

Peering forward: C++'s next decade

Herb Sutter



Major advances are on track

- std::execution (concurrency *and* parallelism)
- Type and memory safety improvements
- Reflection + code generation (aka ‘injection’/...)
- Contracts



means “*some initial parts already voted into C++26*”

Major advances are on track

`std::execution` (*concurrency and parallelism*)



Type and memory safety improvements

- target: **parity** with other modern languages

Reflection + code generation (aka ‘injection’/...)

- part of our **sea change to compile-time programming**
- arguably **most-impactful** feature ever added
- I expect will **dominate** our next decade

Contracts

1



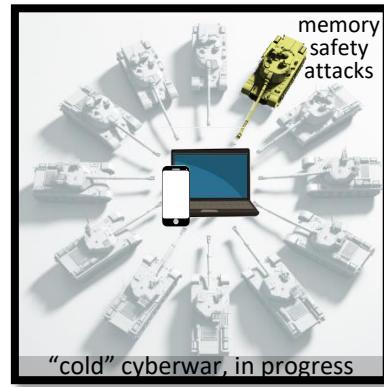
Reflection

2

Safety

$1 \cap 2$ includes

- timing: starting in C++26 🤘
- technical: ↓↓UB
- this week: >dozen talks
- 3: simplifying our code & tools



3

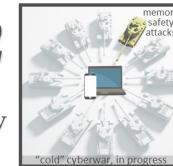
Simplicity

1



Reflection

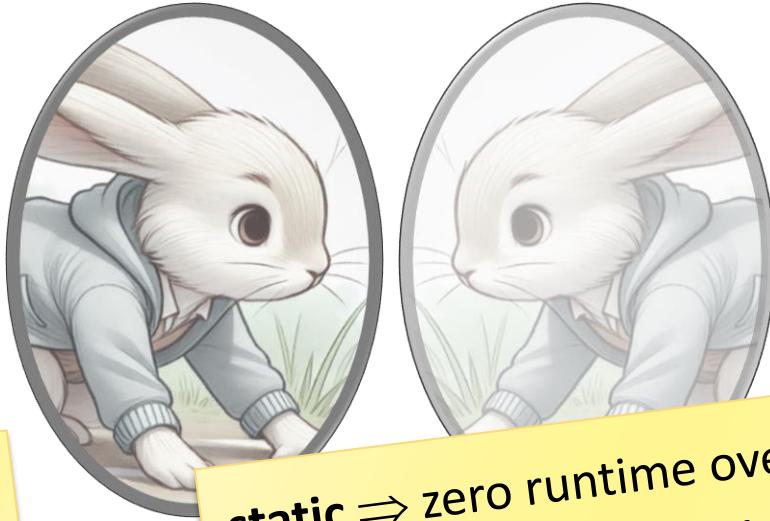
2



1

Reflection

short definition
the program can **see** itself
and **generate** itself
... hence, **meta**programs



static \Rightarrow zero runtime overhead
not run-time reflection
(this is not Java or C#)
of course, to use some information
at run time, just store it!

part of the sea change → **compile-time meta** programming in C++

*“The world’s big things only can be done
by paying attention to their **humble beginnings.**” — Lao Tzu*

Erwin Unruh: The most famous C++ program that **doesn't compile**

```
template <int i> struct D { D(void*); operator int(); };

template <int p, int i> struct is_prime {
    enum { prim = (p%i) && is_prime<(i>2 ? p : 0), i-1>::prim };
};

template <int i> struct Prime_print {
    Prime_print<i-1> a;
    enum { prim = is_prime<i, i-1>::prim };
    void f() { D<i> d = prim; }
};

struct is_prime<0,0> { enum
struct is_prime<0,1> { enum
struct Prime_print<2> { enum

main () {
    Prime_print<10> a;
}
```

1994:
TC!

Original Metaware compiler error messages

```
Type 'enum{}' can't be converted to type 'D<2>' ("primes.cpp", L2/C25).
Type 'enum{}' can't be converted to type 'D<3>' ("primes.cpp", L2/C25).
Type 'enum{}' can't be converted to type 'D<5>' ("primes.cpp", L2/C25).
Type 'enum{}' can't be converted to type 'D<7>' ("primes.cpp", L2/C25).
Type 'enum{}' can't be converted to type 'D<11>' ("primes.cpp", L2/C25).
Type 'enum{}' can't be converted to type 'D<13>' ("primes.cpp", L2/C25).
Type 'enum{}' can't be converted to type 'D<17>' ("primes.cpp", L2/C25).
Type 'enum{}' can't be converted to type 'D<19>' ("primes.cpp", L2/C25).
Type 'enum{}' can't be converted to type 'D<23>' ("primes.cpp", L2/C25).
Type 'enum{}' can't be converted to type 'D<29>' ("primes.cpp", L2/C25).
```

Modern C++ Design

文 A 3 languages ▾

Article Talk

Read Edit View history Tools ▾

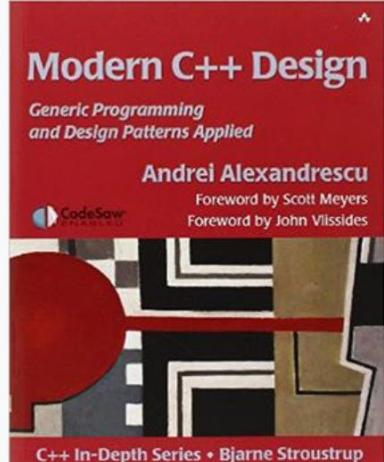
From Wikipedia, the free encyclopedia

Modern C++ Design: Generic Programming and Design Patterns Applied

Patterns Applied is a book written by [Andrei Alexandrescu](#), published in 2001 by [Addison-Wesley](#). It has been regarded as "one of the most important C++ books" by [Scott Meyers](#).^[1]

The book makes use of and explores a [C++](#) programming technique called [template metaprogramming](#). While Alexandrescu didn't invent the technique, he has popularized it among programmers. His book contains solutions to practical problems which C++ programmers

Modern C++ Design



Author	Andrei Alexandrescu
Language	English
Subject	C++
Publisher	Addison-Wesley
Publication date	2001

2001:
TMP 



ISO C++ & **constexpr**

C++
23

more math, **allocators**,
non-literal params, ...

C++
20

new/delete, virtual, lambdas,
try/catch, **vector/string**, ...

C++
17

lambdas, constexpr destructors,
if **constexpr**, static_assert, ...

C++
14

more statements, local variables,
if/switch, **for/while**, member functions, ...

C++
11

one return statement



FROM CPPCON 2017



Ben Deane



Jason Turner

ISO C++ & GPUs

CUDA
12 (2022)

concepts, ranges,
'moar constexpr', ...

CUDA
11 (2020)

new/delete, constexpr, CTAD,
structured bindings, ...

CUDA
7 (2015)

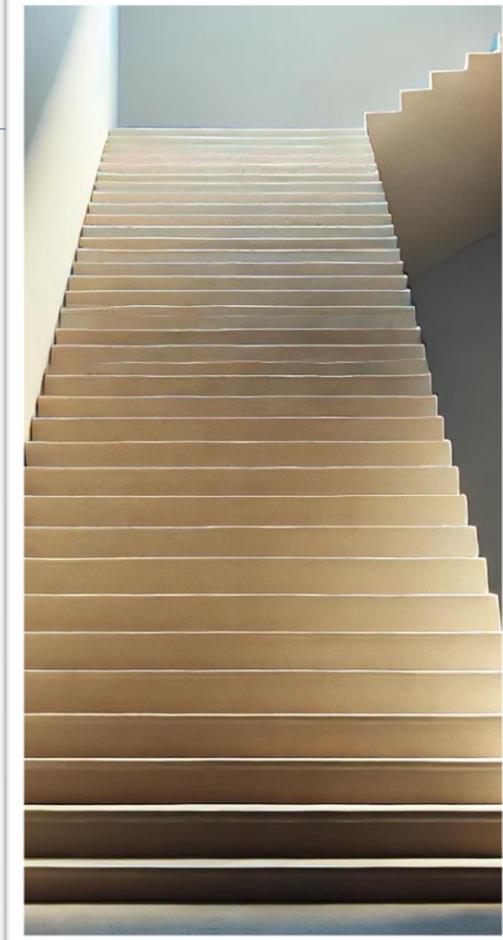
lambdas, range-for, variadic
templates, static_assert, ...

'Moar
shaders!'

more statements, local variables,
if/switch, for/while, ...

Shaders

throw a few instructions at a pixel as it flies by



ISO C++ & `constexpr`

C++
23

more math, **allocators**,
non-literal params, ...

C++
20

new/delete, virtual, lambdas,
try/catch, **vector/string**, ...

C++
17

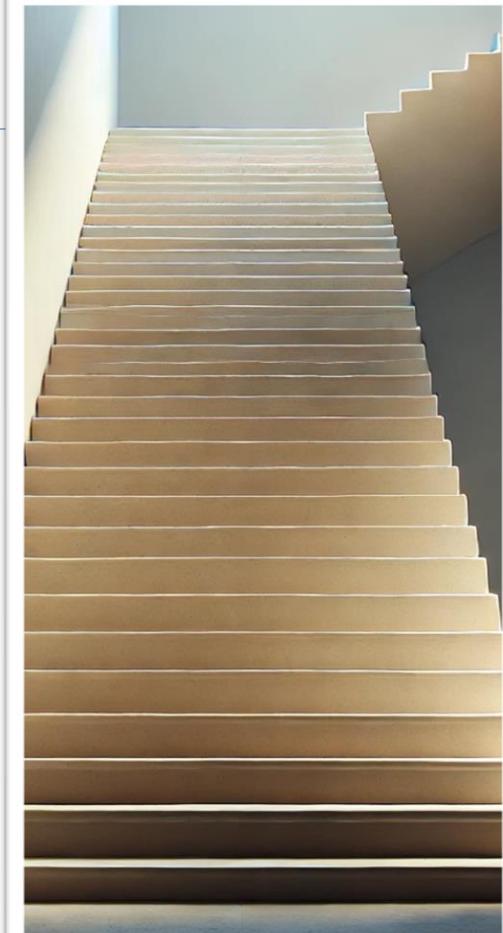
lambdas, constexpr destructors,
if constexpr, static_assert, ...

C++
14

more statements, local variables,
if/switch, for/while, member functions, ...

C++
11

one return statement



pause to consider what this implies

C++ is the language we want at compile time

C++ is the language we want on GPUs

ISO C++ & reflection

P3294

code generation/injection

P3157

generative extensions

P3273

introspection of closure types

P3096

function parameters

P2996

reflection, splicers, meta::info, metafunctions



P2996 example: Basic command-line option parser

Thanks to Matúš Chochlík!

```
struct MyOpts {
    std::string file_name = "input.txt";      // "--file_name <string>"
    int        count     = 1;                  // "--count <int>"
};

int main(int argc, char *argv[]) {
    MyOpts opts = parse_options<MyOpts>(
        std::vector<std::string_view>(argv+1, argv+argc)
    );
    std::cout << "opts.file=" << opts.file_name << '\n';
    std::cout << "opts.count=" << opts.count     << '\n';
}
```

the **type itself** is a function input

```
template<typename Opts>
constexpr auto parse_options(std::span<std::string_view const> args) -> Opts
{
    Opts opts;                                expansion statement
    template for (constexpr auto dm : nonstatic_data_members_of(^Opts)) {
        auto it = std::ranges::find_if(args,
            [](std::string_view arg){
                return arg.starts_with("--")
                    && arg.substr(2) == identifier_of(dm);
            });
        if (it == args.end()) {
            continue; // not provided, use default
        } else if (it + 1 == args.end()) {
            std::print(stderr, "Option {} is missing a value\n", *it);
            std::exit(EXIT_FAILURE);
        }
    }
}
```

^ reflection expression
and std::meta:: metafunctions

```
    && arg.substr(2) == identifier_of(dm));
}

if (it == args.end()) {
    continue; // not provided, use default
} else if (it + 1 == args.end()) {
    std::print(stderr, "Option {} is missing a value\n", *it);
    std::exit(EXIT_FAILURE);
}

using T = typename[:type_of(dm):];
auto iss = std::ispanstream(it[1]);
if (iss >> opts.[:dm:]; !iss) { — splices
    std::print(stderr, "{} is not a valid {}\n",
               *it, display_string_of(dealias(^^T)));
    std::exit(EXIT_FAILURE);
}
return opts;
}
```

e.g., “**int**” at compile time
better than typeid(T).name()
(e.g., “**i**” at **run time**)

As seen on COMPILER EXPLORER

x86-64 clang (experimental P2996) ▾  

```
--count 42 --file_name scott-meyers-emcpp.pdf
```

Program returned: 0

Program stdout

```
opts.file=scott-meyers-emcpp.pdf
opts.count=42
```

x86-64 clang (experimental P2996) ▾  

```
--count scott-meyers-emcpp.pdf
```

Program returned: 1

Program stderr

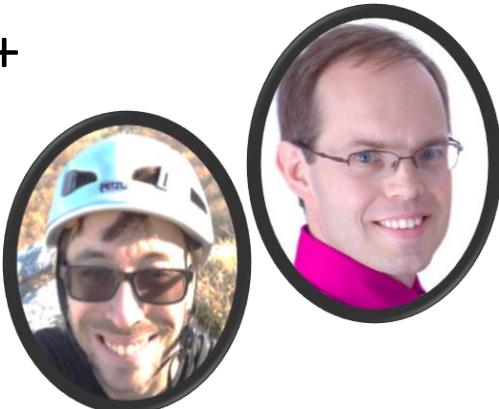
```
-count is not a valid int
```

P2996 implementations

Two current implementations tracking P2996 ++

EDG-based Daveed Vandevoorde (EDG)

Clang-based Dan Katz (Bloomberg)



Other related prototype implementations

Clang-based Andrew Sutton et al. (Lock3)

Cppfront mine



Circle Sean Baxter



Pointers to forward-declared sessions...

Tuesday

09:00



Reflection based libraries to look forward to

Saksham Sharma

Wednesday

16:45



Reflection Is Not Contemplation

Andrei Alexandrescu



Friday

10:30



Implementing Reflection using the new C++20 Tooling
Opportunity: Modules

Maiko Steeman

KEYNOTE



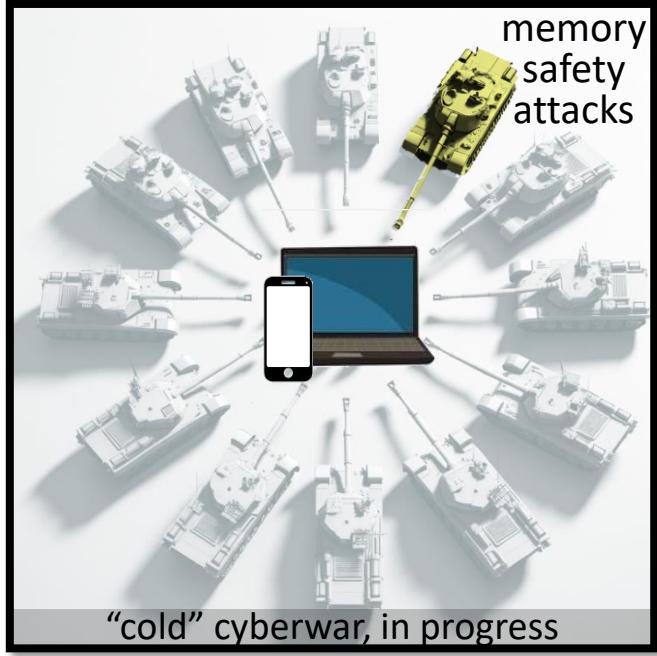
Gazing Beyond Reflection for C++26

Daveed Vandevoorde



2

Safety



See essay here: [*tinyurl.com/sutter-safety*](http://tinyurl.com/sutter-safety)
(herbsutter.com/2024/03/11/safety-in-context)

Terms (per ISO/IEC 23643:2020)

Software **security** (or “cybersecurity” or similar)

making software able to **protect its assets** from a malicious attacker

examples: securing power grids, hospitals, banks, personal data, secrets, ...

Software **safety** (or “life safety” or similar)

making software free from unacceptable risk of **causing unintended harm** to humans, property, or the environment

examples: hospital equipment, autonomous vehicles/weapons

Programming language **safety** (incl. memory safety)

static and dynamic guarantees about **program correctness**

helps both the others — [more on this in section 3]

tl;dr

The actual problem: Safety parity (not perfection)

4 low-hanging priority targets:

type, bounds, initialization & lifetime safety

== 4 most severe CWE categories, 4 that all MSLs do better at

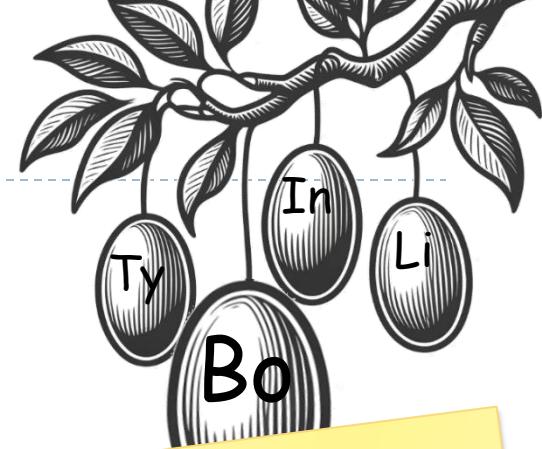
Progress

Key **safe libraries** moving from 3rd-party to standard
(e.g., C++20 std::span<T> to replace pointer math)

Safety-related **undefined behavior** being removed
(e.g., Mar 2024: reading uninit stack vars not UB!)

Key **static safety rules Profiles**: most known, “shift-left” to compile time

Add **dynamic safety checks** as needed (e.g., bounds, null)



+ what implementations are doing outside the standard,
e.g., **-fhardened**, STL safety

Work in the same kitchen, hold the same knife...



today: “watch out”

performance & control by default,
safety always available

Work in the same kitchen, hold the same knife...



today: “watch out”

performance & control by default,
safety always available



tomorrow: “opt out”

safety by default,
performance & control always available

Safety Profiles

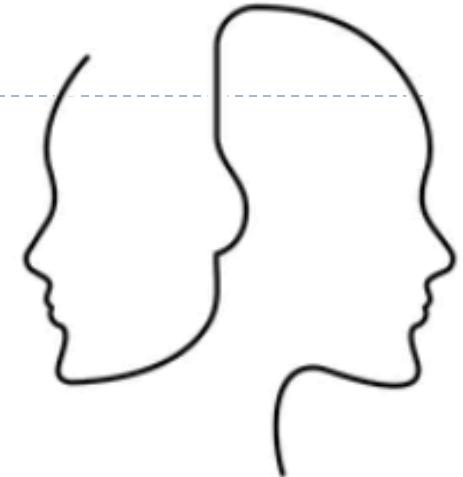
Progressing in WG21: Safety **Profiles** framework
(Stroustrup & Dos Reis)

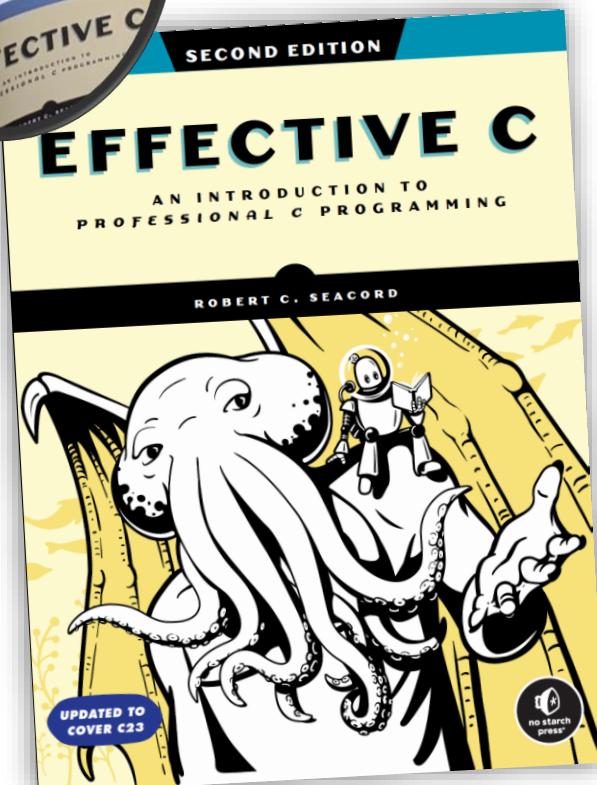
pro·file /'prō,fil/ noun

*a name for a **set of rules enforced at compile time**,
that guarantees the **absence of a class of defects**,
and the programmer can **enable for a volume of code***

Profiles can be added over time, and enabled **selectively & incrementally**

Key: take (mostly-already-known) rules, “shift-left” to compile time





from the Praise page

“... This book’s emphasis on the security aspects of C programming is unmatched ... use all of the available tools it presents to **avoid undefined behavior** in the C programs you write.”

— Pascal Cuoq, Chief Scientist, Trustinsoft

“An excellent introduction to modern C.”

— Francis Glassborow, ACCU

...

“This is why you should program in C. Because **other languages don’t open portals to hell.**”

— Michał Zalewski, former CISO, Snap Inc.

C++26: “erroneous behavior”

Undefined behavior means “anything can happen”

Usual examples nasal daemons, `format c:`

Actual bad examples time travel, leaking secrets

C++26 introduces a new tool: **erroneous** behavior

“Well-defined as being Just Wrong”

Not undefined \Rightarrow no time travel, or leaked secrets!

A general tool, but first applied to...



**Reading uninitialized local variables
is not undefined behavior in C++26!**



A C++26 compiler is required to write an “erroneous” value



C++26: “erroneous behavior”

Pause a moment and consider how this is important:

QUANTITATIVELY Automatically eliminates a significant fraction
(5%? 10%?) of vulnerabilities and other bugs

Qualitatively Delivered to existing code with **no manual
code changes**, just a recompile

⇒ seriousness about **adoptability** & improving safety of **existing** code

Ode to information disclosure

```
auto f1() {
    char a[] = {'s', 'e', 'c', 'r', 'e', 't' };
}

auto f2() {
    char a[6]; // or std::array<char,6>
    print(a); // today this likely prints "secret"
}

int main() {
    f1();
    f2();
}
```

C++26 “erroneous behavior”

```
auto f1() {
    char a[] = {'s', 'e', 'c', 'r', 'e', 't' };
}

auto f2() {
    char a[6]; // or std::array<char,6>
    print(a); // C++26: prints "SECRET" or "" or... but not "secret"
}

int main() {
    f1();
    f2();
}
```

for pre-C++26:
not de jure standard, but de facto available today
GCC, Clang: `-ftrivial-auto-var-init=pattern`
MSVC: `/RTC1`

The screenshot shows a C++ development environment with the following components:

- Editor:** Displays the source code in three files:
 - `f1()`: Contains a character array `a` initialized to "secret".
 - `f2()`: Contains a character array `a[6]` and a `print(a);` statement. This block is highlighted with a red box.
 - `main()`: Calls `f1()` and `f2()`.
- Toolbars:** Includes "File/Load", "+ Add new...", "Vim", "CppInsights", "Quick-bench", "C++", "x86-64 gcc (trunk)", "Compiler options...", and "Output" tabs.
- Output Panels:** Show the compiler output for four different compilers:
 - x86-64 gcc (trunk) (Compiler #1):** Prints "Program returned: 0" and "secret".
 - x86-64 clang (trunk) (Compiler #2):** Prints "Program returned: 0" and "secret".
 - x64 msvc v19.38 VS17.8 (Compiler #3):** Prints "Program returned: 0" and "secret".
 - x64 msvc v19.38 VS17.8 (Editor #1):** Shows the command `/std:c++latest` in the status bar.

```
File/Load + Add new... Vim CppInsights Quick-bench C++  
auto f1() {  
    char a[] = {'s', 'e', 'c', 'r', 'e', 't' };  
}  
  
auto f2() {  
    char a[6];  
    print(a); // today this likely prints "secret"  
}  
  
int main() {  
    f1();  
    f2();  
}
```

x86-64 gcc (trunk) -ftrivial-auto-var-init=partial
Output of x86-64 gcc (trunk) (Compiler #1)
Wrap lines Select all
Program returned: 0
????????
x86-64 clang (trunk) -ftrivial-auto-var-init=partial
Output of x86-64 clang (trunk) (Compiler #2)
Wrap lines Select all
Program returned: 0
????????
x64 msvc v19.38 VS17.8 /RTC1 std:c++latest
Output of x64 msvc v19.38 VS17.8 (Compiler #3)
Wrap lines Select all
Program returned: 0
????????

Q&A

Why not zero-init?

If zero isn't a program-meaningful value, just changes one bug into another

Not like a world where zero-init was always the language rule!

Actively hides real problems — makes uninitialization invisible to sanitizers

Can I opt out?

Yes (this is still C++) `int a [[indeterminate]] ;`

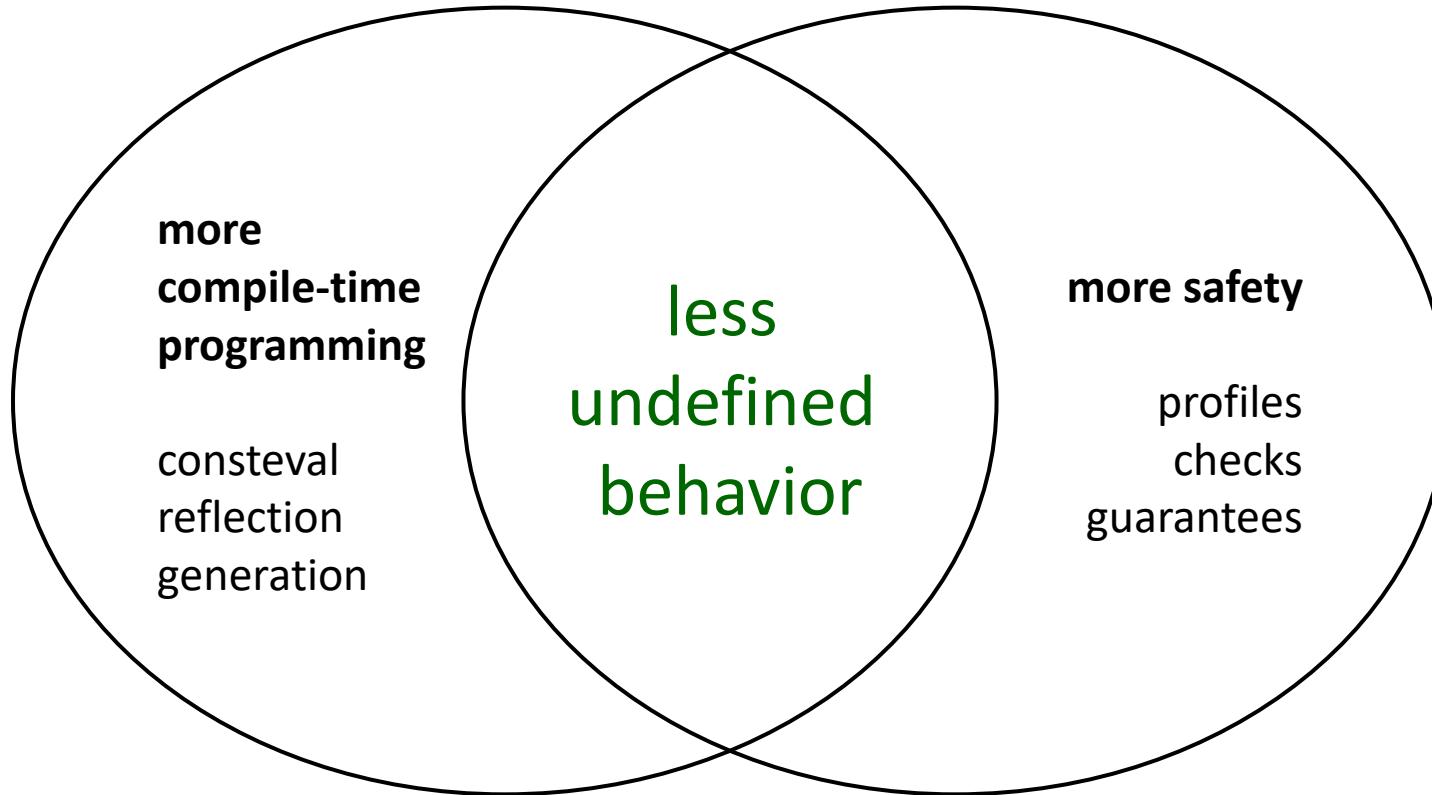
Why didn't C++ always do this?

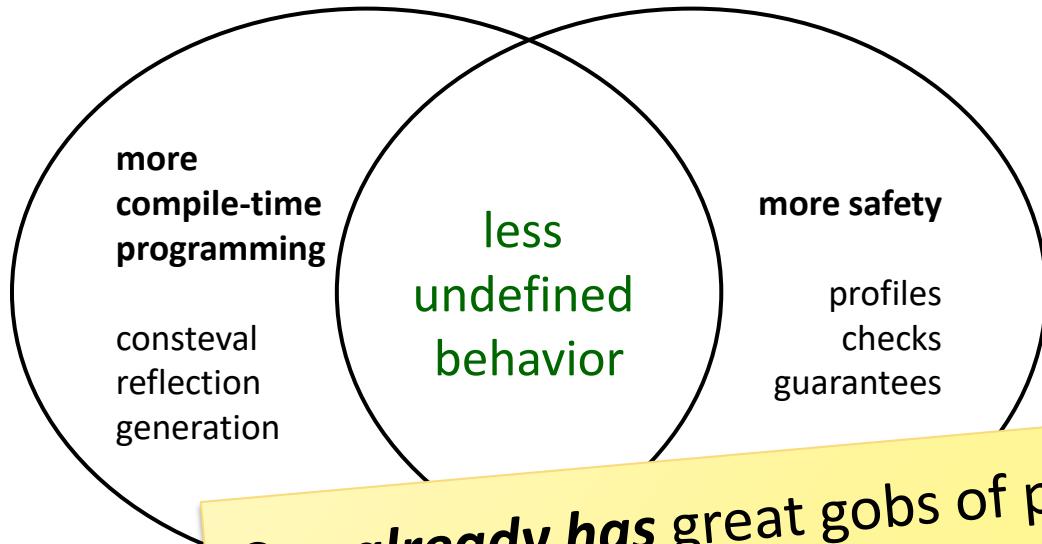
Cost, esp for large objects & buffers...

In section 3, I'll talk about how we can do even better...

Quick poll

Q: Do you think it's believable that C++ could evolve to eliminate most safety-related **undefined behavior**?





C++ **already has** great gobs of production UB-free code (*)

`constexpr/consteval` reject:

signed overflow, some pointer arithmetic, division by zero, ...

many are ***directly safety-related***

(e.g., bounds safety includes 'no pointer arithmetic, use span instead')

(*) core language UB in a context that requires a constant expression

```
#1 C++  
  
constexpr int f(int n)  
{  
    int r = n--;  
    for (; n > 1; --n) r *= n;  
    return r;  
}  
  
int main()  
{  
    constexpr int x = f(13);  
    return x;  
}
```

x86-64 gcc (trunk) (Editor #1) Output of x86-64 gcc (trunk) (Compiler #1)

A Wrap lines Select all

```
<source>: In function 'int main()':  
<source>:10:24:   in 'constexpr' expansion of 'f(13)'  
<source>:4:26: error: overflow in constant expression [-fpermissive]  
4 |     for (; n > 1; --n) r *= n;
```

x86-64 clang (trunk) (Editor #1) Output of x86-64 clang (trunk) (Compiler #2)

A Wrap lines Select all

```
<source>:10:19: error: constexpr variable 'x' must be initialized by  
a constant expression  
10 |     constexpr int x = f(13);  
          |           ^ ~~~~~
```

x64 msvc v19.38 VS17.8 (Editor #1) Output of x64 msvc v19.38 VS17.8 (Compiler #3)

A Wrap lines Select all

```
example.cpp  
<source>(10): error C2131: expression did not evaluate to a constant  
<source>(4): note: failure was caused by the '*' operation causing  
signed overflow during constant evaluation  
<source>(10): note: the call stack of the evaluation (the oldest
```

Pointers to forward-declared sessions...

Monday

11:00

- ✓ Security Beyond Memory Safety - Using Modern C++ to Avoid Vulnerabilities by Design
Max Hoffmann

15:15

- ✓ LLVM's Realtime Safety Revolution: Tools for Modern Mission Critical Systems
Christopher Apple • David Trevelyan

20:30

- ✓ Safety and Security Panel
Michael Wong

Wednesday

09:00

- ✓ Improving our safety with a quantities and units library
Mateusz Pusz
- ✓ Building Cppcheck - What We Learned from 17 Years of Development
Daniel Marjamäki

- ✓ Embracing an Adversarial Mindset for C++ Security



KEYNOTE

14:00

- ✓ Embracing an Adversarial Mindset for C++ Security
Amanda Rousseau



15:15

Thursday

09:00

- ✓ Building Safe and Reliable Surgical Robotics with C++
Milad Khaledyan

14:00

- ✓ Mix Assertion, Logging, Unit Testing and Fuzzing: Build Safer Modern C++ Application
Xiaofan Sun
- ✓ Perspectives on Contracts
Lisa Lippincott

Simplicity

3



simplification through generalization

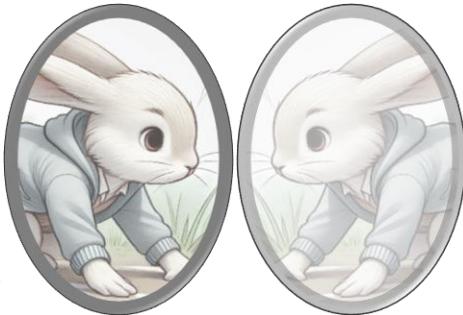
key: enable programmer to directly express intent

- ⇒ elevate coding patterns to declare “what” vs “how”
- ⇒ use the intent already in the code, instead of “how” annotations

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

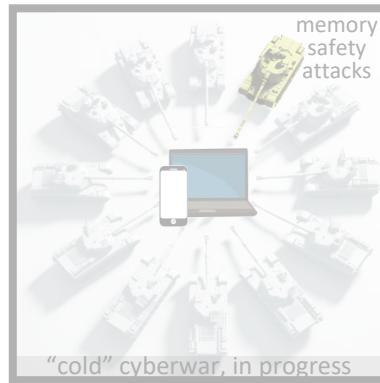
*“Say 10% of the size of C++ ... Most of the simplification would come from **generalization**. ”*
— Bjarne Stroustrup (ACM HOPL-III, 2007)

1



2

Safety



3

Simplicity



Simplification... by addition?

2017: Paper P0707 on metaclass functions

Main purpose: “yes we need **static reflection + source generation as game-changing**, and here are **North Star examples** of what’s possible”

2017 Toronto meeting: “Hi, I’m their [reflection proposers’] customer”

First major Cpp2 feature I brought to WG21 & conferences

Because it was key to **simplification**: Metaclass functions and parameter passing were the two biggest sources of simplification in Cpp2, because they let programmers **declare their intent**

Because it carried the **highest risk**: Would the committee & community accept that huge a leap forward in compile-time programming? Would a full reflection implementation actually work and not hit language/compiler limits?

P0707: Small but important sugar

```
// Example 1: Possible with P2996
namespace __prototype { class widget { /*...*/ }; }
consteval{ metafunc( ^__prototype::widget ); }

// e.g., generates something like:
//     class widget {
//         /* based on the reflection
//             of __prototype::widget */
//     };
```

```
// Example 2: P0707 proposes to let class(M) mean
//                 "apply M to the class being defined"
class(metafunc) widget{ /*...*/ };
// identical meaning as above
```

Quick refresher example

```
class IFoo {  
public:  
    virtual int f() = 0;  
    virtual void g(std::string) = 0;  
    virtual ~IFoo() = default;  
};
```



```
class(interface) IFoo {  
    int f();  
    void g(std::string);  
};
```

declaring our intent
⇒ the right defaults
⇒ generated functions
⇒ checked constraints

Now in EDG... godbolt.org/z/fex55qq5o

```
namespace __proto {
    class Widget {
        int f();
        void g(std::string);
    };
}
consteval { interface(^^__proto::Widget); }
```

// P0707 proposed sugar

```
class(interface) Widget {
    int f();
    void g(std::string);
};
```

```
class MyWidget : public Widget {
    int f()             override { return 42; }
    void g(std::string s) override { std::cout << s; }
};

int main() {
    unique_ptr<Widget> w = make_unique<MyWidget>();
    cout << w->f() << '\n';
    w->g( "xyzzy" );
}
```

EDG (experimental results)

```
Program returned: 0
Program stdout
42
xyzzy
```

Now in EDG... godbolt.org/z/fex55qq5o

```
consteval void interface(std::meta::info proto) {
    std::string_view name = identifier_of(proto);
    queue_injection(^^{
        class \id(name) {
            public:
                \tokens(make_interface_functions(proto))
                virtual ~\id(name)() = default;
        };
    });
}
```

composable ✓

Now in EDG... godbolt.org/z/fex55qq5o

```
consteval auto make_interface_functions(info proto) -> info {
    info ret = ^^{};
    for (info mem : members_of(proto)) {
        if (is_nonspecial_member_function(mem)) {
            ret = ^^{
                \tokens(ret)
                virtual [:\(\return_type_of(mem)):]
                    \id(identifier_of(mem)) (\tokens(parameter_list_of(mem))) = 0;
            };
        }
        else if (is_variable(mem)) {
            // --- reporting compile time errors not yet implemented ---
            // print_error( "interfaces may not contain data members" );
        }
        // etc. for other kinds of interface constraint checks
    }
    return ret;
}
```

In the box with cppfront so far... all build to pure ISO C++ & work with GCC/Clang/MSVC

interface	An abstract class having only pure virtual functions
polymorphic_base	A pure polymorphic base type that is not copyable or movable, and whose destructor is either public+virtual or protected+nonvirtual
ordered	A totally ordered type with <code>operator<=</code> that implements <code>strong_ordering</code> . Also: <code>weakly_ordered</code> , <code>partially_ordered</code>
copyable	A type that has copy/move construction/assignment
basic_value	A <code>copyable</code> type that has public default construction and destruction (generated if not user-written) and no protected or virtual functions
value	An ordered <code>basic_value</code> . Also: <code>weakly_ordered_value</code> , <code>partially_ordered_value</code>
struct	A <code>basic_value</code> with all public members, no virtuals, no custom assignment
enum	An ordered <code>basic_value</code> with all public values
flag_enum	An ordered <code>basic_value</code> with all public values, and bitwise sets/tests
union	A safe (tagged) union with names (unlike <code>std::variant</code>)

interface	An abstract class having only pure virtual functions
polymorphic_base	A pure polymorphic base type that is not copyable or movable, and whose destructor is either public+virtual or protected+nonvirtual
ordered	A totally ordered type with operator<= that implements strong_ordering . Also: weakly_ordered , partially_ordered
copyable	A type that has copy/move construction/assignment
basic_value	A copyable type that has public default construction and destruction (generated if not user-written) and no protected or virtual functions
value	An ordered basic_value . Also: weakly_ordered_value , partially_ordered_value
struct	A basic_value with all public members, no virtuals, no custom assignment
enum	An ordered basic_value with all public values
flag_enum	An ordered basic_value with all public values, and bitwise sets/tests
union	A safe (tagged) union with names (unlike std::variant)
regex	A CRTE-style compile time regex, but using reflection+generation (Max Sagebaum)
print	Print the reflection as source code at compile time

Optimizing C++ regex

Hana Dusíková: Compile Time Regular Expressions (CTRE)

compile-time-regular-expressions.readthedocs.io/en/latest/

www.compile-time.re

`constexpr + templates`

For compile-time defined patterns

Compile time parsing

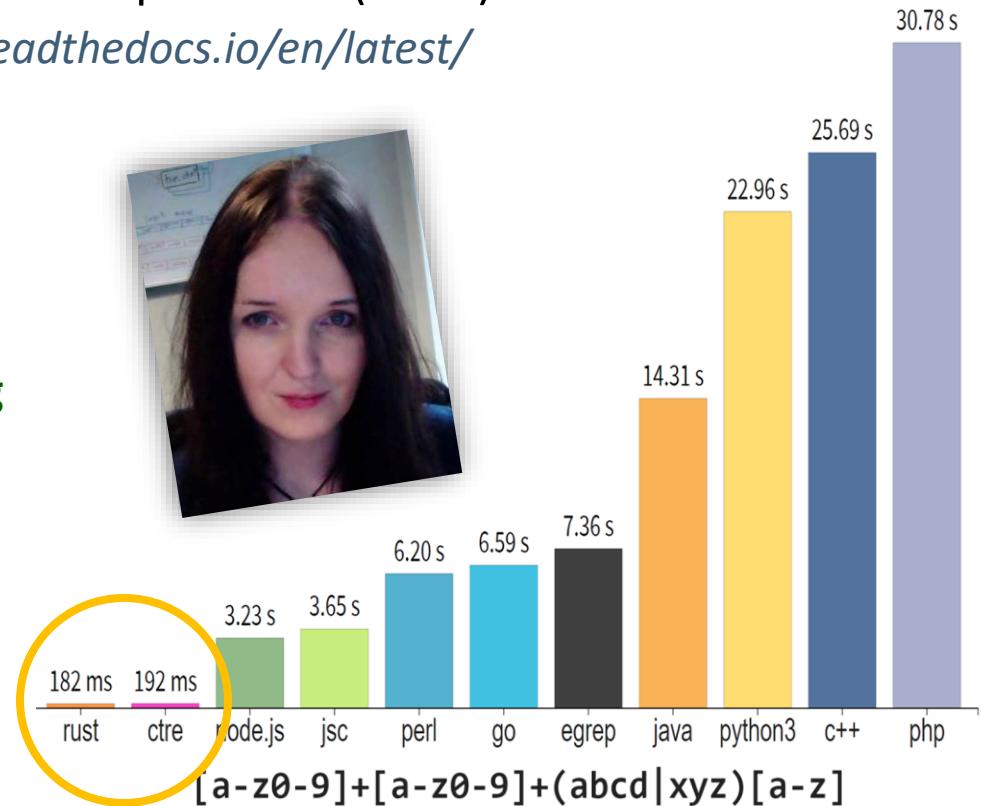
Compile time & run-time matching

Quick regex matching/searching

Structured bindings

DFA without captures

Very efficient assembly



Compile-time regex...



CTRE
(Hana Dusíková, 2017+)

Cppfront @regex
(Max Sagebaum, 2024)

Feature set

Nearly full PCRE (Perl)

Most of PCRE

Parsing

Template stack,
dedicated engine for each regex

Reflection + code gen,
dedicated engine for each regex

Engine

Template classes

Template classes

Compile-time regex...

```
//  
// CTRE template + constexpr code  
  
if (auto m = ctre::match<"ab{0,}bc">(str))  
{  
    ...  
}
```

CTRE

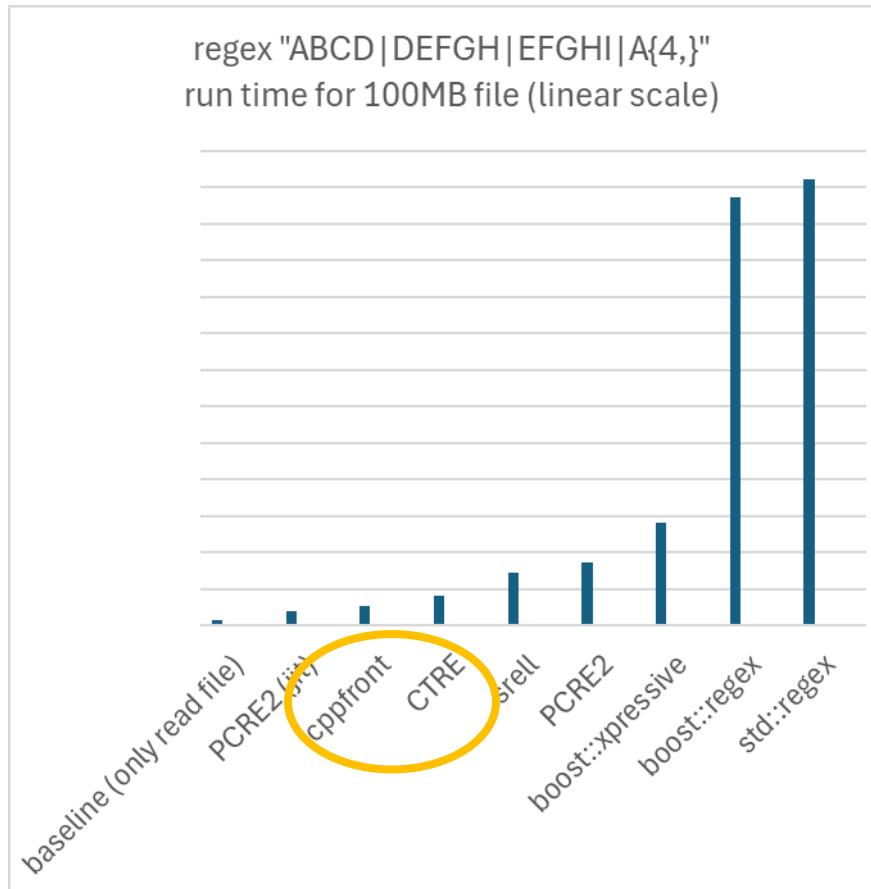
```
my_regexes: @regex type = {  
    r := "ab{0,}bc";  
}  
  
if (my_regexes::r.search(str).matched)  
{  
    ...  
}
```

cppfront

compile-time
reflection + source
generation



Sample **run-time** results (still incomplete)



Sample **compile-time** results (still incomplete)



Haskell Regex POSIX test suite

hackage.haskell.org/package/regex-posix-unittest
wiki.haskell.org/Regex_Posix

One takeaway already: “Another compile step”
does not imply “slower compilation”

When you add new work, also subtract the cost
of the work it replaces (**net cost**)

This is why constexpr code (running a ‘C++
interpreter’ and all!) is often faster than TMP

A pattern: “The compiler helps those who help
themselves” by directly expressing intent

QClass (user code)

Qt moc extensions

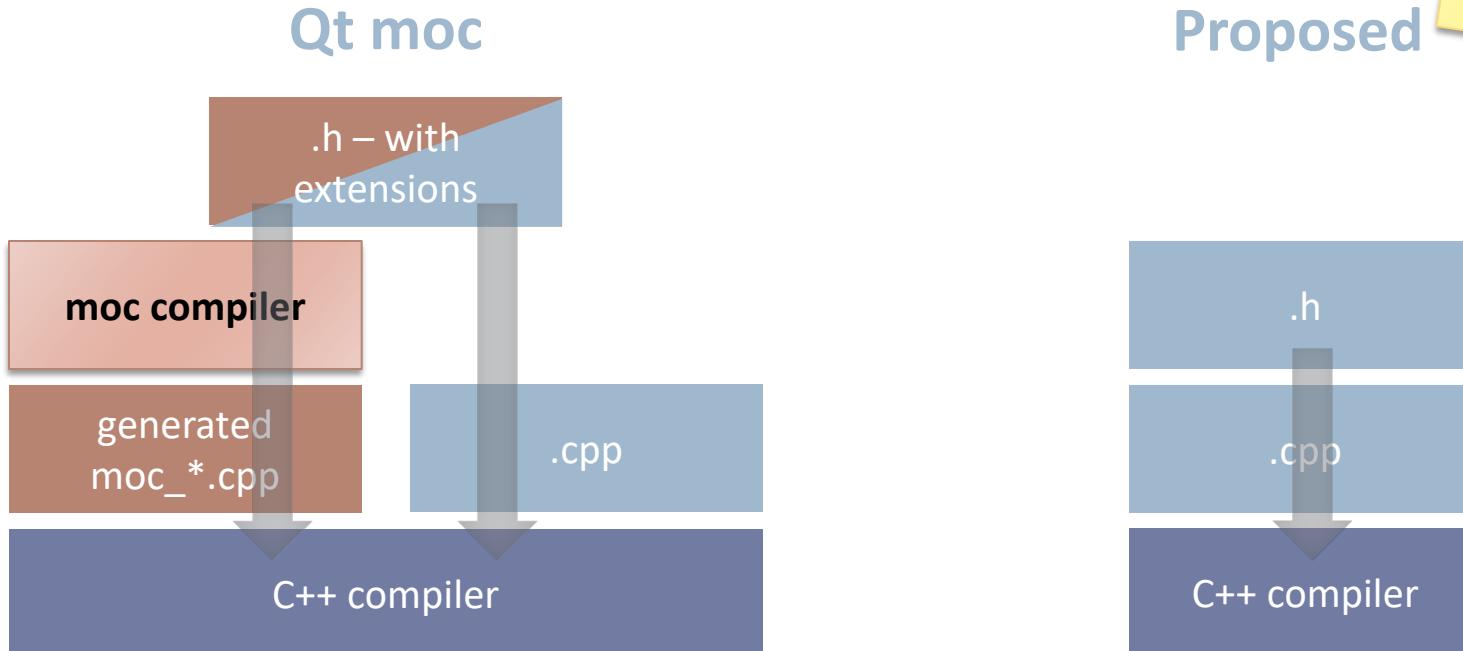
```
class MyClass : public QObject {
    Q_OBJECT
public:
    MyClass( QObject* parent = 0 );
    Q_PROPERTY(int value READ get_value
               WRITE set_value)
    int get_value() const
        { return value; }
    void set_value(int v)
        { value = v; }
private:
    int value;
signals:
    void mySignal();
public slots:
    void mySlot();
};
```

FROM
CPPCON
2017

Proposed (strawman)

```
class(QClass) MyClass {
    property<int> value { }; // default
    signal mySignal();
    slot   mySlot();
};
```

When you can't express it all in C++ code



rt_interface (user code)

COM IDL-style extensions

```
[object,
uuid(a03d1420-b1ec-11d0-8c3a-00c04fc31d2f),
]
interface IFoo : IInspectable {
    [propget]
    HRESULT Get(
        [in] UINT key,
        [out, retval] SomeClass** value
    );
    [propput]
    HRESULT Set(
        [in] UINT key,
        [in] SomeClass* value
    );
};
```

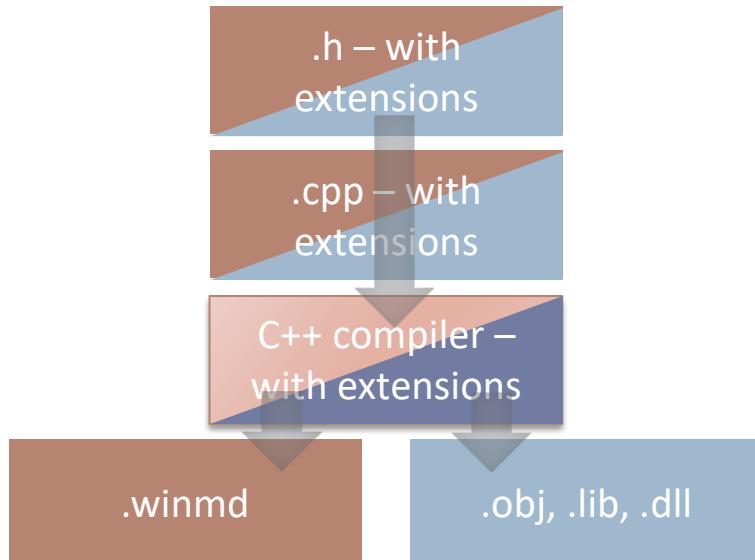
Proposed (strawman)

```
class(rt_interface<
    "a03d1420-b1ec-11d0-8c3a-00c04fc31d2f">)
IFoo {
    property<UINT,SomeClass> value;
};
```

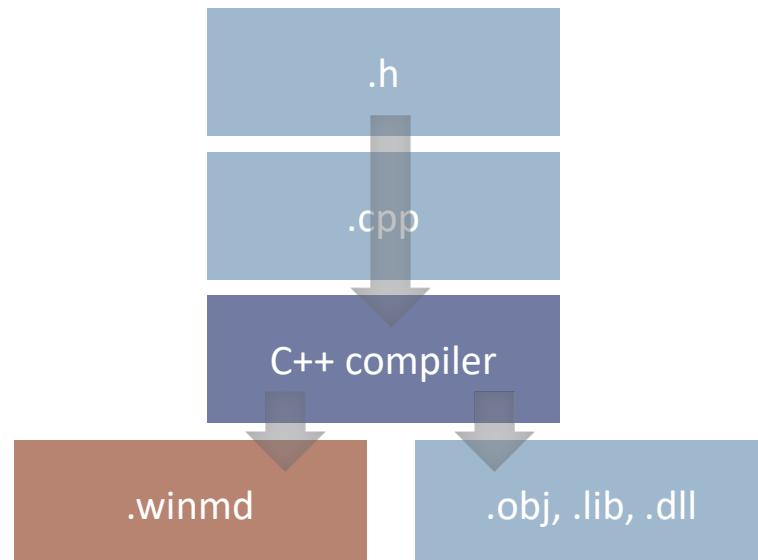
FROM
CPPCON
2017

When you can't express it all in C++ code

C++/CX (for WinRT)

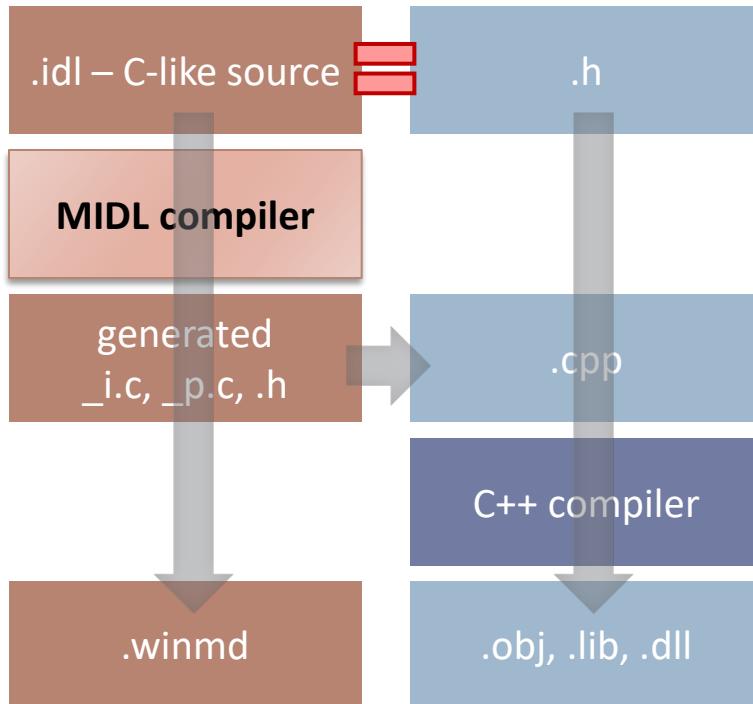


Proposed

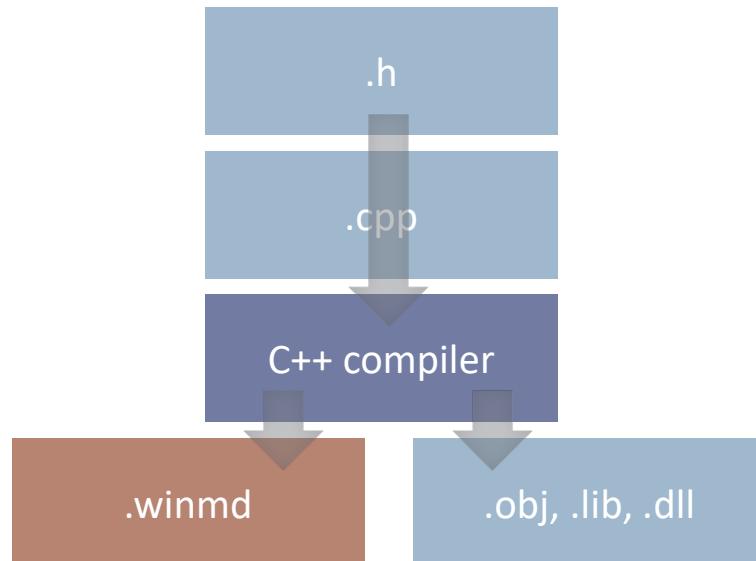


When you can't express it all in C++ code

C++/WinRT IDL (like COM)



Proposed



podio (particle physics data models)

Benedikt Hegner, Axel Naumann

Today (separate YAML script)

ExampleHit :

Description : "Example Hit"

Author : "B. Hegner"

Members:

- double x // x-coordinate
- double y // y-coordinate
- double z // z-coordinate
- double energy // measured

generate: 5 interrelated classes...

X, XCollection, XConst, XData, XObj

how: separate code generator

Proposed (strawman)

```
class(podio::datatype) ExampleHit {  
    string Description = "Example Hit";  
    string Author      = "B. Hegner";  
  
    double x;          // x-coordinate  
    double y;          // y-coordinate  
    double z;          // z-coordinate  
    double energy;    // measured  
};
```

default + enforce: constexpr static strings

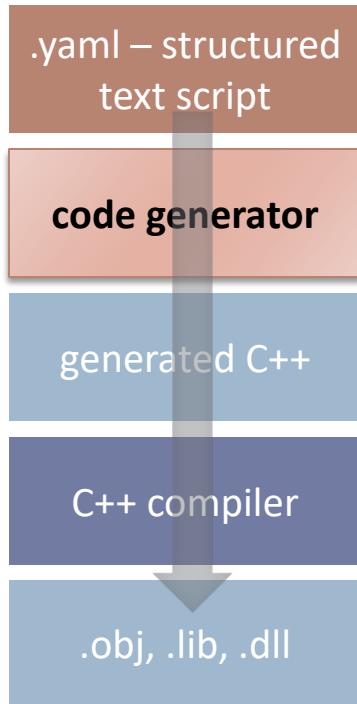
generate: same 5 classes

how: during normal C++ compilation

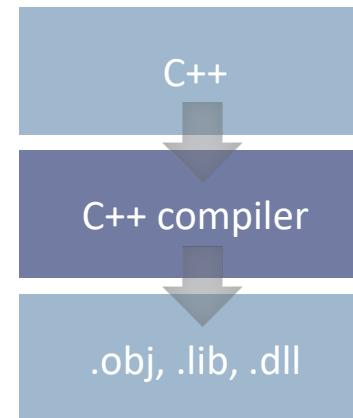
FROM
CPPCON
2017

When you can't express it all in C++ code

podio YAML



Proposed



Example: instrumented_vector

```
template <template<typename... T> class entity>
constexpr auto create_instrumented_template() -> meta::info;

template <typename... Params>
using my_vector = [: create_instrumented_template<std::vector>() :];
```



Generated code

```
template<typename... Params>
class my_vector {
    std::vector<Params...> impl; // ADDED
    // ...
    reference operator[](size_type pos) { // ECHOED
        thread_local uint64_t count_calls = 0; // ADDED
        ++count_calls; // ADDED
        return impl[pos]; // WRAPPED
    }
    // ... etc. for all the other functions
};
```

Dispatch from the Land of Dee...

tour.dLang.org/tour/en/dub/pegged

```
mixin(grammar(`  
Arithmetic:  
    Term      < Factor (Add / Sub)*  
    Add       < "+" Factor  
    Sub       < "-" Factor  
    Factor    < Primary (Mul / Div)*  
    Mul      < "*" Primary  
    Div       < "/" Primary  
    Primary   < Parens / Neg / Pos / Number  
    Parens   < "(" Term ")"  
    Neg      < "-" Primary  
    Pos      < "+" Primary  
    Number   < ~([0-9]+)  
    Variable <- identifier  
`));
```

```
void main() {  
    // Parsing at compile-time:  
    enum parseTree1 = Arithmetic("1 + 2 - (3*x-5)*6");  
  
    pragma(msg, parseTree1.matches);  
    assert(parseTree1.matches == ["1", "+", "2", "-",  
        "(", "3", "*", "x", "-", "5", ")","*", "6"]);  
    writeln(parseTree1);  
  
    // And at runtime too:  
    auto parseTree2 = Arithmetic(" 0 + 123 - 456 ");  
    assert(parseTree2.matches  
        == ["0", "+", "123", "-", "456"]);  
}
```

Welcome to C++'s next decade!

WELCOME TO C++'S NEXT DECADE

Conjecture: Reflection+generation will dominate the next decade of C++

Making easier things that are difficult today (e.g., TMP, expression templates)
Making possible things that are infeasible today (e.g., generative programming)

Expect multiple phases... the first phase is “now in sight”

A requirements roadmap: What will we need?

“All information in the source code must be reflectable”

If the programmer can know it, they’ll want to use it

Examples: Attributes, defaults (meaningful whitespace!)

(note: in the language; likely not preprocessor)

```
class Foo {  
    int func1();  
public:  
    void func2(int);  
};  
  
struct Foo {  
    int func1();  
public:  
    void func2(int);  
};  
  
class(interface) Foo {  
    int func1();  
public:  
    void func2(int);  
};
```

A requirements roadmap: What will we need?

“All information in the source code must be reflectable”

If the programmer can know it, they’ll want to use it

Examples: Attributes, defaults (meaningful whitespace!)

(note: in the language; likely not preprocessor)

“Anything that can be written in source code must be generatable”

If the programmer can write it by hand, they’ll want to write it by generated code

Examples: Types, free functions, specializations of std:: templates

```
class(class) Foo {  
    int func1();  
public:  
    void func2(int);  
};
```

```
class(struct) Foo {  
    int func1();  
public:  
    void func2(int);  
};
```

```
class(interface) Foo {  
    int func1();  
public:  
    void func2(int);  
};
```

A requirements roadmap: What will we need? (2)

“All source code must be visible, whether hand-written or generated”

The final code is the only source of truth.

Entry level: We need to see what we got ⇒ pretty-print generated code

Then tooling:

- Debugging (e.g., step into generated code)

- Visualizing (e.g., expand/collapse generated code)

...

Generalizing: **“All output (text + binary) must be possible at compile time”**

Example: .winmd output files

Output files for other tools

...

Risks

Good: Standardizing some now, adding more later

But requires:

A “North Star”: We have to know the **end use cases we’re aiming for**

Design guardrails: We have to know we’re **not going off on a side track**

Relatively easy for constexpr and GPUs: “support the language feature, not a divergent special-purpose extension” such as a different kind of loop

Risk of bottom-up design is that we may end up with overlapping pieces that don’t fill in the whole picture

Suggested aim: P0707 metafunctions, Andrei’s instrumented_vector,
reflect+regenerate any type (“identity”)

Learn from related experience (C#, D, Lock3, cppfront)

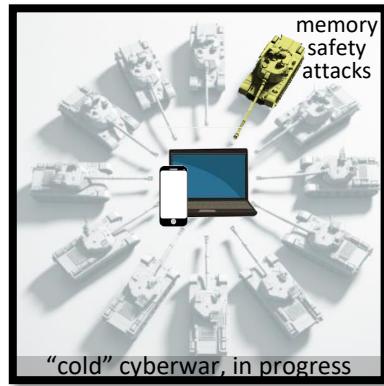


1



2

Safety



3

Simplicity



Terms (per ISO/IEC 23643:2020)

Software **security** (or “cybersecurity” or similar)

making software ~~able~~ to protect its assets from a malicious attacker

examples: securing power grids, hospitals, banks, personal data, secrets, ...

Software **safety** (or “life safety” or similar)

making software ~~safe~~ from unacceptable risk of causing unintended harm to humans, property, or the environment

examples: hospital equipment, autonomous vehicles/weapons

Programming language **safety** (incl. memory safety)

static and dynamic guarantees about program correctness

helps both the others – and increases quality generally



Consider **initialization** again...

Q: Is this “righteous, most excellent” code?

```
std::array<int,1024> a; // A - uninitialized?  
fill(a); // B - call a function that sets a's values
```



For line A:

If uninitialized, this program is actually ideal

... but analyzers/humans can't prove that `fill(a)` fills a

If initialized, we know the dummy writes aren't needed...

... but optimizers are terrible at removing the “dead writes”

... for the same reason: they can't prove that `fill(a)` fills a

Usual plea: “But it's obvious! Compiler writers, just try harder!”

Usual answer: “Fine, you try to look through an opaque function call!”

Initialization safety: Having our cake *and* eating it

Both “force initialize at declaration” (e.g., C++ Core Guidelines)
and “fill with pattern” approaches **jam in dummy values**



What we really want is “initialize before use”

C#, Ada, and other language have “definite initialization” rules for local vars

Experience: easy to specify, **easy to use by mainstream developers**

What I've implemented in Cpp2 (github.com/hsutter/cppfront):

All local vars (all types!) uninitialized by default \Rightarrow **fast by default**

Guaranteed init before first use on any branch path \Rightarrow **correct always**

Via either direct construction or passing to a fill function's “out” parameter

\Rightarrow **fully composable init**, generalized delegating constructors

Example (Cpp2 syntax, will propose for ISO C++ syntax too)

Using a fundamental type, for example: **int**



```
a: int;                      // allocates space, no initialization
// std::cout << a;          // illegal: can't use-before-init!
a = 5;                        // construction => real initialization!
std::cout << a;              // prints 5
```

Using any type, for example: **std::vector<std::string>**



```
b: vector<string>;        // allocates space, no initialization
// std::cout << b;          // illegal: can't use-before-init!
b = ("xyzzy", "plugh");    // construction => real initialization!
std::cout << b.size();     // prints "2"
```

```
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 = {
    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.size() >= count,
  "value must contain at least count chars"]]
= {
    x = value.substr(0, count);
}
```

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

Bounds safety: Low-hanging fruit

What I've already implemented in Cpp2 as proof of concept: **For every `a[b]`** where `a` is a contiguous container (incl. `std::size(a)`) and `b` is integral...

Inject a call-site bounds check for `0 <= b && b < std::size(a)`

Violations reported via normal **contract violation handling** \Rightarrow customizable

Results so far:

Seamless: Works perfectly for **all STL implementations unchanged**

Also works for **C arrays** (before they decay, while their names are in scope)

Also works for **most non-std containers** (`std::size` and `operator[]` are widely used...
“C++ code contains a lot of information!”)

Customization enables easy integration into existing projects' error/logging

Proposal for future ISO C++ code: “**Enable ‘bounds’ Profile and recompile**”

Example: Looks right, ship it?

```
// int          a[] = {1, 2, 3};    // uncomment either one: C array
// std::vector<int> a   = {1, 2, 3};  // or STL container (unmodified)
print(a[1]); print(a[2]); print(a[3]); // line 7
```

```
/mnt/c/demo $./cppfront.sh demo.cpp2 gcc
demo.cpp2 ... ok (mixed Cpp1/Cpp2, Cpp2 code passes safety checks)
C:\demo>cppf...
demo.cpp2... g++ (GCC) 14.2.1 20240801 (Red Hat 14.2.1-1)
Copyright (C) 2024 Free Software Foundation, Inc.
Microsoft (R) This is free software; see the source for copying conditions. There is NO
Copyright (C) warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

demo.cpp      /mnt/c/demo $./a.out
              2
C:\demo>demo 3
2
3
demo.cpp2(7) int main(): Bounds safety violation: out of bounds access atte
mpt detected - attempted access at index 3, [min,max] range is [0,2]
demo.cpp2(7) int __caec1_main(void): Bounds safety violation: out of
bounds access attempt detected - attempted access at index 3, [min,
max] range is [0,2]
```

Risks



Good: Providing safety guarantees

But requires:

Confidence: We have to know the **rules actually work**

Adoptability: We have to know **end user code can adopt it**, incl. in existing code

Impact: We have to know it will help **as much code as possible**, incl. existing code

Risk of ad-hoc safety design is that we may end up with improvements that don't work adoptably at scale and/or for existing code

Suggested aim: embrace existing **known-good** safety rules + **no heavy/viral annotation** + articulate what % of benefit can be had in existing code **without manual code changes** (just a recompile), not just new/updated code

simplification through generalization

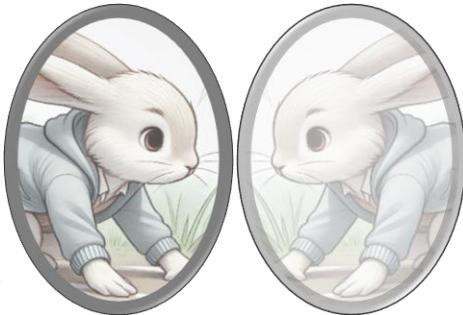
key: enable programmer to directly express intent

- ⇒ elevate coding patterns to declare “what” vs “how”
- ⇒ use the intent already in the code, instead of “how” annotations

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

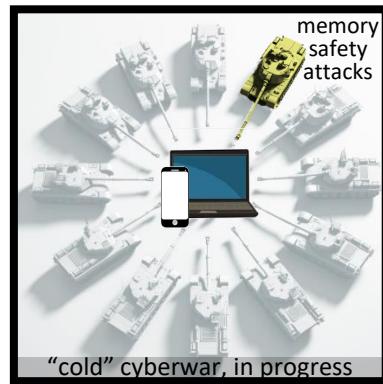
“Say 10% of the size of C++ ... Most of the simplification would come from generalization.”
— Bjarne Stroustrup (ACM HOPL-III, 2007)

1



2

Safety



3

Simplicity



Major advances are on track

`std::execution` (*concurrency and parallelism*)



Type and memory safety improvements

- target: **parity** with other modern languages

Reflection + code generation (aka ‘injection’/...)

- part of our **sea change to compile-time programming**
- arguably **most-impactful** feature ever added
- I expect will **dominate** our next decade

Contracts

Peering forward: C++'s next decade

Herb Sutter

