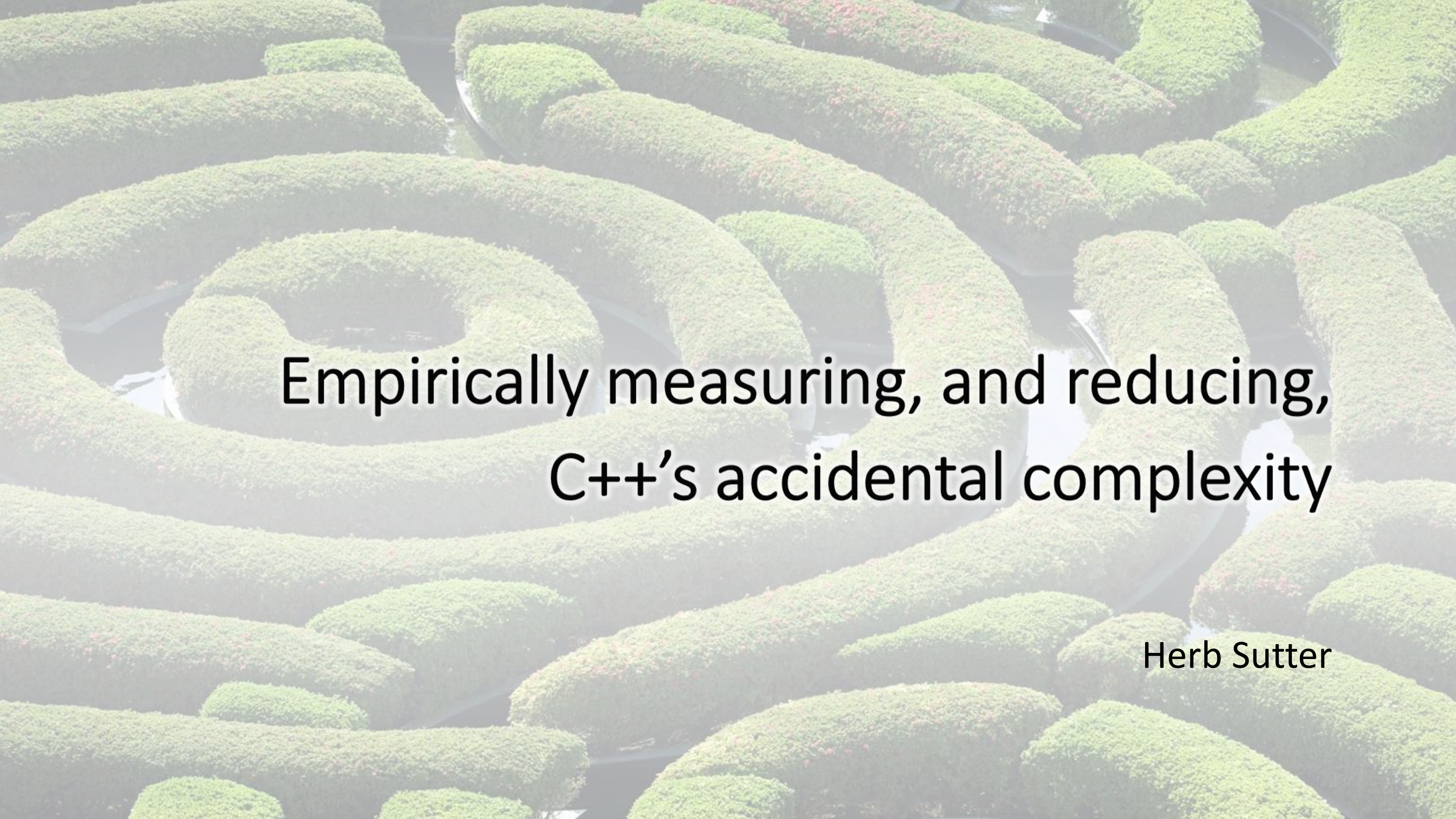


Empirically measuring, and reducing, C++’s accidental complexity

Herb Sutter

Why complexity matters

We’re “paying taxes” all the time

- Productivity
- Correctness and quality
- Tooling
- Teaching, learning, hiring, training

Common claim:
“C++ is too complex”

◀ This talk's contribution:
Empirically catalog,
classify, and count

Fred Brooks: Complexity

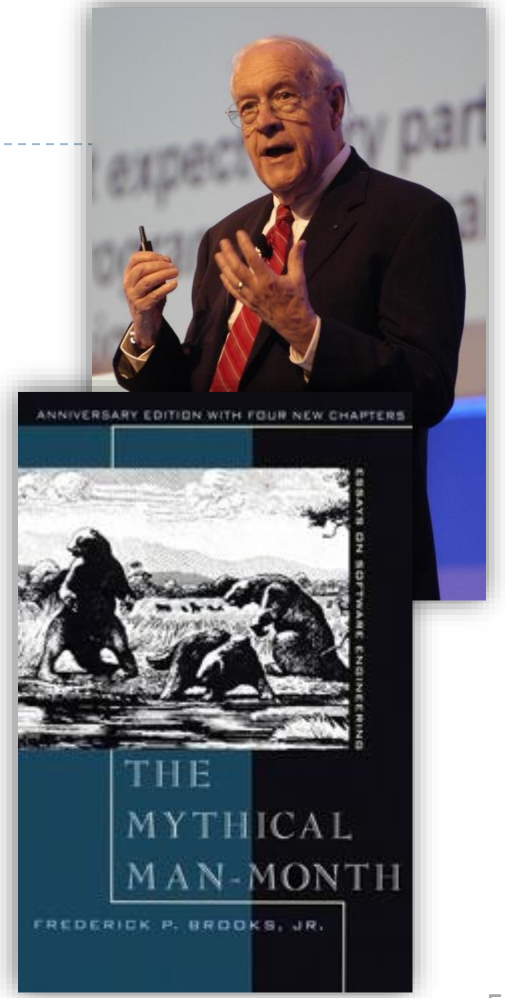
Essential complexity

Inherent in the problem,

present in any solution

Accidental complexity

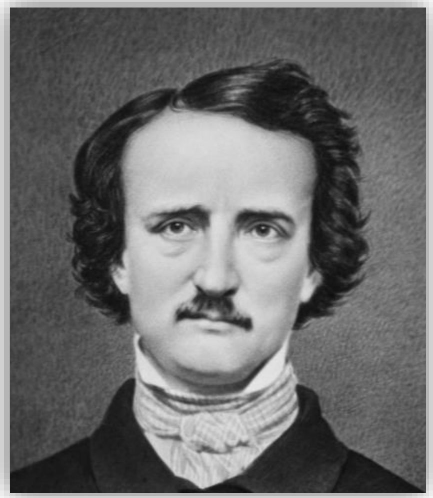
Artifact of a specific solution design


ANNIVERSARY EDITION WITH FOUR NEW CHAPTERS
ESSAYS ON SOFTWARE ENGINEERING
THE MYTHICAL MAN-MONTH
FREDERICK P. BROOKS, JR.

…what is only *complex* is mistaken (a not unusual error) for what is *profound*

— Edgar Allan Poe,

in “The Murders In the Rue Morgue”



Some of C++'s rich "guidance" corpus

Catalogued so far (638 rules)

Google: Abseil Tips

Meyers: Effective C++ Third Edition

Meyers: Effective Modern C++

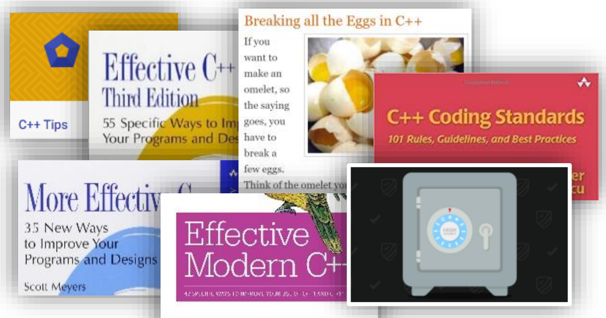
Meyers: More Effective C++

Meyers: "Breaking All the Eggs in C++"

Perforce: High Integrity C++ 4.0

Sutter & Alexandrescu:
C++ Coding Standards

(in progress) PVS-Studio


C++ Tips
Effective C++
Third Edition
55 Specific Ways to Im
Your Programs and Des
Breaking all the Eggs in C++
If you want to make an omelet, so the saying goes, you have to break a few eggs.
Think of the omelet yo
C++ Coding Standards
101 Rules, Guidelines, and Best Practices
More Effectiv
35 New Ways to Improve Your Programs and Designs
Scott Meyers
Effective Modern C+

Pending

CERT: CERT standard checks

Clang: clang-tidy checks

Lockheed-Martin & Stroustrup: Joint Strike Fighter Air Vehicle coding std. for C++, Rev C

(upcoming) MISRA: MISRA C++ 202x

Stroustrup & Sutter, eds.:
C++ Core Guidelines

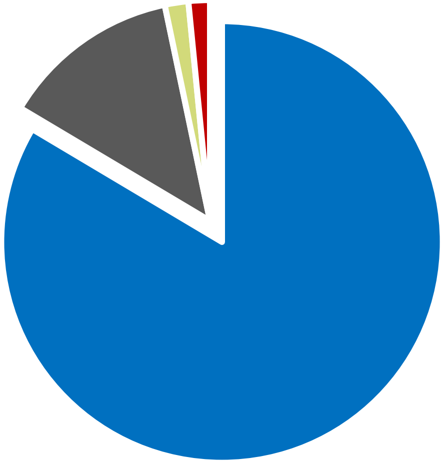
Sutter: Exceptional C++

Sutter: More Exceptional C++

Sutter: Exceptional C++ Style

Breakdown of first 638 rules catalogued

533	**language**
84	std:: library
11	general/local
10	wrong (IMO)

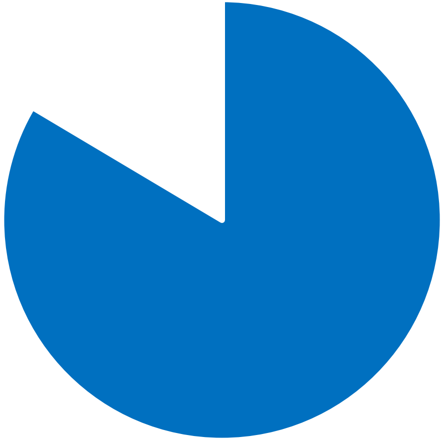


Even “wrong” was informative…

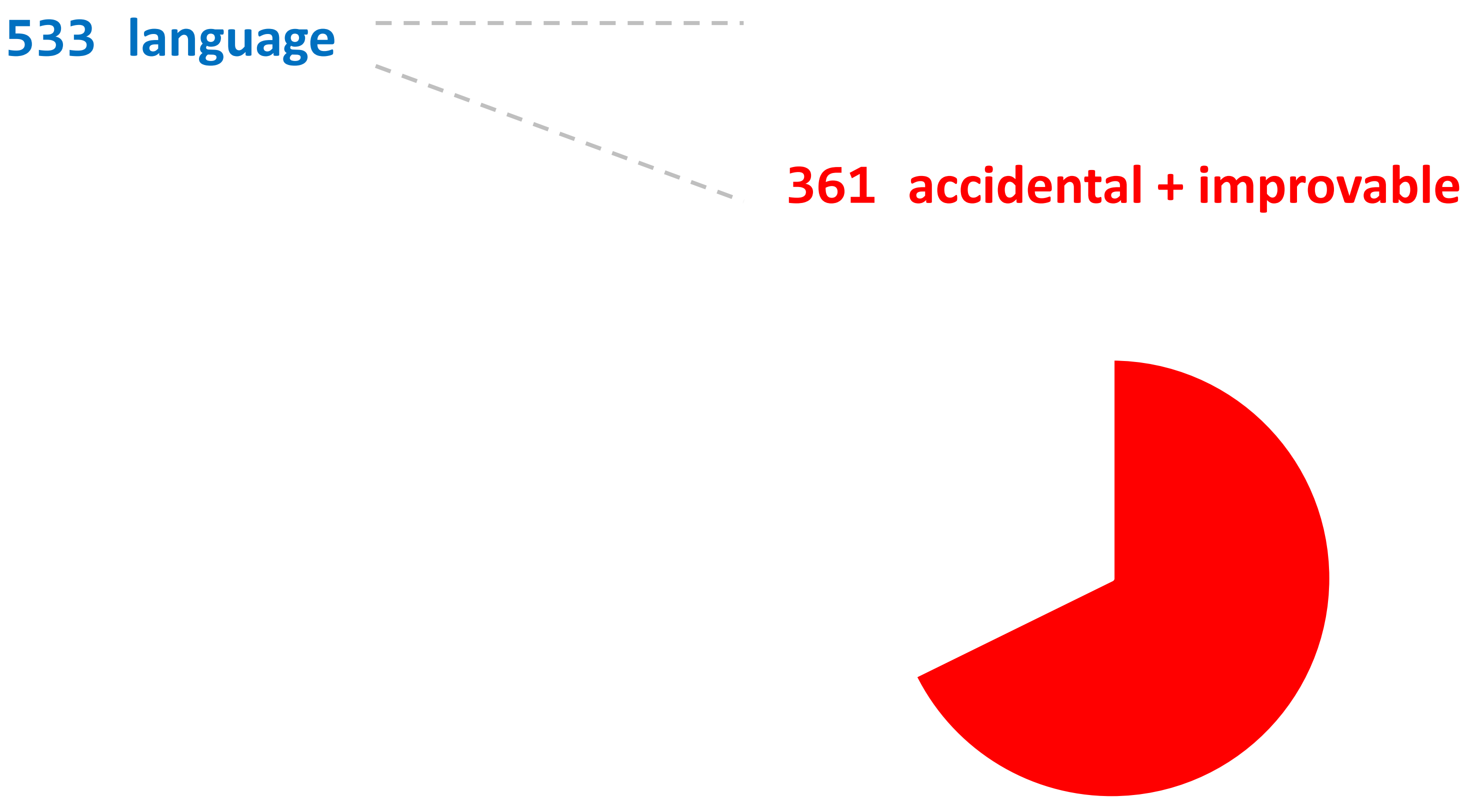
It often it arose because the language was complex / offered multiple ways to do a thing

Breakdown of first 638 rules catalogued

533 language



Breakdown of first 638 rules catalogued



533 language

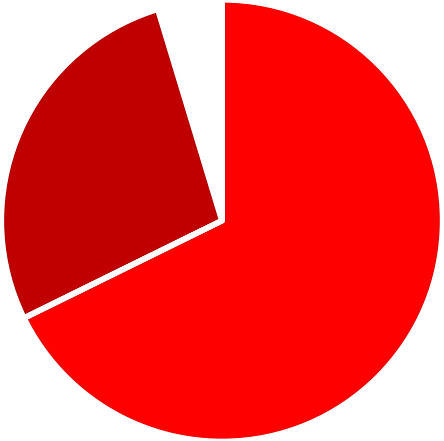
361 accidental + improvable

Breakdown of first 638 rules catalogued

533 language

147 'essential' + improvable

361 accidental + improvable



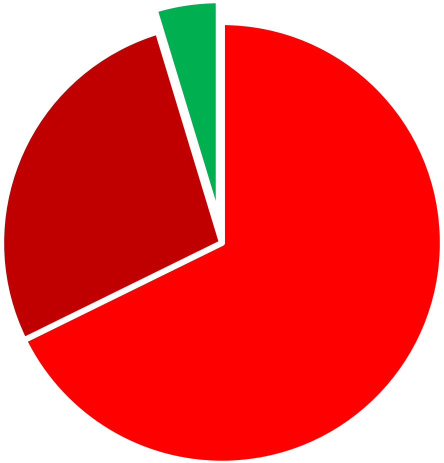
Breakdown of first 638 rules catalogued

533 language

25 essential + minimal

147 'essential' + improvable

361 accidental + improvable



Is there a “10× silver bullet”?

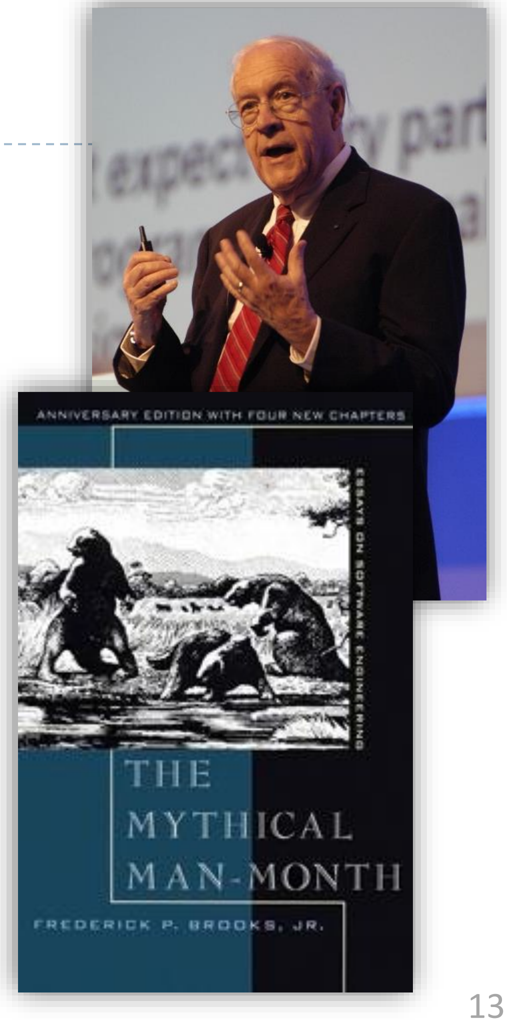
Brooks famously concluded: **“No silver bullet”**

> Conclusion: “There is no single development, in either technology or management technique, **which by itself promises even one order-of-magnitude improvement** within a decade in productivity, in reliability, in simplicity.”

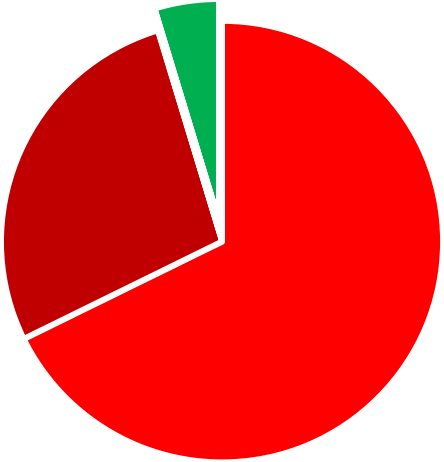
But, note Brooks’ premise:

> Premise: “How much of what software engineers now do is still devoted to the accidental, as opposed to the essential? **Unless it is more than 9/10 of all effort, shrinking the accidental activities to zero time will not give an order of magnitude improvement.**”
>
> Therefore: We have a large problem **and a large opportunity**.

“No Silver Bullet,” 1986; in *The Mythical Man-Month* Anniversary Ed.


ANNIVERSARY EDITION WITH FOUR NEW CHAPTERS
ESSAYS ON SOFTWARE ENGINEERING
THE MYTHICAL MAN-MONTH
FREDERICK P. BROOKS, JR.

Is there a "10× silver bullet"?



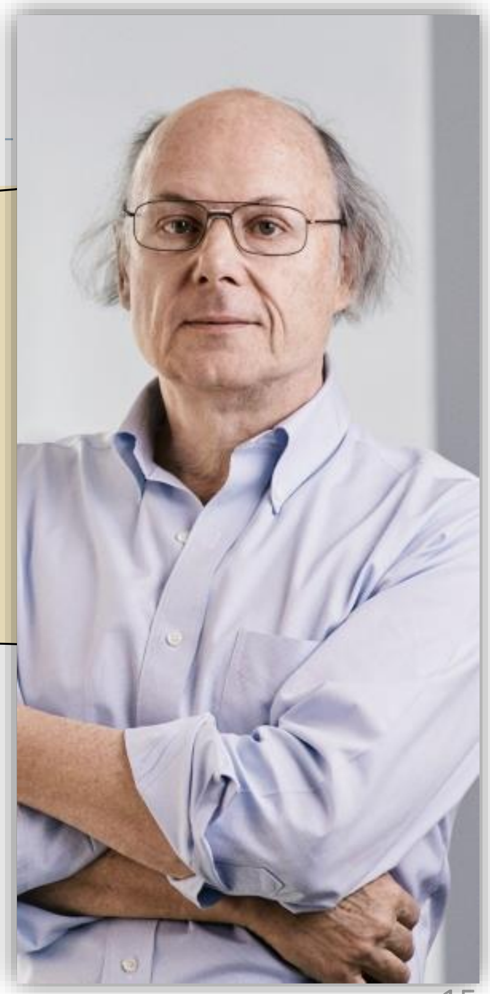
"**Unless it is more than 9/10 of all effort**, shrinking the accidental activities to zero time will not give an order of magnitude improvement."

Therefore: We have a large problem **and a large opportunity**.

Bjarne Stroustrup on “10×”

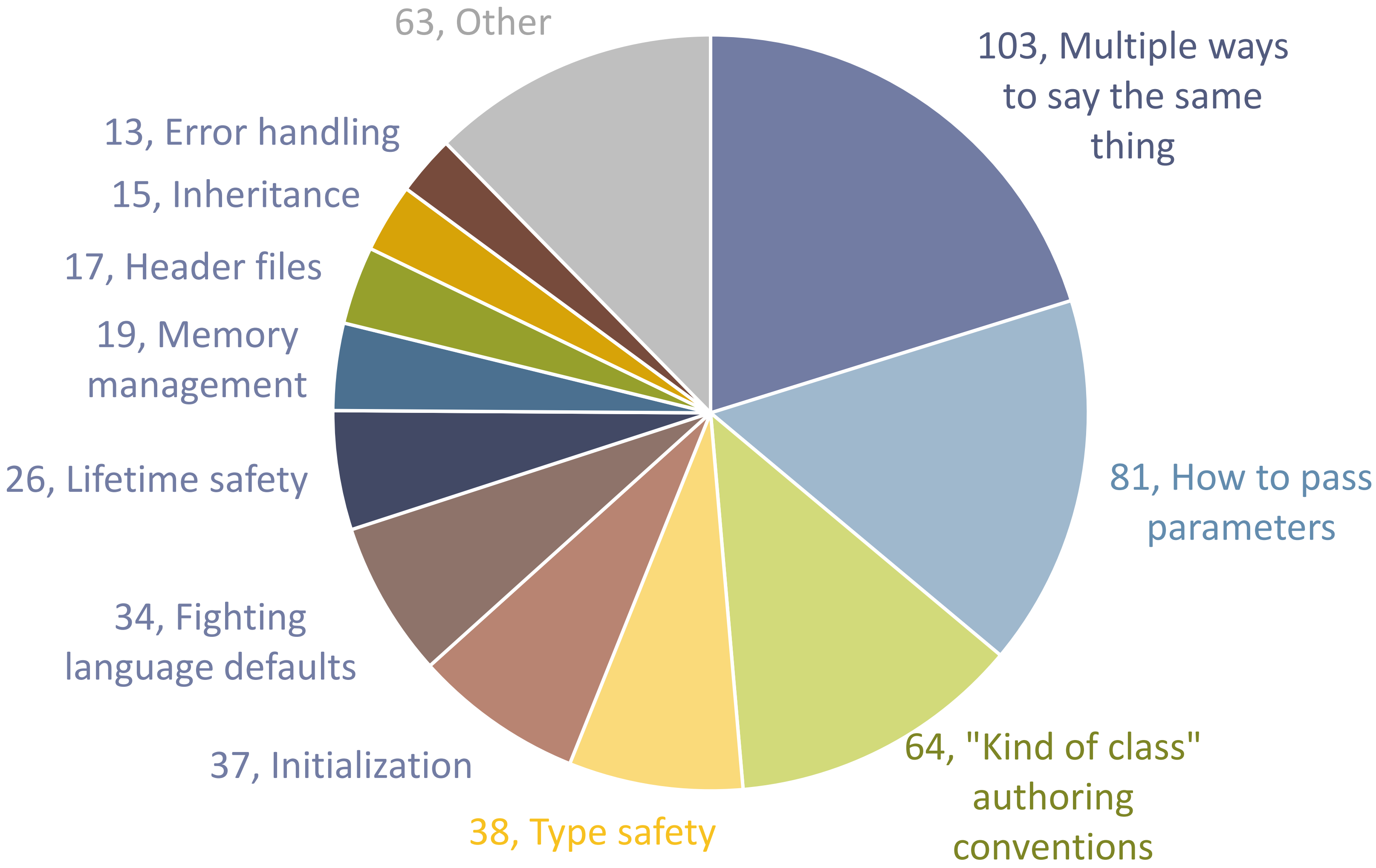
“**Inside C++, there is a much smaller and cleaner language** struggling to get out.”
— B. Stroustrup (D&E, 1994)

“**Say 10% of the size of C++**… Most of the simplification would come from **generalization**.”
— B. Stroustrup (ACM HOPL-III, 2007)



“**Unless it is more than 9/10 of all effort, shrinking the accidental activities to zero time will not give an order of magnitude improvement.**”

Therefore: We have a large problem **and a large opportunity**.


63, Other
103, Multiple ways to say the same thing
81, How to pass parameters
64, "Kind of class" authoring conventions
38, Type safety
37, Initialization
34, Fighting language defaults
26, Lifetime safety
19, Memory management
17, Header files
15, Inheritance
13, Error handling

Common claim:
“C++ is too complex”

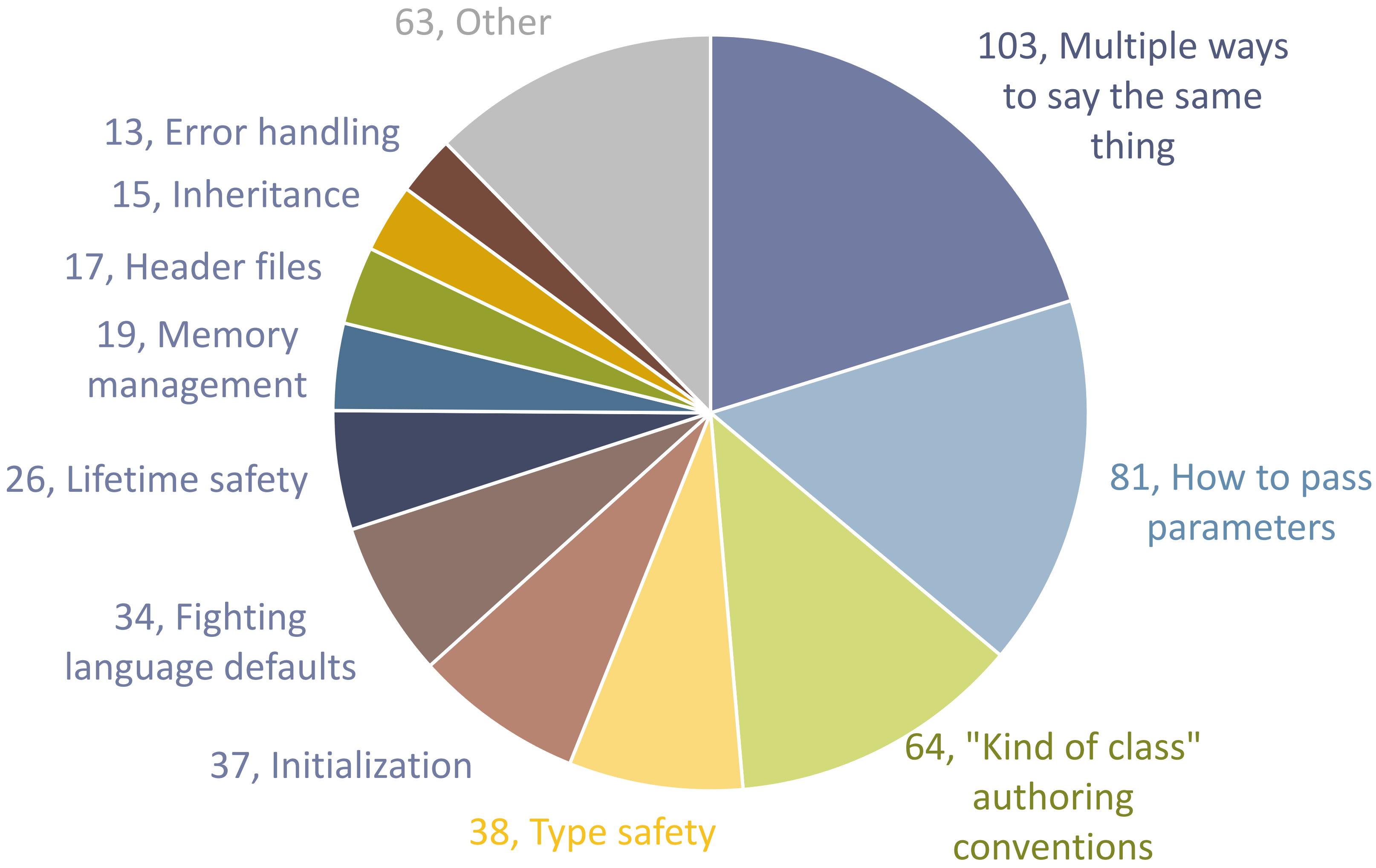
This talk's contribution:
Empirically catalog,
classify, and count

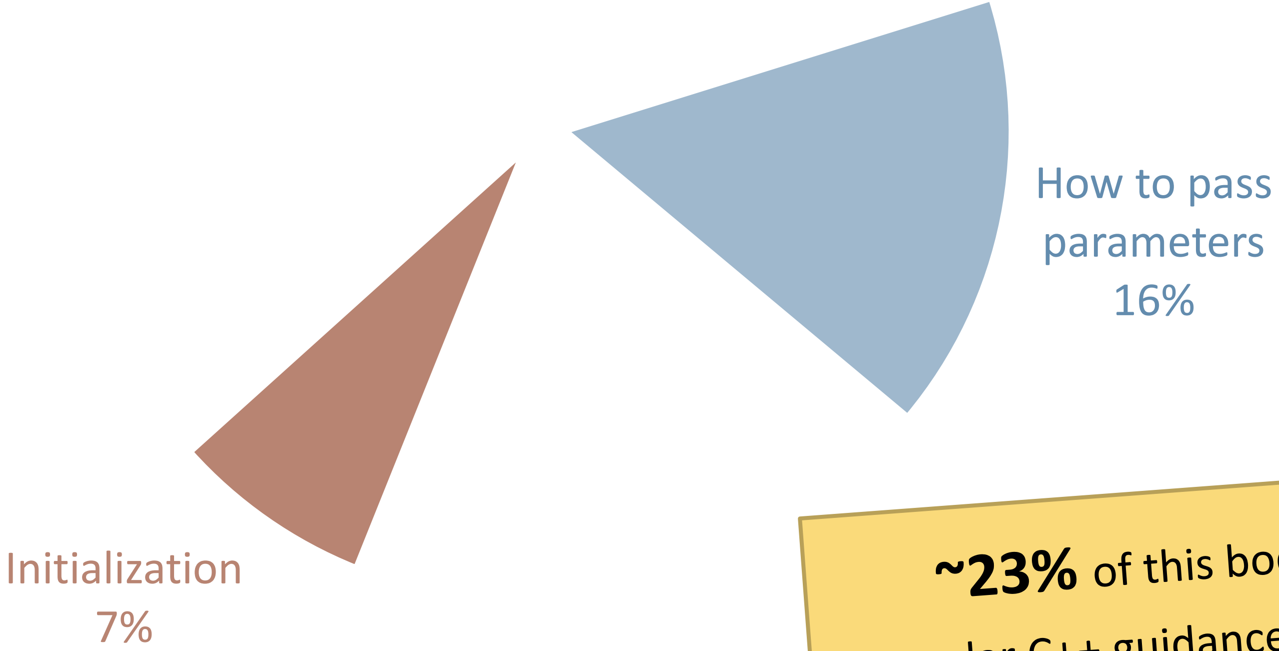
Common claim:
"C++ is too complex"

This talk's contribution:
Empirically catalog, classify, and count

Common despair:
"We can't make things substantially better"

This talk's contribution:
A possible 30% reduction
... 1/3 of the way to 10×


63, Other
103, Multiple ways to say the same thing
81, How to pass parameters
64, "Kind of class" authoring conventions
38, Type safety
37, Initialization
34, Fighting language defaults
26, Lifetime safety
19, Memory management
17, Header files
15, Inheritance
13, Error handling

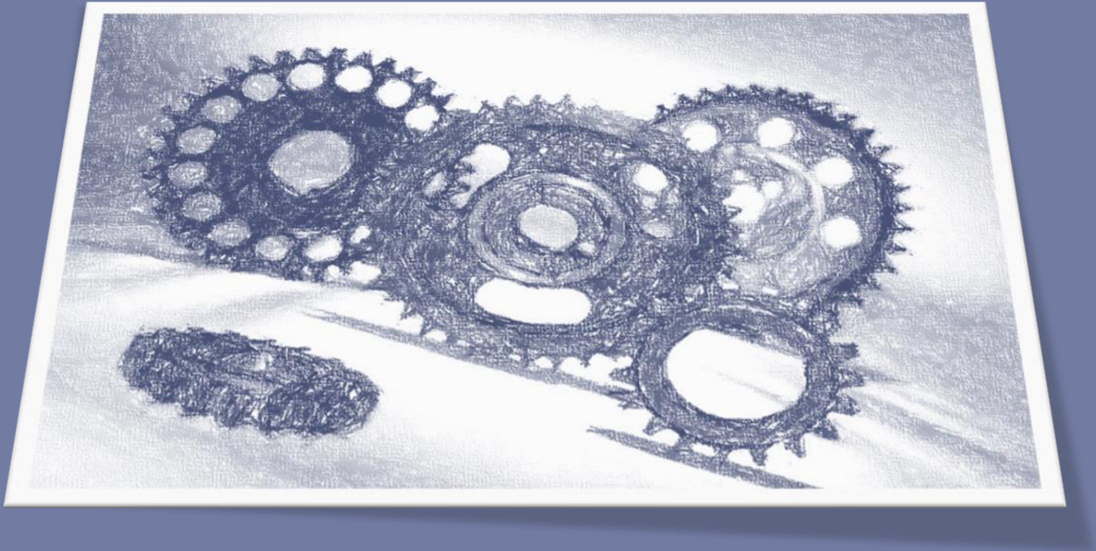


~23% of this body of popular C++ guidance is about how to **pass parameters** and **initialize objects**

Today we teach: “**How**” to pass by value/&/&&

	What we teach today: “How” mechanics
In	Pass by value for “cheap to copy/move” types (incl. builtin types) Otherwise, pass by **const X&** + **Overload** non-templated rvalue reference **X&&** + **std::move** once to optimize rvalues **except** if X must be a type parameter, write templated forwarding reference **X&&** + **enable_if/requires is_lvalue_reference_v<X>** and **std::forward** instead **except** consider passing **X by value** if it’s an “**in+copy**” parameter to a constructor
In-out	Pass by non-const **X&**
Out	Pass by non-const **X&** + nonstd annotations Can’t distinguish from in-out in the language Can’t enforce write-before-read or must-write
Move	Pass by non-templated rvalue reference **X&&** + **std::move** once **except** if X must be a type parameter, write templated forwarding reference **X&&** + **enable_if/requires !is_lvalue_reference_v<X>** and **std::forward** instead
Forward	Pass by templated forwarding reference **T&&** + **std::forward** once **and** if we want only a concrete type X, add **enable_if/requires is_convertible_v<T,X>**

Aim to enable “what,” not “how”



Upgrade: Declare “**what**” instead

Declare intent directly:

```
f (      in X x )            // an X I can read from
f (   inout X x )            // an X I can read and write
f (     out X x )            // an X I will assign to
f (    move X x )            // an X I will move from
f ( forward X x )            // an X I will pass along
```

That’s it… all I’d like to teach about passing parameters in C++.

Most of the following slides are for people who already had to learn **today’s complex thing**, to explain how it maps to the simpler thing.

“Definite first/last use” (see also P1179, Ada, C#)

```
void sample(... x, ... y) {

    process(x);

    if (something(x)) {
        process(y);
        x.hold();
    } else {
        cout << x;
    }

    transfer(y);

}
```

“Definite first/last use” (see also P1179, Ada, C#)

```
void sample(... x, ... y) {
    process(x);                 // definite first use of x
    if (something(x)) {
        process(y);
        x.hold();
    } else {
        cout << x;
    }
    transfer(y);
}
```

“Definite first/last use” (see also P1179, Ada, C#)

```
void sample(... x, ... y) {
    process(x);              // definite first use of x
    if (something(x)) {
        process(y);
        x.hold();            // definite last use of x
    } else {
        cout << x;           // definite last use of x
    }
    transfer(y);
}
```

“Definite first/last use” (see also P1179, Ada, C#)

```
void sample(... x, ... y) {
    process(x);                 // definite first use of x
    if (something(x)) {
        process(y);
        x.hold();               // definite last use of x
    } else {
        cout << x;              // definite last use of x
    }
    transfer(y);                // definite last use of y
}
```

	in X x
Calling convention	X if cheap to copy, else X*
Caller arguments	Initialized object (l- or rvalue)
Callee uses	x is treated as a const lvalue Except each definite last use preserves the arg’s l/rvalue-ness (incl. can move from rvalue arg)

50kft overview: “in”

C++20

```
void f1(int x) {
    g(x);
}
```

```
void f2(const X& x) { // for lvalues
    g(x);
}
void f2(X&& x) {      // for rvalues
    g(std::move(x));
}   // remember to move only once
```

```
template<typename T>
void f3(const T& t) {
    g(t);
}
// hard to overload to pass by value
// hard to overload for rvalues
```

Proposed equivale

```
void f1(in int x) {
    g(x);
}
```

```
void f2(in X x) {
    g(x);
}
```

```
template<typename T>
void f3(in T t) {
    g(t);
}
```

efficient: copies builtins and moves from rvalues (even if f2 is a template)

simple and safe: can’t modify param, implicitly move for last copy if rvalue

simple and clear: no need to overload to optimize values, call std::move, or remember to pass builtins by value

	in X x	inout X x
Calling convention	X if cheap to copy, else X*	X*
Caller arguments	Initialized object (l- or rvalue)	Initialized non-const lvalue
Callee uses	x is treated as a const lvalue Except each definite last use preserves the arg's l/rvalue-ness (incl. can move from rvalue arg)	x is treated as a non-const lvalue If function is not virtual, some path must have a non-const use of x (else use in)

50kft overview: “inout”

C++20

```
void f1(/*inout*/ X& x) {
    g(x); // ok
    ++x;  // ok modifies but can omit
}
```

```
void f2(/*inout*/ X& x) {
    y = x * 2;  // ok
} // not flagged: did not write to x
```

```
// can’t distinguish inout vs out
```

Proposed equivalent

```
void f1(inout X x) {
    g(x); // ok
    ++x;  // ok modifies and required
}
```

```
void f2(inout X x) {
    y = x * 2;
} // error, did not write to x
```

simple and safe: read-before-write from x is okay, but failure to write to x is not okay

simple and clear: can distinguish between inout and out

	in X x	inout X x	out X x
Calling convention	X if cheap to copy, else X*	X*	X*
Caller arguments	Initialized object (l- or rvalue)	Initialized non-const lvalue	Any non-const lvalue
Callee uses	x is treated as a const lvalue Except each definite last use preserves the arg's l/rvalue-ness (incl. can move from rvalue arg)	x is treated as a non-const lvalue If function is not virtual, some path must have a non-const use of x (else use in)	Every path must have a definite first use, that either assigns to x or passes x to another out param

50kft overview: “out”

C++20

```
void f1(/*out*/ X& x) {
    g(x);       // not flagged: read
    x = 42;     // ok but can omit
    g(x);       // ok
}
```

```
void f2(/*out*/ X& x) {
    /* ... no write to x ... */
} // not flagged: did not write to x
```

```
// can’t distinguish inout vs out
```

Proposed equivalent

```
void f1(out X x) {
    g(x);       // error
    x = 42;     // ok, required
    g(x);       // ok
}
```

```
void f2(out X x) {
    /* ... no write to x ... */
} // error, did not write to x
```

simple and safe: error to read-before-write or fail to write; use-after-write is ok

simple and clear: can distinguish between inout and out; out *is* value return where the caller allocates the storage

	in X x	inout X x	out X x	move X x
Calling convention	X if cheap to copy, else X*	X*	X*	X*
Caller arguments	Initialized object (l- or rvalue)	Initialized non-const lvalue	Any non-const lvalue	Initialized non-const rvalue
Callee uses	x is treated as a const lvalue Except each definite last use preserves the arg's l/rvalue-ness (incl. can move from rvalue arg)	x is treated as a non-const lvalue If function is not virtual, some path must have a non-const use of x (else use in)	Every path must have a definite first use, that either assigns to x or passes x to another out param	x is treated as a non-const lvalue Except each definite last use of x treats it as an rvalue and must be to a move parameter

50kft overview: “move”

C++20

```
void f1(X&& x) {
    g(std::move(x));
}
```

```
template<typename T>
    requires
      (!std::is_lvalue_reference_v<T>)
void f2(T&& t) {   // not an rref...
    container.emplace_back
        (std::forward<T>(t));
} // ... so “forward” instead of move
```

moving generic types is cumbersome

Proposed equivalent

```
void f1(move X x) {
    g(x);
}
```

```
template<typename T>
void f2(move T t) {
    container.emplace_back(t);
}
```

simple and clear: allows consuming a parameter even in a template

	in X x	inout X x	out X x	move X x	forward X x
Calling convention	X if cheap to copy, else X*	X*	X*	X*	X*
Caller arguments	Initialized object (l- or rvalue)	Initialized non-const lvalue	Any non-const lvalue	Initialized non-const rvalue	Any object (l- or rvalue)
Callee uses	x is treated as a const lvalue Except each definite last use preserves the arg’s l/rvalue-ness (incl. can move from rvalue arg)	x is treated as a non-const lvalue If function is not virtual, some path must have a non-const use of x (else use in)	Every path must have a definite first use, that either assigns to x or passes x to another out param	x is treated as a non-const lvalue Except each definite last use of x treats it as an rvalue and must be to a move parameter	x is treated as a const lvalue Except each definite last use preserves the arg’s const-ness and l/r-valueness

50kft overview: “forward”

C++20

```
template<typename T>
void f1(T&& t) {
    container.emplace_back
        (std::forward<T>(t));
}
```

```
template<typename T> // must be template
  requires is_convertible_v<T, X>
   // or: is_same_v<remove_cvref_t<T>,X>
void f2(T&& x) {
    g(std::forward<T>(x));
}
```

forwarding concrete types is difficult

Proposed equivalent

```
template<typename T>
void f1(forward T t) {
    container.emplace_back(t);
}
```

```
void f2(forward X x) {
    g(x);
}
```

simple and clear: allows forwarding a parameter without a template or std::forward

supports generic and concrete types: allows forwarding generic and concrete types

Demos

Clang-based prototype available at

***cppx**.godbolt.org*

Prototype implemented by **Andrew Sutton** (Lock3 Software)

and hosted with thanks by **Matt Godbolt** (Aquatic)

Demo’s little helpers

```
//  copy_from: take any number of arguments by value/copy
void copy_from(auto...) { }

//  run_history: Run some code and return the history it generated
std::string history;
auto run_history(auto f) {  history = {};  f();  return history;  }

//  noisy<T>: A little helper to conveniently instrument T's SMF history
template<typename T> struct noisy {
    T t;
    noisy()                                  { history += "default-ctor "; }
    ~noisy()                                 { history += "dtor "; }
    noisy(const noisy& rhs) : t{rhs.t}       { history += "copy-ctor "; }
    noisy(noisy&& rhs) : t{std::move(rhs.t)} { history += "move-ctor "; }
    auto operator=(const noisy& rhs)         { history += "copy-assign ";
                                               t = rhs.t; return *this; }
    auto operator=(noisy&& rhs)              { history += "move-assign ";
                                               t = std::move(rhs.t); return *this; }
};
```

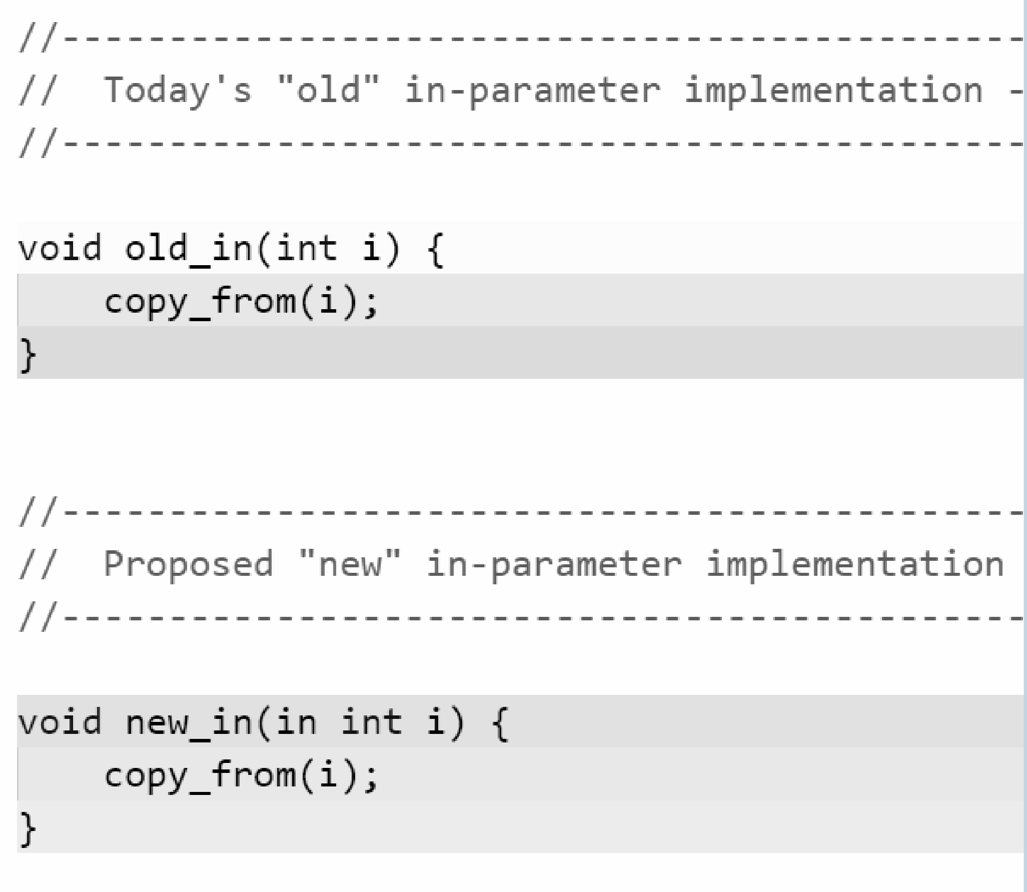
demo-in-1

***Simple** guidance,*

***non**-template,*

***one** parameter*

cppx.godbolt.org/z/ xEx15c

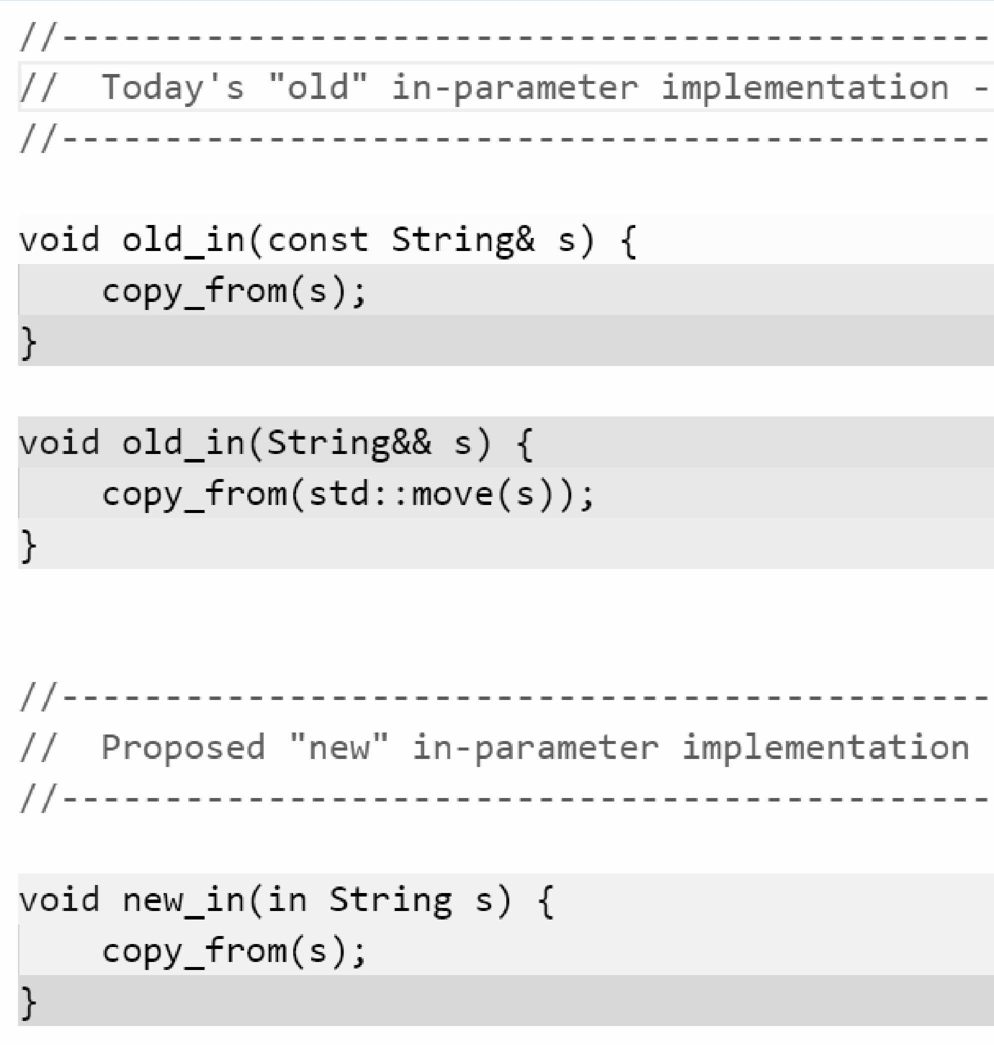


```
//----------------------------------------------
//  Today's "old" in-parameter implementation -
//----------------------------------------------

void old_in(int i) {
    copy_from(i);
}



//----------------------------------------------
//  Proposed "new" in-parameter implementation
//----------------------------------------------

void new_in(in int i) {
    copy_from(i);
}
```

demo-in-2

***Simple** guidance,*
***non**-template,*
***one** parameter*

cppx.godbolt.org/z/fGTbc6



```
//-------------------------------------
//  Today's "old" in-parameter implementation -
//-------------------------------------

void old_in(const String& s) {
    copy_from(s);
}

void old_in(String&& s) {
    copy_from(std::move(s));
}


//-------------------------------------
//  Proposed "new" in-parameter implementation
//-------------------------------------

void new_in(in String s) {
    copy_from(s);
}
```

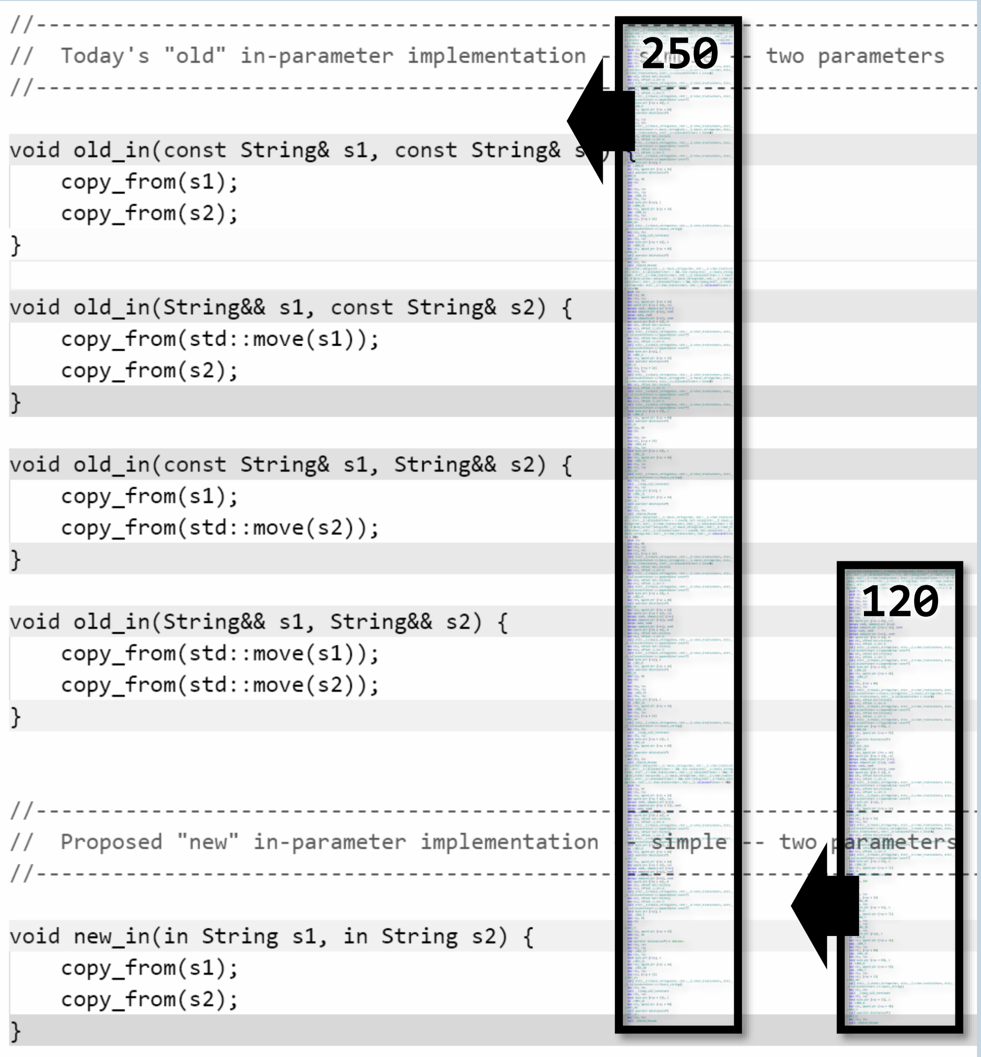
demo-in-3

***Simple** guidance,*

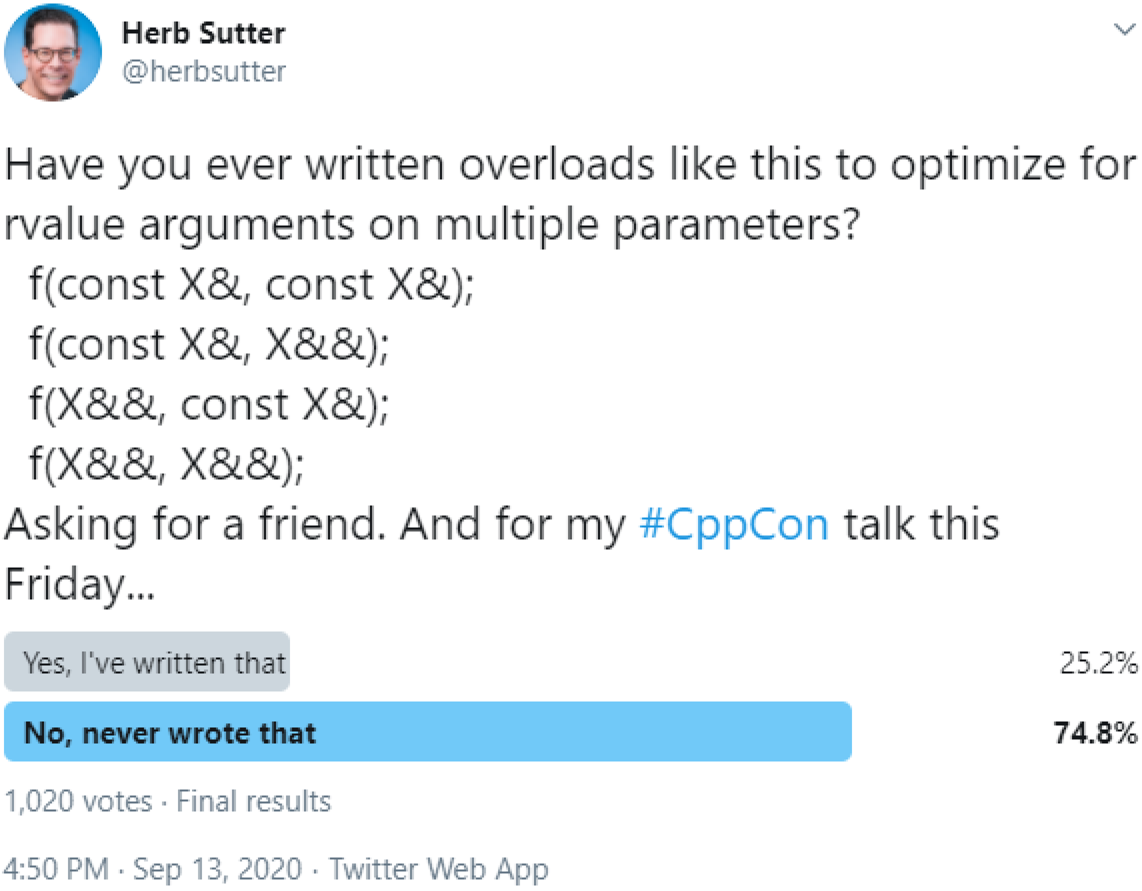
***non**-template,*

***two** parameters*

cppx.godbolt.org/z/ne1dv1



```
//--------------------------------------------------------------------------------
//  Today's "old" in-parameter implementation -- simple -- two parameters
//--------------------------------------------------------------------------------

void old_in(const String& s1, const String& s2) {
    copy_from(s1);
    copy_from(s2);
}

void old_in(String&& s1, const String& s2) {
    copy_from(std::move(s1));
    copy_from(s2);
}

void old_in(const String& s1, String&& s2) {
    copy_from(s1);
    copy_from(std::move(s2));
}

void old_in(String&& s1, String&& s2) {
    copy_from(std::move(s1));
    copy_from(std::move(s2));
}


//--------------------------------------------------------------------------------
//  Proposed "new" in-parameter implementation -- simple -- two parameters
//--------------------------------------------------------------------------------

void new_in(in String s1, in String s2) {
    copy_from(s1);
    copy_from(s2);
}
```



Herb Sutter
@herbsutter

Have you ever written overloads like this to optimize for rvalue arguments on multiple parameters?

```
  f(const X&, const X&);
  f(const X&, X&&);
  f(X&&, const X&);
  f(X&&, X&&);
```

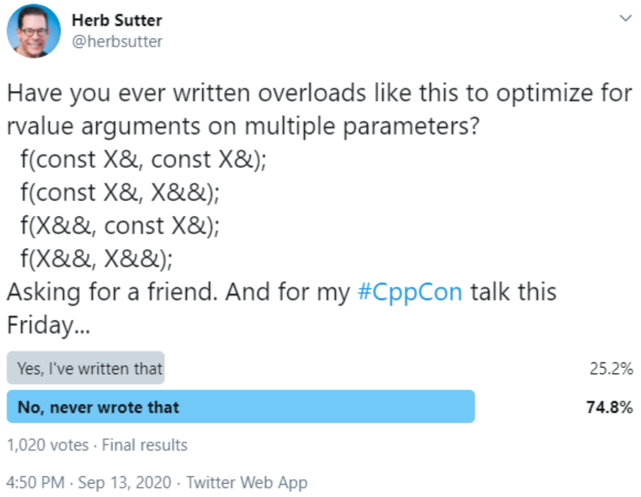
Asking for a friend. And for my #CppCon talk this Friday...

Yes, I've written that 25.2%

No, never wrote that 74.8%

1,020 votes · Final results

4:50 PM · Sep 13, 2020 · Twitter Web App



Herb Sutter @herbsutter

Have you ever written overloads like this to optimize for rvalue arguments on multiple parameters?

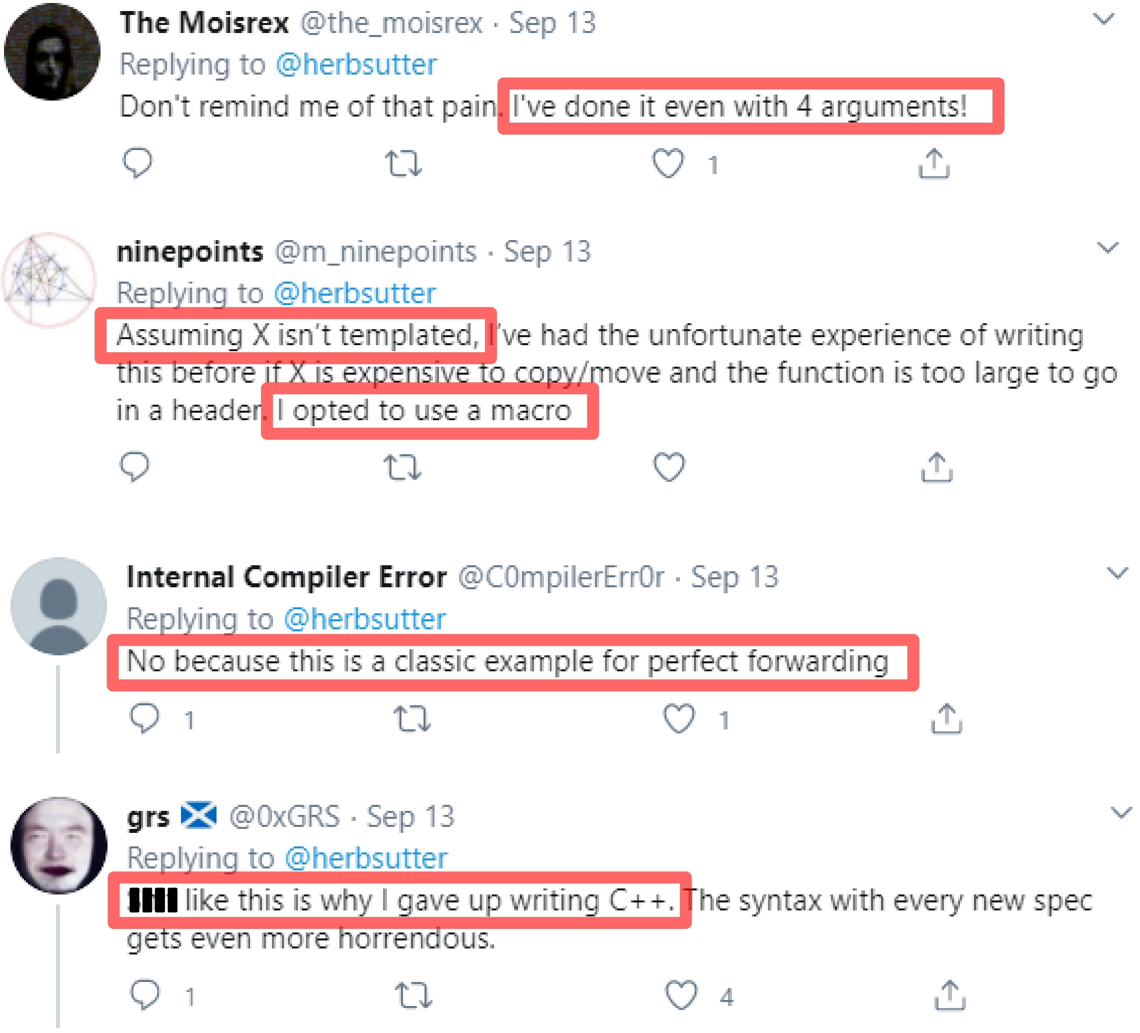
```
  f(const X&, const X&);
  f(const X&, X&&);
  f(X&&, const X&);
  f(X&&, X&&);
```

Asking for a friend. And for my #CppCon talk this Friday...

Yes, I've written that	25.2%
No, never wrote that	**74.8%**

1,020 votes · Final results

4:50 PM · Sep 13, 2020 · Twitter Web App



The Moisrex @the_moisrex · Sep 13

Replying to @herbsutter

Don't remind me of that pain. I've done it even with 4 arguments!

ninepoints @m_ninepoints · Sep 13

Replying to @herbsutter

Assuming X isn't templated, I've had the unfortunate experience of writing this before if X is expensive to copy/move and the function is too large to go in a header. I opted to use a macro

Internal Compiler Error @C0mpilerErr0r · Sep 13

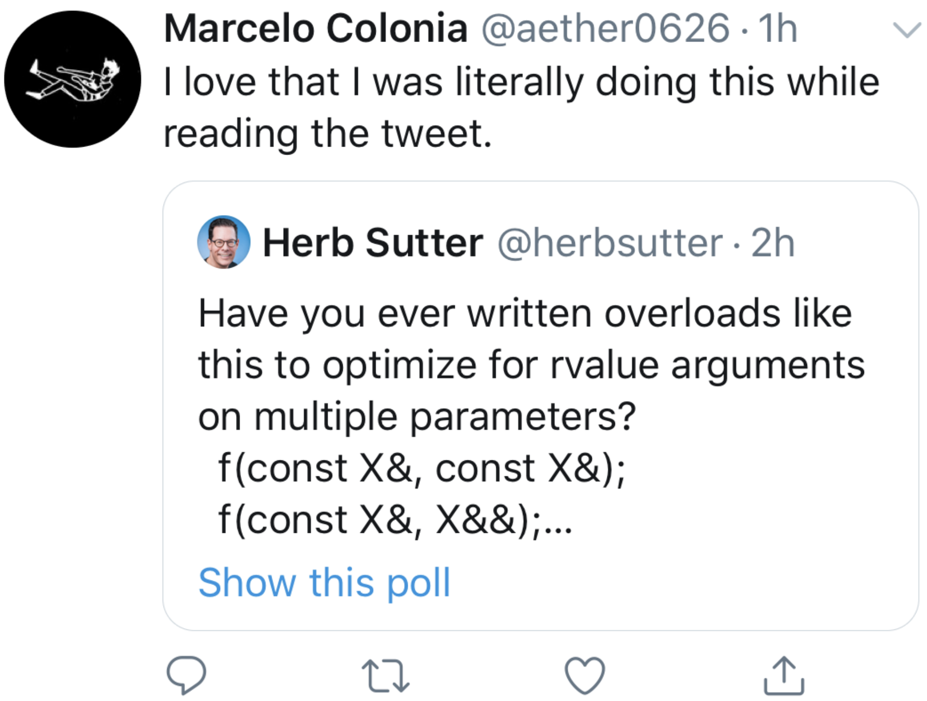
Replying to @herbsutter

No because this is a classic example for perfect forwarding

grs 🏴󠁧󠁢󠁳󠁣󠁴󠁿 @0xGRS · Sep 13

Replying to @herbsutter

[redacted] like this is why I gave up writing C++. The syntax with every new spec gets even more horrendous.



Marcelo Colonia @aether0626 · 1h

I love that I was literally doing this while reading the tweet.

> **Herb Sutter** @herbsutter · 2h
>
> Have you ever written overloads like this to optimize for rvalue arguments on multiple parameters?
>
> ```
>   f(const X&, const X&);
>   f(const X&, X&&);...
> ```
>
> Show this poll

“Not gonna” demo… (but: /z/WYWWcf)

```
//------------------------------------------------------------------------------
//  Today’s "old" in-parameter implementation -- simple -- three parameters
//------------------------------------------------------------------------------


    // ...
    // Ctrl-C/Ctrl-V and tweak (8 combinations)
    // ...


//------------------------------------------------------------------------------
//  Proposed "new" in-parameter implementation -- simple -- three parameters
//------------------------------------------------------------------------------

void new_in(in String s1, in String s2, in String s3) {
    copy_from(s1, s2, s3);
}
```

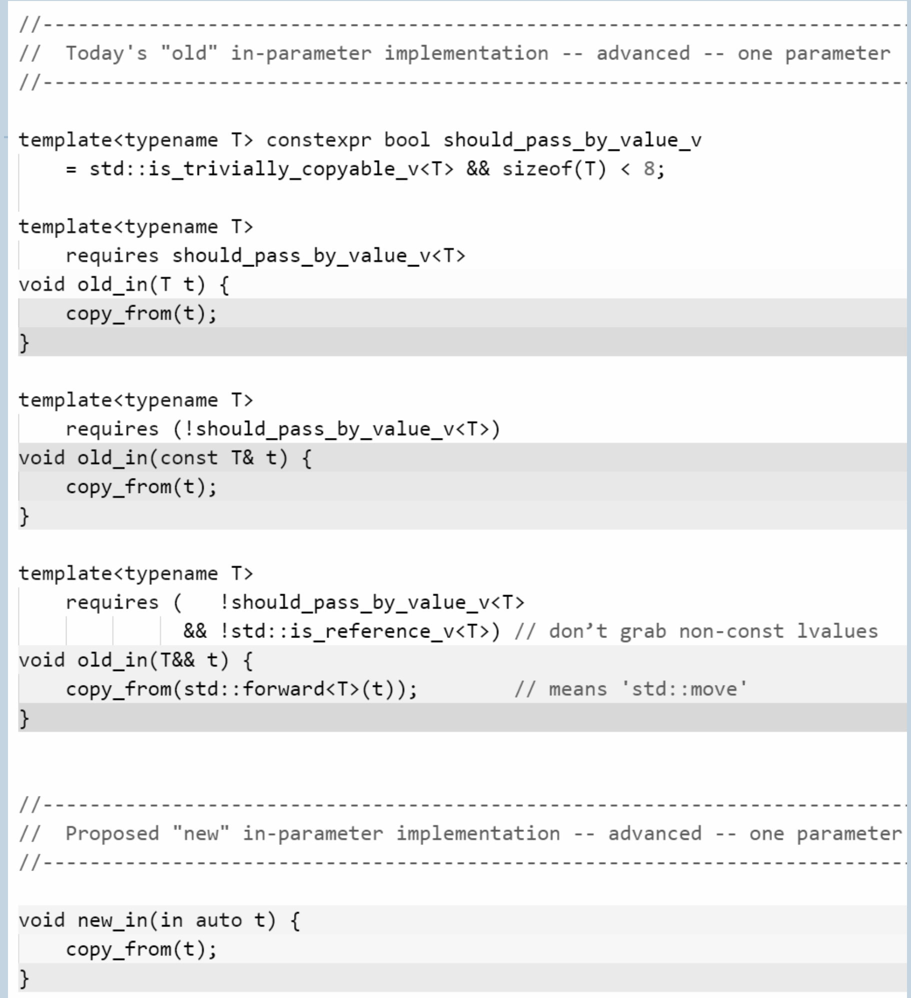
demo-in-4

***Advanced** guidance,*

***template**,*

***one** parameter*

cppx.godbolt.org/z/
498MaK



```
//---------------------------------------------------------------------------
//  Today's "old" in-parameter implementation -- advanced -- one parameter
//---------------------------------------------------------------------------

template<typename T> constexpr bool should_pass_by_value_v
    = std::is_trivially_copyable_v<T> && sizeof(T) < 8;

template<typename T>
    requires should_pass_by_value_v<T>
void old_in(T t) {
    copy_from(t);
}

template<typename T>
    requires (!should_pass_by_value_v<T>)
void old_in(const T& t) {
    copy_from(t);
}

template<typename T>
    requires (   !should_pass_by_value_v<T>
              && !std::is_reference_v<T>) // don’t grab non-const lvalues
void old_in(T&& t) {
    copy_from(std::forward<T>(t));        // means 'std::move'
}


//---------------------------------------------------------------------------
//  Proposed "new" in-parameter implementation -- advanced -- one parameter
//---------------------------------------------------------------------------

void new_in(in auto t) {
    copy_from(t);
}
```

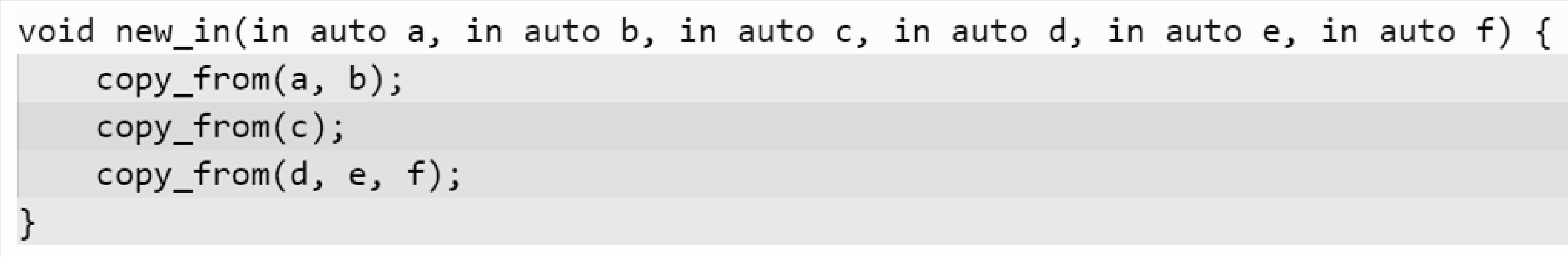
“Not gonna” demo...

```
//------------------------------------------------------------------------------
//  Todays "old" in-parameter implementation -- advanced -- three parameters
//------------------------------------------------------------------------------


    // ...
    // choose your own adventure (24 constrained overloads)
    // ...


//------------------------------------------------------------------------------
//  Proposed "new" in-parameter implementation -- advanced -- three parameters
//------------------------------------------------------------------------------

void new_in(in auto x, in auto y, in auto z) {
    copy_from(x, y, z);
}
```

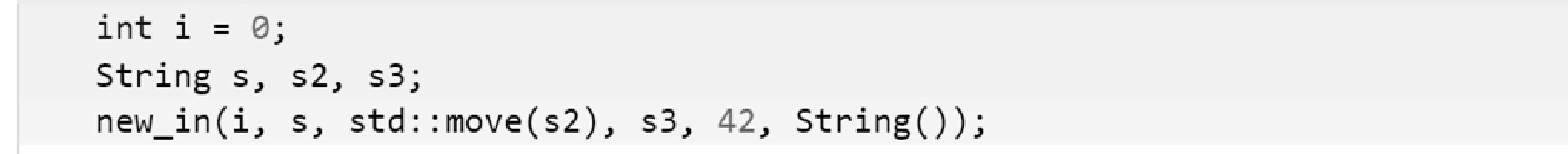
demo-in-5

***Advanced** guidance, **template, N** parameters*

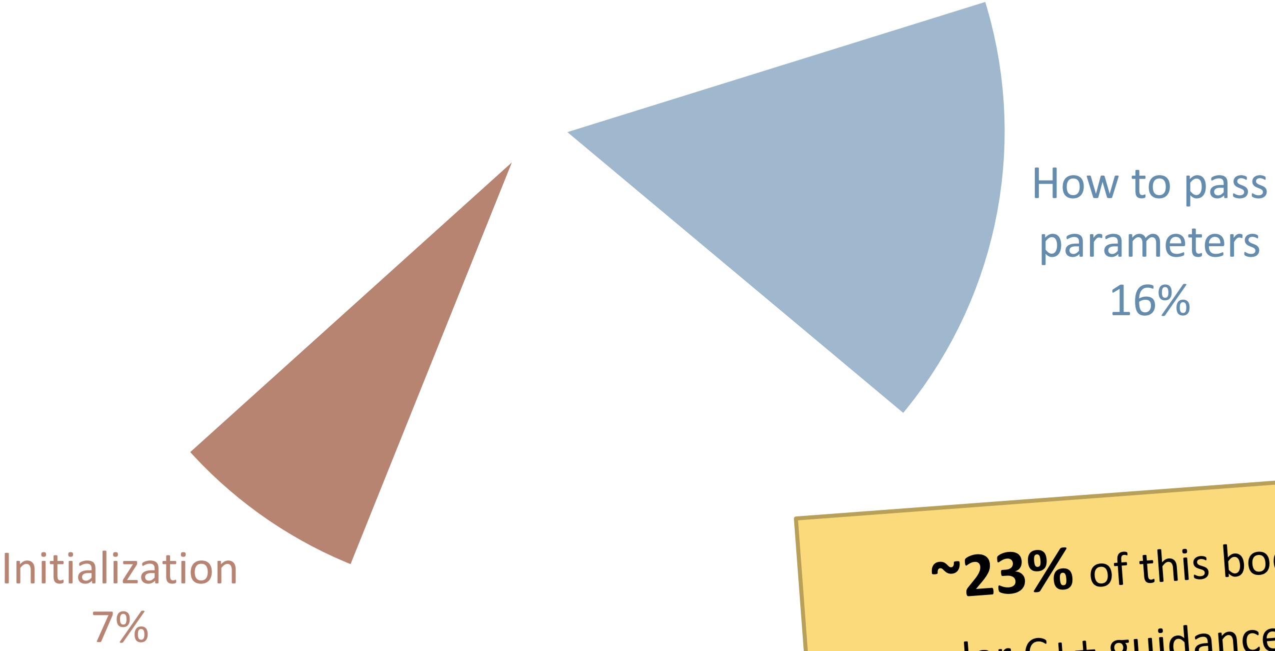
cppx.godbolt.org/z/oxT6aq



```
void new_in(in auto a, in auto b, in auto c, in auto d, in auto e, in auto f) {
    copy_from(a, b);
    copy_from(c);
    copy_from(d, e, f);
}
```



```
    int i = 0;
    String s, s2, s3;
    new_in(i, s, std::move(s2), s3, 42, String());
```



~23% of this body of popular C++ guidance is about how to **pass parameters** and **initialize objects**

Common claim:
“C++ is too complex”

This talk's contribution:
Empirically catalog,
classify, and count

Common despair:
“We can’t make things
substantially better”

This talk's contribution:
A possible 30% reduction
... 1/3 of the way to 10×

Resources and teasers

- Where to read more: *github.com/hsutter/**708***
  - Current draft of *d*0708, examples, test cases
- Where to try an in-progress implementation: ***cppx**.godbolt.org*
  - Please file any issues at the repo above

- Teasers (answers in the paper):
  - What would **out this** mean?
  - What would X::operator= taking **in X** mean?
  - What would writing *both* mean?

```
class X {
  // ...
public:
  X& operator=(in X that) out;
};
```

Simplification: 1..7 of N

	1179 (2015-) Lifetime	0515 (2017-) <=> Comparison	0707 (2017-) Metaclasses	0709 (2018-) Static EH	0708 (2020-) Parameters
Simplification	Directly support "owners" and "pointers," eliminate classes of use-after-free/invalid	Directly express comparison intent, eliminate boilerplate & errors	Directly express class authoring intent, eliminate boilerplate & errors	Eliminate largest fracture in C++ usage/libs	Directly express param intent, eliminate boilerplate, guaranteed unified init
Prototype	● MSVC, Clang		● Clang		○ Clang
			cppx.*godbolt.org*		
Product/spec adoption	● Guidelines ● MSVC ○ Clang	● C++20 (incl. std:: lib)			
WG21 encouraged	n/a	●	○	○	
Next steps	Continue Clang upstreaming (& WG21?)		C++2x reflection & consteval programming	Prototype	Finish prototype WG21 (when face-to-face)

“Efficient abstraction” – in that order!

“Efficient abstraction” – in that order!

Don’t design an abstraction, *then* try to make it efficient

Examples: Smalltalk classes, C++0x concepts

Do learn from “what we already do.” For important abstractions,

“efficient” way we’ve already learned to implement them (but by hand)

then “abstraction” to let us directly express intent (and automate it!)

Examples: vtables (since C!), metaclasses, by-value EH, parameters