Can C++ be 10× simpler & safer … ?

Herb Sutter

*"Inside C++, there is a **much smaller and cleaner language** struggling to get out."*

— B. Stroustrup (D&E, 1994)

*"Say **10% of the size of C++** in definition and similar in front-end compiler size. …*
***Most of the simplification would come from generalization**."*

— B. Stroustrup (ACM HOPL-III, 2007)

we ❤️ 

C++ has lots of challenges

The industry is doing lots of ***major C++ evolution*** experiments — this is one of those

Let's look for ways to push the boundaries to ***bring C++ itself forward and double down on C++*** — not to switch to something else

Let's aim for major C++ ***evolution directed toward things that will make us better C++ programmers*** — not programmers of something else

green-field language

invent new idioms/styles

new modules

new ecosystem/packagers

compatibility bridges

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

Roadmap

- Motivation & approach
- History (since 2015)
- Safety
  - Type safety — CppCon 2021
  - Bounds safety
  - Lifetime safety — CppCon 2015
  - Initialization safety — CppCon 2020
- Simplicity examples
  - Parameter passing — CppCon 2020

Metrics to aim for

“50× safer” means
98% fewer CVEs & bugs in these categories

“10× simpler” means
90% less total guidance to teach in C++ books and courses

So what *is* C++?

Zero-overhead abstraction

Determinism & control

Friction-free interop with C and C++*prev*

=

We've been making progress on all these … but *incremental* (10%), not *game-changing* (10×)

Major reason: 100% syntax backward compatibility

Specific syntax	Unsafe code	Security exploits
Tedium	Vexing parsing	Obsolete features
Lack of good defaults	Difficulty writing tools	1,000-page lists of guidelines
Sharp edges	(General: Not having nice things)	

What if we could have our compatibility cake and eat it too?

Approach: Apply the zero-overhead principle to backward source compatibility… pay only if you use it

the cake is not a lie

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?

Roadmap

- Motivation & approach
- **History (since 2015)**
- Safety
  - Type safety — CppCon 2021
  - Bounds safety
  - Lifetime safety — CppCon 2015
  - Initialization safety — CppCon 2020
- Simplicity examples
  - Parameter passing — CppCon 2020

Metrics to aim for

“50× safer” means
98% fewer CVEs & bugs in these categories

“10× simpler” means
90% less total guidance to teach in C++ books and courses

Last 7 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

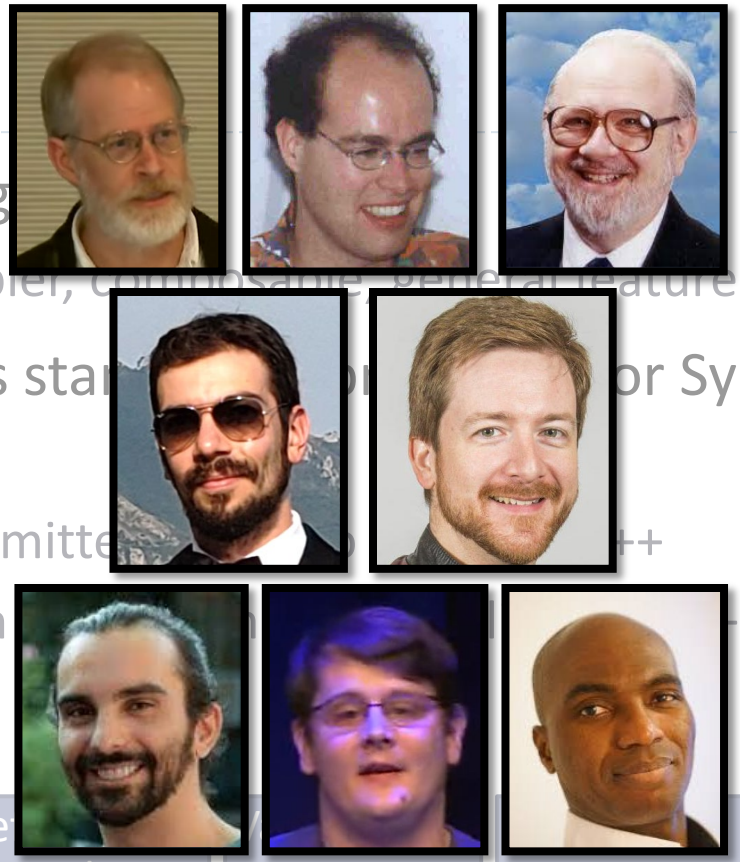
Last 7 years

2015-16: Basic language desig

- "Refactor C++" into fewer, simpler, composable, general features

2016 - : Try individual parts as sta or Syntax 1

- Flesh each out in more detail
- Validate it's a problem the committe ++
- Validate it's a solution direction



Lifetime	gc_arena	<=>	Re 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

Problem: Dependent on prototyping in production C++ compilers

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

What would Bjarne do?

What would Bjarne do?

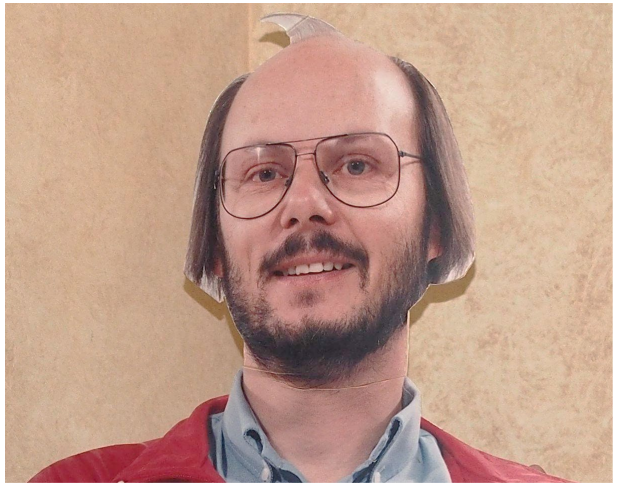
What did Bjarne do?

"C with Classes" goals

1) Value
Address key issues of C:
lack of abstraction

2) Availability
"Usable anywhere C is," incl. environment:
optimizers, linkers, debuggers, tools, …

3) Compatibility
Full interop with C, incl. mix C & C++ source



cfront

C++ → C compiler

What could we do?

“C++ syntax 2 experiment” goals

1) Value
Address key issues of today’s C++:
lack of safety, simplicity, toolability

2) Availability
“Usable anywhere C++ is,” incl. environment:
optimizers, linkers, debuggers, tools, …

3) Compatibility
Full interop with Syntax 1 and C, incl. mix source

cppfront

Cpp2 → Cpp1 compiler

Caveats

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

Structure & build & targets

To build cppfront itself: Use any major C++20 compiler

MSVC	`cl cppfront.cpp -std:c++20 -EHsc`
gcc	`g++-10 cppfront.cpp -std=c++20 -o cppfront`
Clang	`clang++-12 cppfront.cpp -std=c++20 -o cppfront`

To build “syntax 2” code: Use any major C++20 compiler with `cppfront/include` in the include path

MSVC	`cppfront your.cpp2`	→	`cl your.cpp -std:c++20`
gcc	`cppfront your.cpp2`	→	`g++-10 your.cpp -std=c++20`
Clang	`cppfront your.cpp2`	→	`clang++-12 your.cpp -std=c++20`

Design principle

Conceptual integrity

refactor: fewer composable general features

be consistent	Don't make similar things different Make important differences visible
be orthogonal	Avoid arbitrary coupling Let features be used freely in combination
be general	Don't restrict what is inherent Don't arbitrarily restrict a complete set of uses Avoid special cases and partial features

Recall:

*"Say **10% of the size of C++ …***

*Most of the simplification would come from **generalization**."*

— B. Stroustrup (ACM HOPL-III, 2007)

Design stakes

Embracing constraints

goals: safety, simplicity, toolability

stay measurable	Each change must address a known C++ weakness in a measurable way (e.g., remove X% of rules we teach, remove X% of reported vulnerabilities)
stay C++	never violate zero-overhead, opt-in to “open the hood”

syntax & grammar

context-free	Esp. parsing never requires sema (e.g., lookup)
order-independent	No forward declarations or ordering gotchas
declare l-to-r	Declarations are written left to right
declare ≡ use	Declaration syntax mirrors use syntax

“Declare l-to-r”: *name : type = value*

```
class shape             { /* syntax 1 code since 1980, can't update semantics
                             without backward compatibility breakage concerns */ };

shape: type             = { /* syntax 2 code doesn't exist today, can update
                             semantics as desired without any breaking change */ }


auto f(int i) -> string { /* syntax 1 code since C++11, can't update semantics
                             without backward compatibility breakage concerns */ }

f: (i: int) -> string = { /* syntax 2 code doesn't exist today, can update
                             semantics as desired without any breaking change */ }
```

```
main: () -> int = {
    vec: std::vector<std::string>
       = ("hello", "2022");
    view: std::span = vec;

    for view do :(inout str:_) = {
        len := decorate(str);
        println(str, len);
    }
}
```

name : type = value
left-to-right declaration

C++23 "import std;" is implicit under -pure-cpp2

```
main: () -> int = {
    vec: std::vector<std::string>
         = ("hello", "2022");
    view: std::span = vec;

    for view do :(inout str: _) = {
        len := decorate(str);
        println(str, len);
    }
}

decorate: (inout thing: _ ) -> int = {
    thing = "[" + thing + "]";
    return thing.ssize();
}

println: (x: _, len: _) =
    std::cout
        << ">> " << x
        << " - length "
        << len << "\n";
```

modules-first, fast build, strong ODR
"skating to where the puck is going"

d0708 param passing CppCon 2020

_ wildcard, implicit template

optional {return … }
for single-expression function

order independence
(no forward declarations)

```
main: () -> int = {
    vec: std::vector<std::string>
       = ("hello", "2022");
    view: std::span = vec;

    for view do :(inout str:_) = {
        len := decorate(str);
        println(str, len);
    }
}

decorate: (inout thing: _ ) -> int = {
    thing = "[" + thing + "]";
    return thing.ssize();
}

println: (x: _, len: _) =
    std::cout
        << ">> " << x
        << " - length "
        << len << "\n";
```

```
#line 1 "demo.cpp2"
[[nodiscard]] auto main() -> int{
    std::vector<std::string> vec {
          "hello", "2022" };
    std::span view { vec };

    for ( auto&& cpp2_range = view;  auto& str : cpp2_range ) {
        auto len { decorate(str) };
        println(str, len);
    }
}

[[nodiscard]] auto decorate(auto& thing) -> int{
    thing = "[" + thing + "]";
    return CPP2_UFCS_0(ssize, thing);
}

auto println(auto const& x, auto const& len) -> void {
    std::cout
        << ">> " << x
        << " - length "
        << len << "\n"; }
```

default [[nodiscard]]

normal CTAD

UFCS

Readable Cpp1… I want to be able to switch back to Cpp1 anytime and keep my code

```
#line 1 "demo.cpp2"
[[nodiscard]] auto main() -> int{
    std::vector<std::string> vec {
                "hello", "2022" };
    std::span view { vec };

    for ( auto&& cpp2_range = view;  auto& str : cpp2_range ) {
        auto len { decorate(str) };
        println(str, len);
    }
}

[[nodiscard]] auto decorate(auto& thing) -> int{
    thing = "[" + thing + "]";
    return CPP2_UFCS_0(ssize, thing);
}

auto println(auto const& x, auto const& len) -> void {
    std::cout
        << ">> " << x
        << " - length "
        << len << "\n"; }
```

self-contained support library header (e.g., `in<T>`)

order independence (no forward declarations in Cpp2 because we forward-declare everything in the Cpp1 code)

```
// ----- Cpp2 support -----
#define CPP2_USE_MODULES         Yes
#include "cpp2util.h"


#line 1 "demo.cpp2"
[[nodiscard]] auto main() -> int;
#line 12 "demo.cpp2"
[[nodiscard]] auto decorate(auto& thing) -> int;
#line 17 "demo.cpp2"
auto println(auto const& x, auto const& len) -> void;
#line 22 "demo.cpp2"



//=== Cpp2 definitions ============================================

#line 1 "demo.cpp2"
[[nodiscard]] auto main() -> int{
    std::vector<std::string> vec {
                "hello", "2022" };
    std::span view { vec };
```

left-to-right unary operators
(e.g., `*` `.` is natural, don't need duplicate `->`)

track positions, incl. comments

```
test: (a:_) -> std::string = {

    return call( a,
        b(c)*++, "hello", /* polite
                          greeting
                          goes here */ " there",
        d::e( f*.g()++, // because f is foobar
              h.i(),
              j(k,l) )
        );

}
```

```
// ----- Cpp2 support -----
#include "cpp2util.h"


#line 1 "demo.cpp2"
[[nodiscard]] auto test(auto const& a) -> std::string;

//=== Cpp2 definitions ================================

#line 1 "demo.cpp2"
[[nodiscard]] auto test(auto const& a) -> std::string{

    return call( a,
        ++*b(c), "hello", /* polite
                          greeting
                          goes here */ " there",
        d::e( ++(*f).g(), // because f is foobar
              CPP2_UFCS_0(i, h),
              j(k, l))
        ) ;

}
```

with `-clean-cpp1`

Readable Cpp1 in my
personal formatting style…
I want to be able to switch back
to Cpp1 anytime and keep my code

```
#include "cpp2util.h"


[[nodiscard]] auto test(auto const& a) -> std::string;
[[nodiscard]] auto test(auto const& a) -> std::string{

    return call( a,
        ++*b(c), "hello", /* polite
                          greeting
                          goes here */ " there",
        d::e( ++(*f).g(), // because f is foobar
              CPP2_UFCS_0(i, h),
              j(k, l))
        ) ;

}
```

“Everyone knows”

Everyone knows that compiling to C++ emits `__uGLy #UnRead@bul` generated code, right?

But the worst examples are compiling a foreign language that’s *unlike* C++ to C++

Ugliness ∝ impedance mismatch

Demo: Overview

demo.cpp2

```
#include <iostream>
#include <string>

name: () -> std::string = {
    s: std::string = "world";
    decorate(s);
    return s;
}

decorate: (inout s: std::string) =
    s = "[" + s + "]";

auto main() -> int {
    std::cout << "Hello " << name() << "\n";
}

```

demo.cpp

```
// ----- Cpp2 support -----
#include "cpp2util.h"

#line 1 "demo.cpp2"
#include <iostream>
#include <string>

[[nodiscard]] auto name() -> std::string;
#line 10 "demo.cpp2"
auto decorate(std::string& s) -> void;
#line 12 "demo.cpp2"

auto main() -> int {
    std::cout << "Hello " << name() << "\n";
}

//=== Cpp2 definitions ==========

#line 3 "demo.cpp2"

[[nodiscard]] auto name() -> std::string{
    std::string s { "world" };
    decorate(s);
    return s;
}

auto decorate(std::string& s) -> void {
    s = "[" + s + "]"; }
```

with -pure-cpp2
and -clean-cpp1

demo.cpp2

```
1
2   main: () -> int = {
3       std::cout << "Hello " << name() << "\n";
4   }
5
6   name: () -> std::string = {
7       s: std::string = "world";
8       decorate(s);
9       return s;
10  }
11
12  decorate: (inout s: std::string) =
13      s = "[" + s + "]";
14
15
```

demo.cpp

```
1   #define CPP2_USE_MODULES          Yes
2   #include "cpp2util.h"
3
4
5   [[nodiscard]] auto main() -> int;
6   [[nodiscard]] auto name() -> std::string;
7   auto decorate(std::string& s) -> void;
8
9
10  [[nodiscard]] auto main() -> int{
11      std::cout << "Hello " << name() << "\n";
12  }
13
14  [[nodiscard]] auto name() -> std::string{
15      std::string s { "world" };
16      decorate(s);
17      return s;
18  }
19
20  auto decorate(std::string& s) -> void {
21      s = "[" + s + "]"; }
```

with -pure-cpp2
and -clean-cpp1

demo.cpp2

```
1
2   main: () -> int = {
3       std::cout << "Hello " << name() << "\n";
4   }
5
6   name: () -> std::string = {
7       s := new<std::string>( "world" );
8       decorate(s*);
9       return s*;
10  }
11
12  decorate: (inout s: std::string) =
13      s = "[" + s + "]";
14
15
```

demo.cpp

```
1   #define CPP2_USE_MODULES          Yes
2   #include "cpp2util.h"
3
4
5   [[nodiscard]] auto main() -> int;
6   [[nodiscard]] auto name() -> std::string;
7   auto decorate(std::string& s) -> void;
8
9
10  [[nodiscard]] auto main() -> int{
11      std::cout << "Hello " << name() << "\n";
12  }
13
14  [[nodiscard]] auto name() -> std::string{
15      auto s { cpp2_new<std::string>("world") };
16      decorate(*s);
17      return *s;
18  }
19
20  auto decorate(std::string& s) -> void {
21      s = "[" + s + "]"; }
```

demo.cpp2

```
1
2   //  "A better C than C" ... ?
3   //
4   main: () -> int = {
5       s: std::string = "Fred";
6       myfile := fopen("xyzzy", "w");
7       myfile.fprintf( "Hello %s!", s.c_str() );
8       myfile.fclose();
9   }
```

with -pure-cpp2
and -clean-cpp1

demo.cpp

```
1   #include "cpp2util.h"
2
3
4   [[nodiscard]] auto main() -> int;
5
6   //  "A better C than C" ... ?
7   //
8   [[nodiscard]] auto main() -> int{
9       std::string s { "Fred" };
10      auto myfile { fopen("xyzzy", "w") };
11      CPP2_UFCS(fprintf, myfile, "Hello %s!", CPP2_UFCS_0(c_str, s));
12      CPP2_UFCS_0(fclose, myfile);
13  }
```

"Don't pay for what you don't use" …
100% source compat, pay only when you use it

Mixed Syntax 1 & 2 in the same source file: Incremental adoption

You can	**Grail A: "Write one line and start seeing benefit"**
You get	Perfect source compatibility (macros, SFINAE, `#include`, …)
You avoid	Python 2/3 problem

Standalone Syntax 2 in a separate source file: C++ 10× simpler and safer

You can	**Grail B: "Write in a 10× simpler and safer C++"**
You get	Safe by construction, seamless interop via module `import`
You avoid	90% of pitfalls, 90% of teaching/learning, slow compilers

Roadmap

Motivation & approach

History (since 2015)

Safety

- Type safety — CppCon 2021
- Bounds safety
- Lifetime safety — CppCon 2015
- Initialization safety — CppCon 2020

Simplicity examples

- Parameter passing — CppCon 2020

Metrics to aim for

“50× safer” means
98% fewer CVEs & bugs in these categories

“10× simpler” means
90% less total guidance to teach in C++ books and courses

MITRE: 2022 Most Dangerous Software Weaknesses

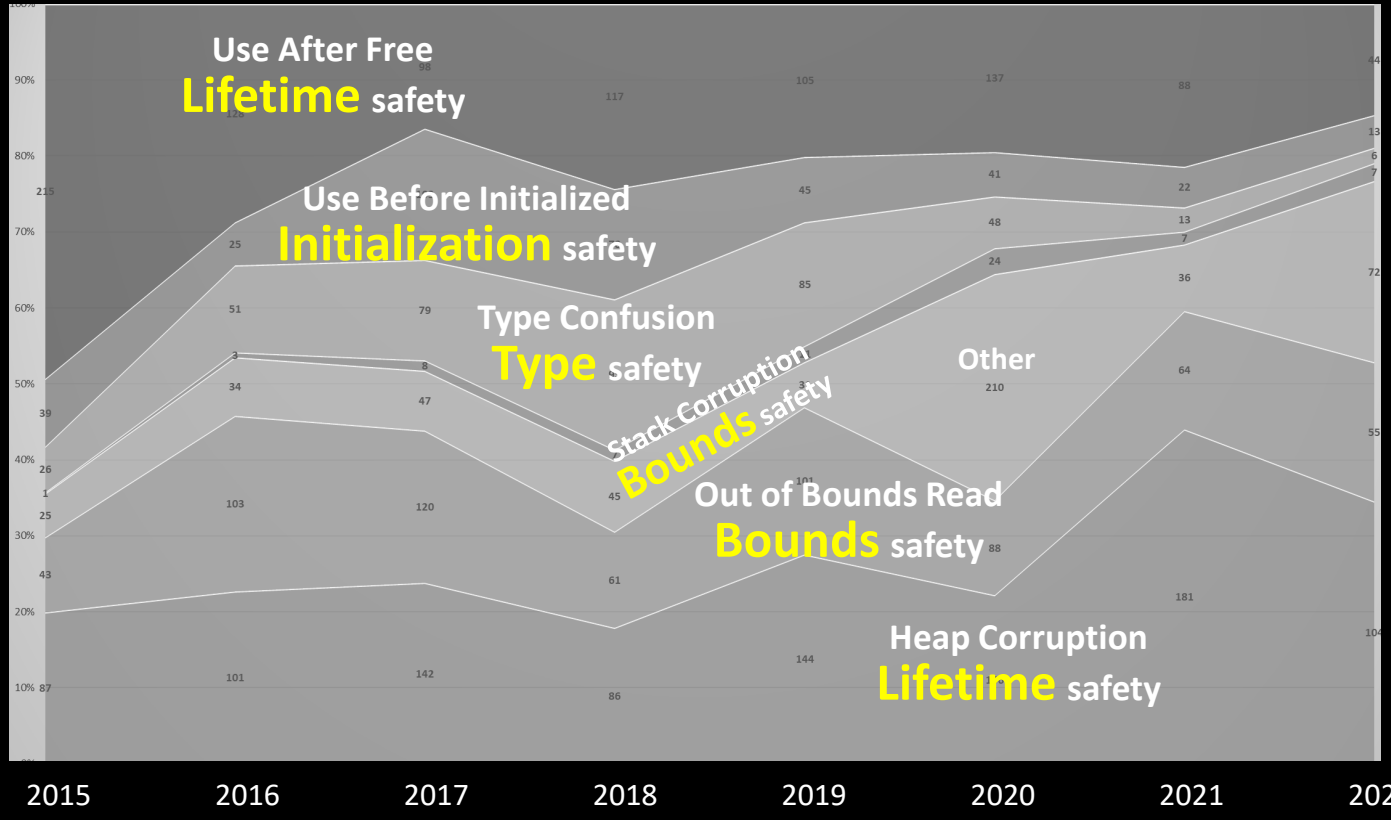
Rank	Name	Score
1	**Out-of-bounds Write**	**64.20**
2	… ('Cross-site Scripting')	45.97
3	… ('SQL Injection')	22.11
4	Improper Input Validation	20.63
5	**Out-of-bounds Read**	**17.67**
6	… ('OS Command Injection')	17.53
7	**Use After Free**	**15.50**
8	… Pathname to a Restricted Directory …	14.08
9	Cross-Site Request Forgery (CSRF)	11.53
10	…Upload of File with Dangerous Type	9.56
11	**NULL Pointer Dereference**	**7.15**
12	Deserialization of untrusted data	6.68
13	**Integer Overflow or Wraparound**	**6.53**

1, 5, 7, 11, 13 relate to PL safety

play the classic hits & party like it's 1999!

https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

Memory Safety CVEs: Root cause by patch year



Safety and the zero-overhead principle

Static enforcement by default: Safety by construction

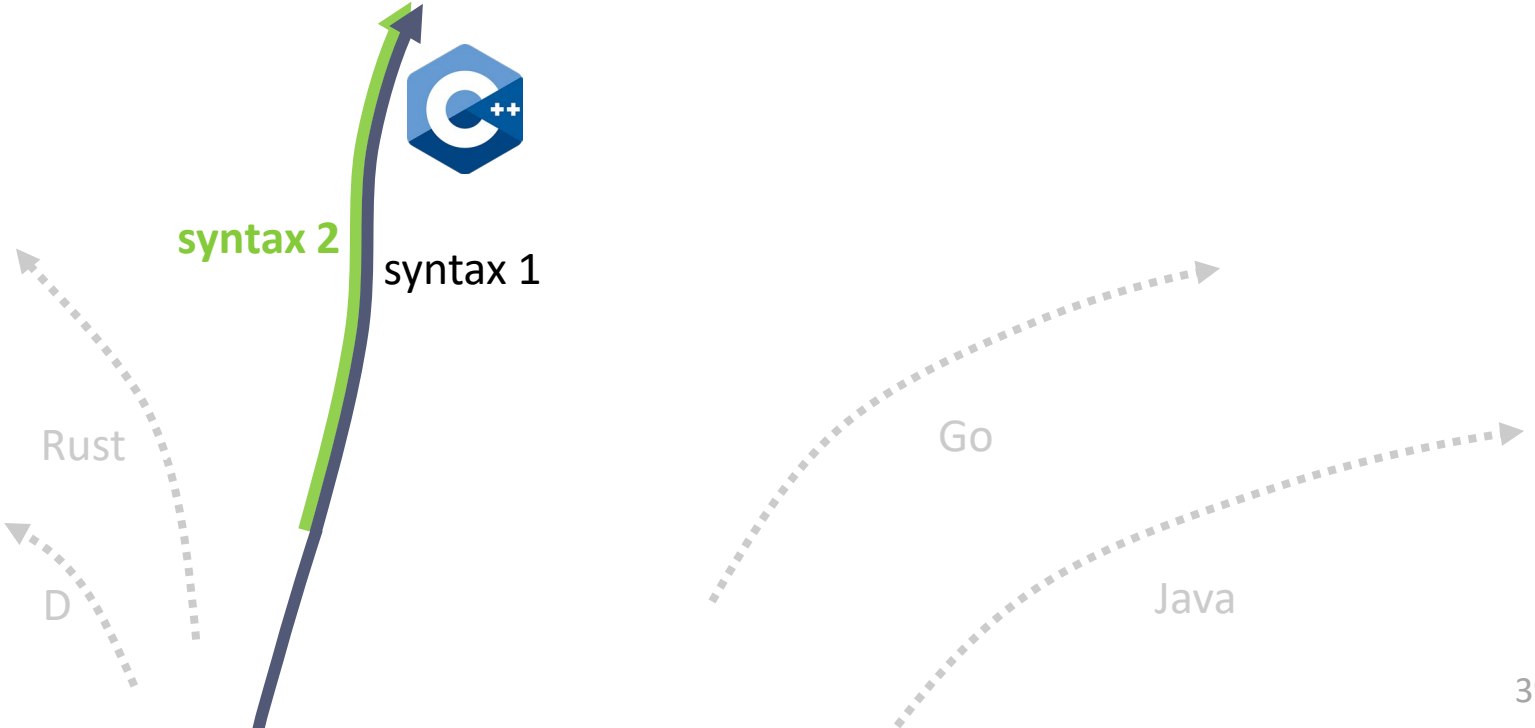
Dynamic enforcement where needed: Visible + pay-for-use

approach: take the best practices **we already teach and promote** and

1. enforce them **by default**
2. direct programmers to what we already say to "**do this instead**"
3. focus any new additions on filling remaining holes

approach: take the best practices **we already teach and promote** and

1. enforce them **by default**
2. direct programmers to what we already say to "**do this instead**"
3. focus any new additions on filling remaining holes



Roadmap

Motivation & approach

History (since 2015)

Safety

- Type safety — CppCon 2021
  - Bounds safety
  - Lifetime safety — CppCon 2015
  - Initialization safety — CppCon 2020

Simplicity examples

- Parameter passing — CppCon 2020

Metrics to aim for

“50× safer” means
98% fewer CVEs & bugs in these categories

“10× simpler” means
90% less total guidance to teach in C++ books and courses

C++ Core Guidelines

as = implemented using as

Pro.safety: Type-safety profile

- Type.1: Avoid casts:

as 1. Don't use `reinterpret_cast`; A strict version of Avoid casts and prefer named casts.

as 2. Don't use `static_cast` for arithmetic types; A strict version of Avoid casts and prefer named casts.

as 3. Don't cast between pointer types where the source type and the target type are the same; A strict version of Avoid casts.

as 4. Don't cast between pointer types when the conversion could be implicit; A strict version of Avoid casts.

as • Type.2: Don't use `static_cast` to downcast: Use `dynamic_cast` instead.

as • Type.3: Don't use `const_cast` to cast away `const` (i.e., at all): Don't cast away const.

as • Type.4: Don't use C-style `(T)expression` or functional `T(expression)` casts: Prefer construction or named casts or `T{expression}`.

✓ • Type.5: Don't use a variable before it has been initialized: always initialize.

TODO classes • Type.6: Always initialize a member variable: always initialize, possibly using default constructors or default member initializers.

✓ • Type.7: Avoid naked union: Use `variant` instead.

✓ • Type.8: Avoid varargs: Don't use `va_arg` arguments.

from **CppCon 2021**

safe

		Queries	P2392	Casts	P2392
static	language			(Y)x	—
				reinterpret_cast<Y>(x)	—
				const_cast<X&>(cx)	—
		is_same_v<X,Y>	X **is** Y	Y(x), Y{x}	x **as** Y
		is_base_of_v<B,D>	D **is** B	static_cast<B*>(pd)	pd **as** B*
dynamic	language	dynamic_cast<D*>(pb)	pb **is** D*	dynamic_cast<D*>(pb)	pb **as** D*
	library	std::holds_alternative<T>(v)	v **is** T	std::get<T>(v) std::get<T&>(v)	v **as** T v **as** T&
		a.type() == typeid(T)	a **is** T	std::any_cast<T>(a) *std::any_cast<T*>(&a)	a **as** T a **as** T&
		o.has_value()	o **is** T	o.value()	o **as** T
		f.wait_for(chrono::seconds(0)) == future_status::ready	f **is** T	f.get()	f **as** T

Demo: Type safety

```
main: () -> int = {
    v: std::variant<int, double> = 42.0;
    a: std::any = "xyzzy" as std::string;
    o: std::optional<int> = ();

    test_generic(3.14);
    test_generic(v);
    test_generic(a);
    test_generic(o);
    std::cout << "\n";

    v = 1;
    a = 2;
    o = 3;
    test_generic(42);
    test_generic(v);
    test_generic(a);
    test_generic(o);
}
```

```
test_generic: ( x: _ ) = {
    std::cout
        << std::setw(30) << typeid(x).name()
        << " value is "
        << inspect x -> std::string {
            is int = std::to_string(x as int);
            is std::string = x as std::string;
            is _   = "not an int or a string";
        }
        << "\n";
}
```

```
C:\demo>demo
                        double value is not an int or a string
class std::variant<int,double> value is not an int or a string
                class std::any value is xyzzy
     class std::optional<int> value is not an int or a string

                           int value is 42
class std::variant<int,double> value is 1
                class std::any value is 2
     class std::optional<int> value is 3
```

Roadmap

- Motivation & approach
- History (since 2015)
- Safety
  - Type safety — CppCon 2021
  - **Bounds safety**
  - Lifetime safety — CppCon 2015
  - Initialization safety — CppCon 2020
- Simplicity examples
  - Parameter passing — CppCon 2020

Metrics to aim for

“50× safer” means
98% fewer CVEs & bugs in these categories

“10× simpler” means
90% less total guidance to teach in C++ books and courses

C++ Core Guidelines

Pro.bounds: Bounds safety profile

- Bounds.1: Don't use pointer arithmetic. Use `span` instead: Pass pointers to single objects (only) and Keep pointer arithmetic simple.

- Bounds.2: Only index into arrays using constant expressions: Pass pointers to single objects (only) and Keep pointer arithmetic simple.

- Bounds.3: No array-to-pointer decay: Pass pointers to single objects (only) and Keep pointer arithmetic simple.

TODO
`std::`

- Bounds.4: Don't use standard-library functions and types that are not bounds-checked: Use the standard library in a type-safe manner.

Pointers point to a single object

Q: Why does this need syntax 2?

A: Can't ban pointer arithmetic today…

Compatibility: It would break the world, including all the C code

Static enforcement

Arithmetic: Reject `++`, `--`, `+`, `-`, et al. on raw pointers

Bitwise operations: Reject `~` et al. on raw pointers

Demo: Bounds safety

```
main: () -> int = {
    words: std::vector<std::string> =
        ( "decorated", "hello", "world" );

    first: *std::string = words.front()&;
    last : *std::string = words.back()&;

    while first <= last {
        print_and_decorate(first*);
        first++;    // unsafe
    }
}
```

```
demo.cpp2...
demo.cpp2(9,14): error: ++ - pointer arithmetic is illegal - use std::span or gsl
::span instead
  ==> program violates bounds safety guarantee - see previous errors
```

```
main: () -> int = {
    words: std::vector<std::string> =
        ( "decorated", "hello", "world" );

    first: *std::string = words.front()&;
    last : *std::string = words.back()&;

    while first <= last {
        delete first;
    }
}
```

```
demo.cpp2...
demo.cpp2(9,13): error: 'delete' and owning raw pointers are not supported in Cpp
2
demo.cpp2(9,13): error:   - use unique.new<T>, shared.new<T>, or gc.new<T> instea
d (in that order)
```

```
main: () -> int = {
    words: std::vector<std::string> =
        ( "decorated", "hello", "world" );

    s: std::span<std::string> = words;

    i := 0;
    while i < s.ssize() next i++ {
        print_and_decorate( s[i] );
    }
}

print_and_decorate: (thing:_) =
    std::cout << ">> " << thing << "\n";
```

```
C:\test\demo>cppfront demo.cpp2 -p
demo.cpp2... ok (all Cpp2, passes safety checks)
```

```
C:\test\demo>demo
>> decorated
>> hello
>> world
```

```
main: () -> int = {
    words: std::vector<std::string> =
        ( "decorated", "hello", "world" );

    print_and_decorate( words[3] );
}

print_and_decorate: (thing:_) =
    std::cout << ">> " << thing << "\n";
```

```
C:\test\demo>cppfront demo.cpp2 -s -a -p
demo.cpp2... ok (all Cpp2, passes safety checks)
```

```
C:\test\demo>demo
demo.cpp2(5) main: Bounds safety violation: out of bounds access attempt detected
```

```
main: () -> int = {
    cpp2::Bounds.set_handler(call_my_framework&);
    words: std::vector<std::string> =
        ( "decorated", "hello", "world" );

    print_and_decorate( words[3] );
}

print_and_decorate: (thing:_) =
    std::cout << ">> " << thing << "\n";

call_my_framework: (msg: * const char) =
    std::cout
        << "sending error to my framework... ["
        << msg << "]\n";
```

```
C:\test\demo>demo
sending error to my framework... [dynamic null dereference attempt detected]
```

Roadmap

- Motivation & approach
- History (since 2015)
- Safety
  - Type safety — CppCon 2021
  - Bounds safety
  - **Lifetime safety** — CppCon 2015
  - Initialization safety — CppCon 2020
- Simplicity examples
  - Parameter passing — CppCon 2020

Metrics to aim for

“50× safer” means
98% fewer CVEs & bugs in these categories

“10× simpler” means
90% less total guidance to teach in C++ books and courses

C++ Core Guidelines

Pro.lifetime: Lifetime safety profile

Accessing through a pointer that doesn't point to anything is a major source of errors, and very hard to avoid in many traditional C or C++ styles of programming. For example, a pointer might be uninitialized, the `nullptr`, point beyond the range of an array, or to a deleted object.

See the current design specification here.

Lifetime safety profile summary:

partial, mostly to-do

- Lifetime.1: Don't dereference a possibly invalid pointer: detect or avoid.

Lifetime

P1179
CppCon 2015/18

Experiment: Pointers should not be null?

Why does this need Syntax 2? Can't make pointers non-null today…

Compatibility	It would break the world
Defaults	Today null is the default value(!)

(*cue Kate Gregory: "what you say when you say nothing at all"*)

Static enforcement

Initialization/assignment	Reject setting a pointer to `nullptr`/`0`/`NULL`/`{}`
Profile.Lifetime	Local static analysis for use-after-free + nulls

Dynamic enforcement

Check for non-null after every Cpp1 code expression used to initialize/assign a Pointer, or that has mutable access to a Pointer

"Pointer" concept includes iterators — `{}` means null

Demo: Lifetime safety

```
main: () -> int = {
    words: std::vector<std::string> =
        ( "decorated", "hello", "world" );

    p: *std::string = nullptr;

    // ... more code ...

    print_and_decorate( p* );
}
```

```
demo.cpp2(6,23): error: pointer cannot be initialized to null or int - leav
e it uninitialized and then set it to a non-null value when you have one (a
t 'nullptr')
demo.cpp2: error: null initialization detected
  ==> program violates lifetime safety guarantee - see previous errors
```

```
main: () -> int = {
    words: std::vector<std::string> =
        ( "decorated", "hello", "world" );

    p: *std::string;

    // ... more code ...

    print_and_decorate( p* );
}
```

```
demo.cpp2(10,25): error: local variable p is used before it was initialized
  ==> program violates initialization safety guarantee - see previous errors
```

```
main: () -> int = {
    words: std::vector<std::string> =
        ( "decorated", "hello", "world" );
    p: *std::string;

    // ... more code ...
    if std::rand()%2 {
        p = words.front()&;
    }

    print_and_decorate( p* );
}
```

```
demo.cpp2(5,5): error: local variable p must be initialized on both branches
or neither branch
demo.cpp2(8,5): error: "if" initializes p on:
  branch starting at line 8
but not on:
  implicit else branch
  ==> program violates initialization safety guarantee - see previous errors
```

```
main: () -> int = {
    words: std::vector<std::string> =
        ( "decorated", "hello", "world" );
    p: *std::string;

    // ... more code ...
    if std::rand()%2 {
        p = words.front()&;
    }
    else {
        p = words.back()&;
    }

    print_and_decorate( p* );
}
```

```
demo.cpp2... ok (all Cpp2, passes safety checks)
```

Roadmap

- Motivation & approach
- History (since 2015)
- Safety
  - Type safety — CppCon 2021
  - Bounds safety
  - Lifetime safety — CppCon 2015
  - Initialization safety — CppCon 2020
- Simplicity examples
  - Parameter passing — CppCon 2020

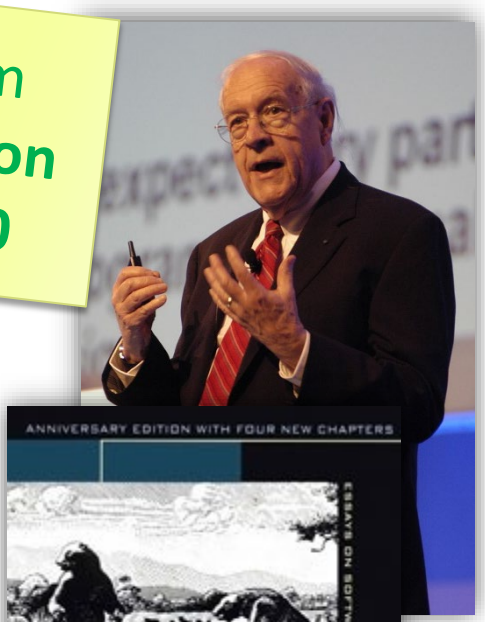
Metrics to aim for

“50× safer” means
98% fewer CVEs & bugs in these categories

“10× simpler” means
90% less total guidance to teach in C++ books and courses

Fred Brooks: Complexity

from CppCon 2020

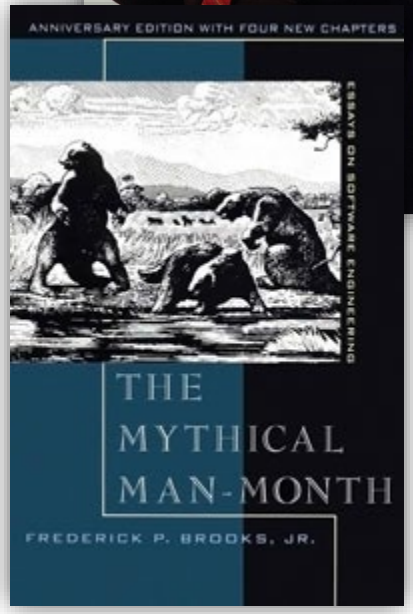


Essential complexity

Inherent in the problem,
present in any solution

Accidental complexity PUSH

Artifact of a specific solution design



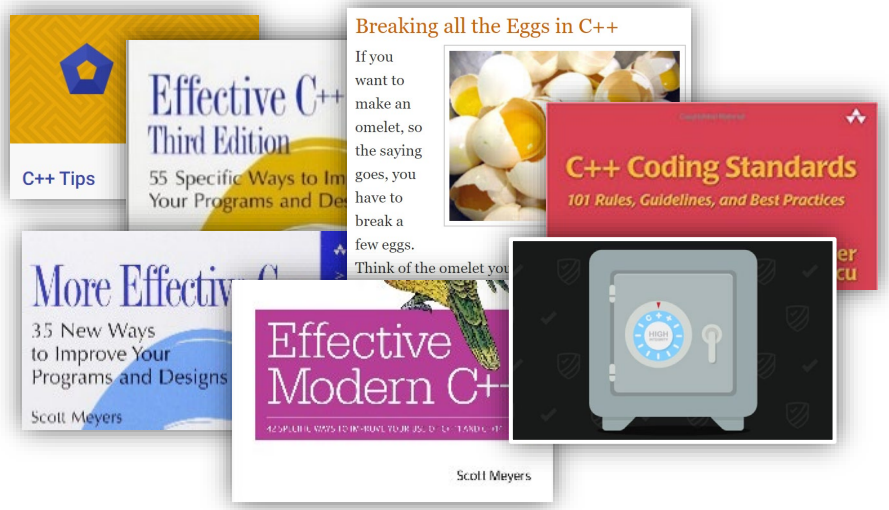
Breakdown of first 638 rules catalogued

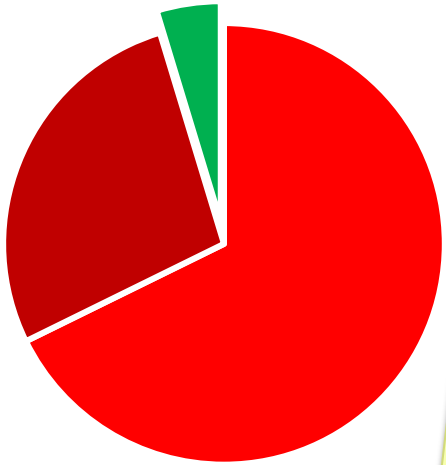
533 language

25 essential + minimal

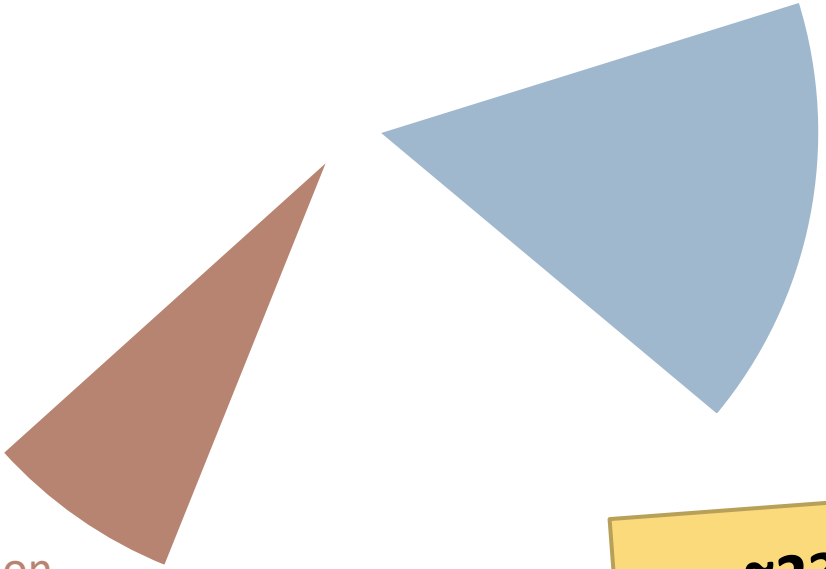
147 ‘essential’ + improvable

361 accidental + improvable





from **CppCon 2020**



from **CppCon 2020**

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

“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);
}
```

from **CppCon 2020**

“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);
}
```

from CppCon 2020

“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);
}
```

from CppCon 2020

“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
}
```

from CppCon 2020

Demo: Initialization safety

```
main: () -> int = {
    x: std::string;          // note: uninitialized!
    if flip_a_coin() {
        x = "xyzzy";
    } else {

    }
    print_decorated(x);
}
```

```
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 );  // note: constructs x!
    }
    print_decorated(x);
}

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

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

```
main: () -> int = {
    x: std::string;                 // note: uninitialized!
    if flip_a_coin() {
        x = "xyzzy";
    } else {
        fill( out x, "plugh", 40 ); // note: constructs x!
    }
    print_decorated(x);
}

fill: (out x: std::string,
       value: std::string,
       count: int)
    [[pre: value.ssize() >= count,
           "value must contain at least count chars"]]
```

```
C:\test\demo>cppfront demo.cpp2 -a
demo.cpp2... ok (mixed Cpp1/Cpp2, Cpp2 code passes safety checks)
```

```
C:\test\demo>demo
>> [xyzzy]

C:\test\demo>demo
demo.cpp2(20) fill: Contract violation: value must contain at least count chars
```

```
f: () -> (i: int, s: std::string) = {
    //  note: i and s are uninitialized!

    i = 10;              // constructs i
    if flip_a_coin() {
        s = "xyzzy";     // constructs s
    }
    else {
        s = "plugh";     // constructs s
        i = 998;         // assigns to i
    }
    s = s + "-ish";      // assigns to s
    return;              // moves from i and s
}

auto main() {            // normal Cpp1 code
    auto [a,b] = f();    // structured bindings
    print("a", a);
    print("b", b);
}
```

```
C:\test\demo>cppfront demo.cpp2
demo.cpp2... ok (mixed Cpp1/Cpp2, Cpp2 code
passes safety checks)
```

```
C:\test\demo>demo
a is 10
b is xyzzy-ish

C:\test\demo>demo
a is 998
b is plugh-ish
```

Roadmap

- Motivation & approach
- History (since 2015)
- Safety
  - Type safety — CppCon 2021
  - Bounds safety
  - Lifetime safety — CppCon 2015
  - Initialization safety — CppCon 2020
- Simplicity examples
  - Parameter passing — CppCon 2020

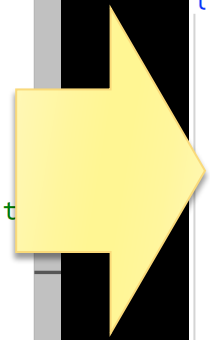
Metrics to aim for

“50× safer” means
98% fewer CVEs & bugs in these categories

“10× simpler” means
90% less total guidance to teach in C++ books and courses

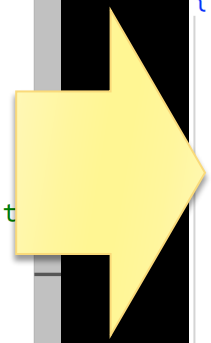
Demo: (More) parameter passing

```
parameter_styles: (
    in      a: std::string, // "in" is default
    copy    b: std::string,
    inout   c: std::string,
    move    d: std::string,
    forward e: std::string
    )
= {
    b += "plugh";

    if std::time(nullptr)%2 == 0 {
        copy_from(b);   // definite last use
    } else {
        copy_from(b&);  // NB: better not move from t
        copy_from(d);
    }

    copy_from(e);
}

main: () -> int = {
    v: std::string = "xyzzy";
    w: std::string = "xyzzy";
    x: std::string = "xyzzy";
    y: std::string = "xyzzy";
    z: std::string = "xyzzy";
    parameter_styles( v, w, x, move y, z );
}
```



```
auto parameter_styles(
    cpp2::in<std::string> a, // "in" is default
    std::string b,
    std::string& c,
    std::string&& d,
    auto&&  e
    ) -> void
requires std::is_same_v<CPP2_TYPEOF(e), std::string>
{
    b += "plugh";

    if (std::time(nullptr) % 2 == 0) {
        copy_from(std::move(b));// definite last use
    } else {
        copy_from(&b);  // NB: better not move from this
        copy_from(std::move(d));
    }

    copy_from(CPP2_FORWARD(e));
}

[[nodiscard]] auto main() -> int{
    std::string v { "xyzzy" };
    std::string w { "xyzzy" };
    std::string x { "xyzzy" };
    std::string y { "xyzzy" };
    std::string z { "xyzzy" };
    parameter_styles( v, w, x, std::move(y), z);
}
```

```
parameter_styles: (
    in      a: std::string, // "in" is default
    copy    b: std::string,
    inout   c: std::string,
    move    d: std::string,
    forward e: std::string
    )
= {
    b += "plugh";

    if std::time(nullptr)%2 == 0 {
        copy_from(b);   // definite last use
    } else {
        copy_from(b&);  // NB: better not move from t
        copy_from(d);
    }

    copy_from(e);
}

main: () -> int = {
    v: std::string = "xyzzy";
    w: std::string = "xyzzy";
    x: std::string = "xyzzy";
    y: std::string = "xyzzy";
    z: std::string = "xyzzy";
    parameter_styles( v, w, x, move y, z );
}
```

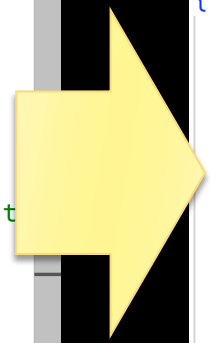


```
auto parameter_styles(
    cpp2::in<std::string> a, // "in" is default
    std::string b,
    std::string& c,
    std::string&& d,
    auto&&  e
    ) -> void
requires std::is_same_v<CPP2_TYPEOF(e), std::string>
{
    b += "plugh";

    if (std::time(nullptr) % 2 == 0) {
        copy_from(std::move(b));// definite last use
    } else {
        copy_from(&b);  // NB: better not move from this
        copy_from(std::move(d));
    }

    copy_from(CPP2_FORWARD(e));
}

[[nodiscard]] auto main() -> int{
    std::string v { "xyzzy" };
    std::string w { "xyzzy" };
    std::string x { "xyzzy" };
    std::string y { "xyzzy" };
    std::string z { "xyzzy" };
    parameter_styles( v, w, x, std::move(y), z);
}
```

```
parameter_styles: (
    in      a: std::string, // "in" is default
    copy    b: std::string,
    inout   c: std::string,
    move    d: std::string,
    forward e: std::string
    )
= {
    b += "plugh";

    if std::time(nullptr)%2 == 0 {
        copy_from(b);   // definite last use
    } else {
        copy_from(b&);  // NB: better not move from t
        copy_from(d);
    }

    copy_from(e);
}

main: () -> int = {
    v: std::string = "xyzzy";
    w: std::string = "xyzzy";
    x: std::string = "xyzzy";
    y: std::string = "xyzzy";
    z: std::string = "xyzzy";
    parameter_styles( v, w, x, move y, z );
}
```

```
auto parameter_styles(
    cpp2::in<std::string> a, // "in" is default
    std::string b,
    std::string& c,
    std::string&& d,
    auto&&  e
    ) -> void
requires std::is_same_v<CPP2_TYPEOF(e), std::string>
{
    b += "plugh";

    if (std::time(nullptr) % 2 == 0) {
        copy_from(std::move(b));// definite last use
    } else {
        copy_from(&b);  // NB: better not move from this
        copy_from(std::move(d));
    }

    copy_from(CPP2_FORWARD(e));
}

[[nodiscard]] auto main() -> int{
    std::string v { "xyzzy" };
    std::string w { "xyzzy" };
    std::string x { "xyzzy" };
    std::string y { "xyzzy" };
    std::string z { "xyzzy" };
    parameter_styles( v, w, x, std::move(y), z);
}
```

```
parameter_styles: (
    in      a: std::string, // "in" is default
    copy    b: std::string,
    inout   c: std::string,
    move    d: std::string,
    forward e: std::string
    )
= {
    b += "plugh";

    if std::time(nullptr)%2 == 0 {
        copy_from(b);   // definite last use
    } else {
        copy_from(b&);  // NB: better not move from t
        copy_from(d);
    }

    copy_from(e);
}

main: () -> int = {
    v: std::string = "xyzzy";
    w: std::string = "xyzzy";
    x: std::string = "xyzzy";
    y: std::string = "xyzzy";
    z: std::string = "xyzzy";
    parameter_styles( v, w, x, move y, z );
}
```



```
auto parameter_styles(
    cpp2::in<std::string> a, // "in" is default
    std::string b,
    std::string& c,
    std::string&& d,
    auto&&  e
    ) -> void
requires std::is_same_v<CPP2_TYPEOF(e), std::string>
{
    b += "plugh";

    if (std::time(nullptr) % 2 == 0) {
        copy_from(std::move(b));// definite last use
    } else {
        copy_from(&b);  // NB: better not move from this
        copy_from(std::move(d));
    }

    copy_from(CPP2_FORWARD(e));
}

[[nodiscard]] auto main() -> int{
    std::string v { "xyzzy" };
    std::string w { "xyzzy" };
    std::string x { "xyzzy" };
    std::string y { "xyzzy" };
    std::string z { "xyzzy" };
    parameter_styles( v, w, x, std::move(y), z);
}
```

```
parameter_styles: (
    in      a: std::string, // "in" is default
    copy    b: std::string,
    inout   c: std::string,
    move    d: std::string,
    forward e: std::string
    )
= {
    b += "plugh";

    if std::time(nullptr)%2 == 0 {
        copy_from(b);   // definite last use
    } else {
        copy_from(b&);  // NB: better not move from t
        copy_from(d);
    }

    copy_from(e);
}

main: () -> int = {
    v: std::string = "xyzzy";
    w: std::string = "xyzzy";
    x: std::string = "xyzzy";
    y: std::string = "xyzzy";
    z: std::string = "xyzzy";
    parameter_styles( v, w, x, move y, z );
}
```

```
auto parameter_styles(
    cpp2::in<std::string> a, // "in" is default
    std::string b,
    std::string& c,
    std::string&& d,
    auto&&  e
    ) -> void
requires std::is_same_v<CPP2_TYPEOF(e), std::string>
{
    b += "plugh";

    if (std::time(nullptr) % 2 == 0) {
        copy_from(std::move(b));// definite last use
    } else {
        copy_from(&b);  // NB: better not move from this
        copy_from(std::move(d));
    }

    copy_from(CPP2_FORWARD(e));
}

[[nodiscard]] auto main() -> int{
    std::string v { "xyzzy" };
    std::string w { "xyzzy" };
    std::string x { "xyzzy" };
    std::string y { "xyzzy" };
    std::string z { "xyzzy" };
    parameter_styles( v, w, x, std::move(y), z);
}
```

```
parameter_styles: (
    in      a: std::string, // "in" is default
    copy    b: std::string,
    inout   c: std::string,
    move    d: std::string,
    forward e: std::string
    )
= {
    b += "plugh";

    if std::time(nullptr)%2 == 0 {
        copy_from(b);   // definite last use
    } else {
        copy_from(b&);  // NB: better not move from t
        copy_from(d);
    }

    copy_from(e);
}

main: () -> int = {
    v: std::string = "xyzzy";
    w: std::string = "xyzzy";
    x: std::string = "xyzzy";
    y: std::string = "xyzzy";
    z: std::string = "xyzzy";
    parameter_styles( v, w, x, move y, z );
}
```

```
auto parameter_styles(
    cpp2::in<std::string> a, // "in" is default
    std::string b,
    std::string& c,
    std::string&& d,
    auto&&  e
    ) -> void
requires std::is_same_v<CPP2_TYPEOF(e), std::string>
{
    b += "plugh";

    if (std::time(nullptr) % 2 == 0) {
        copy_from(std::move(b));// definite last use
    } else {
        copy_from(&b);  // NB: better not move from this
        copy_from(std::move(d));
    }

    copy_from(CPP2_FORWARD(e));
}

[[nodiscard]] auto main() -> int{
    std::string v { "xyzzy" };
    std::string w { "xyzzy" };
    std::string x { "xyzzy" };
    std::string y { "xyzzy" };
    std::string z { "xyzzy" };
    parameter_styles( v, w, x, std::move(y), z);
}
```

Roadmap

- Motivation & approach
- History (since 2015)
- Safety
  - Type safety — CppCon 2021
  - Bounds safety
  - Lifetime safety — CppCon 2015
  - Initialization safety — CppCon 2020
- Simplicity examples
  - Parameter passing — CppCon 2020

Metrics to aim for

“50× safer” means
98% fewer CVEs & bugs in these categories

“10× simpler” means
90% less total guidance to teach in C++ books and courses

Consistency: Functions

```
named_function : (i: int) = print(i);

main: ()->int = {
    vec: std::vector = ( 1, 2, 3, 5, 8, 13 );

    std::ranges::for_each
        ( vec, : (i: int) = print(i); );

    for vec do : (i: int) = print(i);
}
```

named function

unnamed function

range-for body

Consistency: Capture (aka “paste value”)

```
main: ()->int = {
    s := "-ish\n";
    vec: std::vector = ( 1, 2, 3, 5, 8, 13 );

    std::ranges::for_each
        ( vec, : (i:_) = { std::cout << i << s$; } );


}
```

unnamed function capture

Consistency: Capture (aka “paste value”)

```
push_back: (coll:_, value:_)
    [[post: coll.size() == coll.size()$ + 1]]
= { ... }

main: ()->int = {
    s := "-ish\n";
    vec: std::vector = ( 1, 2, 3, 5, 8, 13 );

    std::ranges::for_each
        ( vec, : (i:_) = { std::cout << i << s$; } );


}
```

post: “old” state capture

unnamed function capture

Consistency: Capture (aka “paste value”)

```
push_back: (coll:_, value:_)
    [[post: coll.size() == coll.size()$ + 1]]
= { ... }

main: ()->int = {
    s := "-ish\n";
    vec: std::vector = ( 1, 2, 3, 5, 8, 13 );

    std::ranges::for_each
        ( vec, : (i:_) = { std::cout << i << s$; } );

    message := "Someone 2 meters high is tall(s)$";
    std::cout << message;
}
```

post: “old” state capture

unnamed function capture

string interpolation

Roadmap

Motivation & approach

History (since 2015)

Safety

- Type safety — CppCon 2021
- Bounds safety
- Lifetime safety — CppCon 2015
- Initialization safety — CppCon 2020

Simplicity examples

- Parameter passing — CppCon 2020

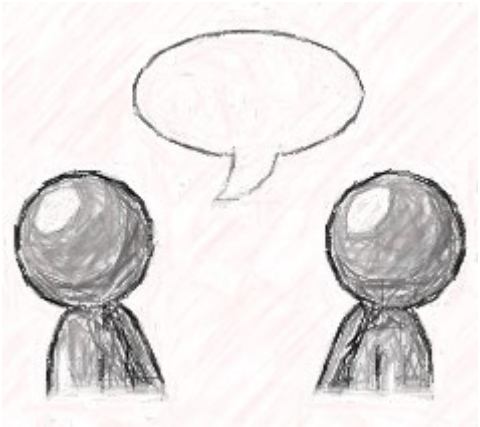
Metrics to aim for

“50× safer” means
98% fewer CVEs & bugs in these categories

“10× simpler” means
90% less total guidance to teach in C++ books and courses

An observation

Think about the ***words and ideas*** we've been using



An observation

Think about the ***words and ideas*** we've been using

None of them are weird foreign terms or concepts
from Haskell/Lisp/Ada/Java/Eiffel/Go/Scheme/…

All of them are already deeply familiar to C++ developers,
they're how we talk, how we think… ***only nicer***

Medium term: Complete basic language

`<T: type is Concept, I: int>`	explicit template parameter lists
classes	user-defined `type` incl. defaults (e.g., explicit ctors) incl. type invariants (completing contracts) trying `operator=(out this, ...)` unification
reflection, generation, metaclasses	using the parse tree
lightweight exceptions	`std::error_condition` value-based

Want to help? Medium-term project ideas

Editor support Syntax highlighting, UFCS autocomplete, …

Godbolt CE … with choice of Cpp1 compiler?

gc.new Opt-in arena, pay only for what you use
Real tracing GC alloc + real C++ destructors
Adapt and expand `github.com/hsutter/gcpp`

cpp2::draw Basic 2D canvas: lines, PNG, text
Basic keyboard & pointer input
“21st-century curses/conio.h”
… Header-only?

If you’re interested or have more ideas, please send me mail

Want to help? Longer-term project idea

1→2

frontcpp

Cpp1 → Cpp2 compiler – adapt a Cpp1 pretty-printer

Cpp1 idioms/patterns → use Cpp2 features

Ex: All pure virtual functions → `type(interface)`

Ex: Unconditional param deref → `[[pre Null: ptr]]`

If you're interested or have more ideas, please send me mail

what if we could do “C++11 ***feels like*** a new language” again, but broadly for the whole language?

support all C++20/23… evolution

embrace C++20/23… (e.g., default to C++20 modules, C++23 `import std;`)

“directed evolution” of C++ itself — compiling to C++20/23… keeps us honest

bring any results to ISO C++ evolution

Cpp2	Cpp1
	preprocessor, #define, #include, which std header to include, auto, [[nodiscard]], forward declarations, ordering dependencies, unsafe casts, uninitialized variables
one l-to-r decl syntax	most vexing parse, east const vs west const, inside-out declaration syntax, two variable declaration syntaxes, two free function declaration syntaxes, two irregular member function declaration syntaxes, lambda function declaration syntax
in, copy, inout, out	X vs X const params, deciding X vs X const& params, T vs T const& in templates
move	references (&, X&&, T&&) throughout the language, and explaining X&& vs T&&
forward	std::move, why std::move doesn't move, general overuse of std::move, why not "return std::move," why && isn't rvalue reference for template types, how to write move parameters for template types
named return values	std::forward, spelling perfect forwarding idiom right, why forwarding && is only for templated types, how to write forwarding && params for non-template types
new<T>, span	how to enable NRVO, how to return multiple values via anonymous pair/tuple, how to return multiple named values using separately defined struct
postfix operators	new, delete, owning raw *, memory leaks, 0 as int/pointer, NULL, null dereference
is	pointer arithmetic, out of bounds subscripting, raw arrays, implicit array→ptr decay
as, gsl::narrow	(*x)++, ++x vs x++, and (int)-for-postfix dummy parameter convention
	is_same_v, is_base_of_v, dynamic_cast, std::holds_alternative<T>, my_any.type() == typeid(T), my_optional.has_value
	union, va_arg arguments, C-style casts, reinterpret_cast, const_cast, function-style casts, static_cast, dynamic_cast, std::get<T>/<T&>, std::any_cast<T>/<T*>, opt.value()
	don't use reinterpret_cast, don't use static_cast for arithmetic types, don't cast between pointer types that are the same, don't cast between pointer types where the conversion could be implicit, don't use const_cast, don't use static_cast to downcast, don't use a variable before it has been initialized
$	lambda capture introducers (+ postcondition 'old' values? + string interpolation?)

Can C++ be 10× simpler & safer … ?

Herb Sutter

*"Inside C++, there is a **much smaller and cleaner language** struggling to get out."*

— B. Stroustrup (D&E, 1994)

*"Say **10% of the size of C++** in definition and similar in front-end compiler size. …*
***Most of the simplification would come from generalization**."*

— B. Stroustrup (ACM HOPL-III, 2007)