Reflection

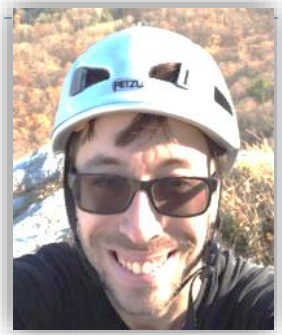
C++'s Decade-Defining Rocket Engine

HERB SUTTER

CITADEL | Securities

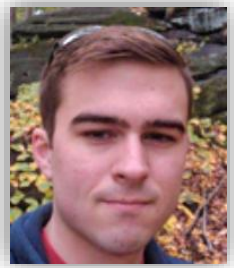
Reflection paper authors, implementers, contributors, improvers, debuggers, …

Too many to list!



Dan Katz
(Bloomberg, Adobe)

Many reviewers
(Thanks!)



Wyatt Childers
(Lock3, EDG)

Barry Revzin
(Jump Trading)

Andrew Sutton
(U. Akron, Lock3)

Matúš Chochlík
(GlobalLogic)

Axel Naumann
(CERN)

David Sankel
(Bloomberg)

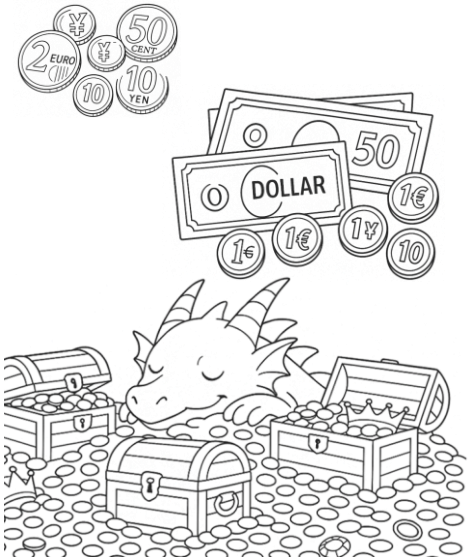
Daveed Vandevoorde
(EDG)

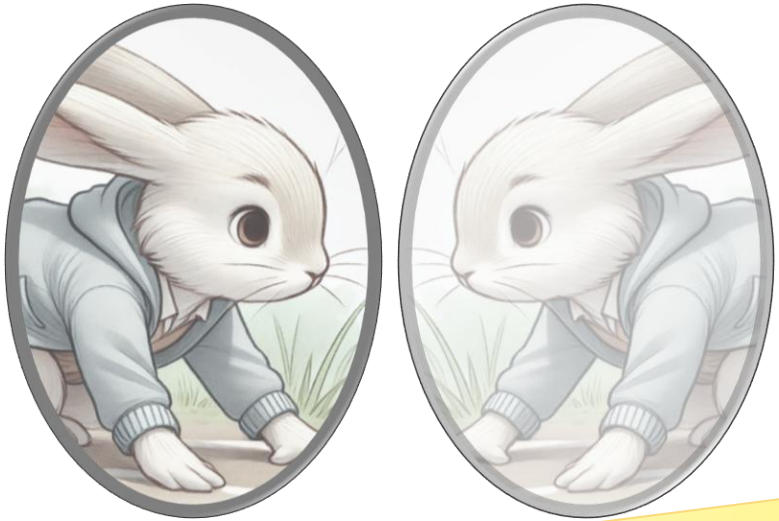
Louis Dionne
(Amazon, Apple)

Antony Polukhin
(Yandex)

Roadmap

Overview	A few good tools
Today	Some is good …
Tomorrow	… more is better …
Beyond	… and “everything” is just about right





short definition

the program can **see** itself

and **generate** itself

… hence, **meta**programs

static ⇒ zero runtime overhead

(this isn't Java or C#)

to use some information at run time, just store it:

`define_static_array`,

`define_static_string`

It's all C++ code

godbolt.org/z/WeaKzWc56

```
struct MyOpts {
    string fname;
    int    count;
};

auto main() -> int
{
    template for( auto constexpr m: data_members_of(^^MyOpts) ) {
        println( "member named: {}", display_string_of(m) );
    }
}
```

```
Program stdout
member named: fname
member named: count
```

We are about to see a huge explosion in C++ compile-time code

And it will make C++ compilation *net faster*

by replacing slower things we do today

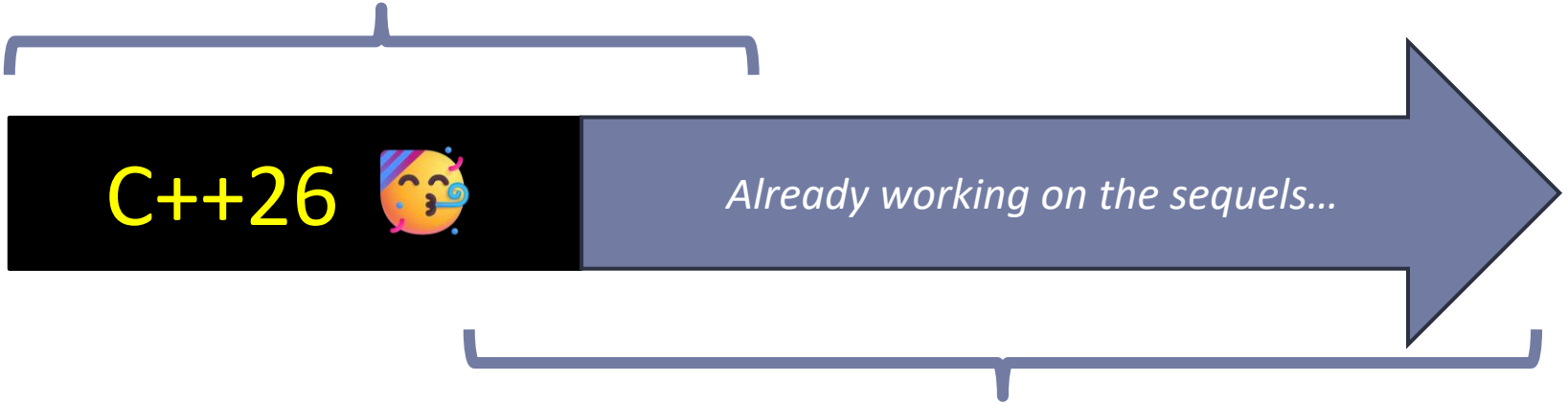
(also: more debuggable, testable, …)

TMP $\rightarrow$ **“just C++ code”** (constexpr is already faster)

extra side compilers / build steps $\rightarrow$ **“in the C++ compiler”**

Compilers we'll use for examples

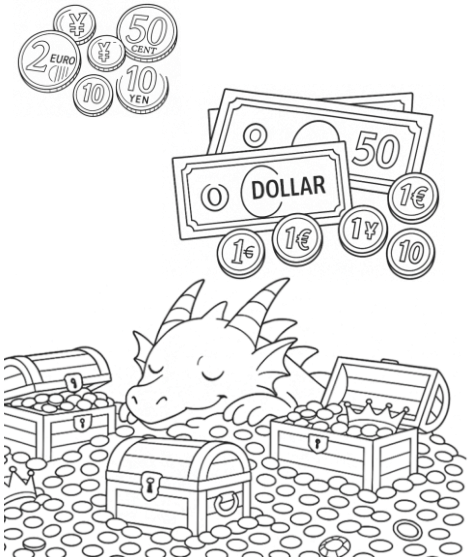
Existing C++26++ reflection impls:
Dan Katz's (Clang)
Daveed Vandevoorde's (EDG)



My cppfront reflection implementation
(lowers to 100% pure C++)

Roadmap

Overview	A few good tools
Today	Some is good …
Tomorrow	… more is better …
Beyond	… and “everything” is just about right



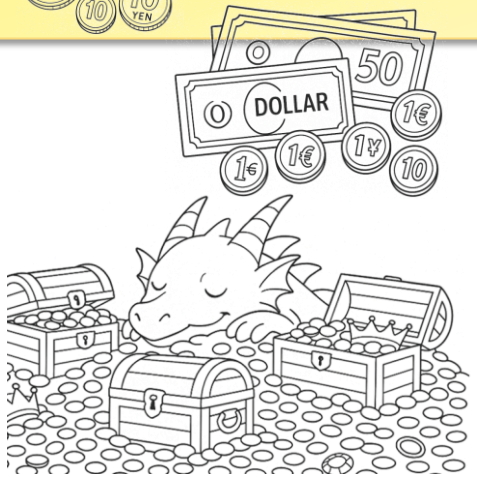
Roadmap

Overview A few good tools

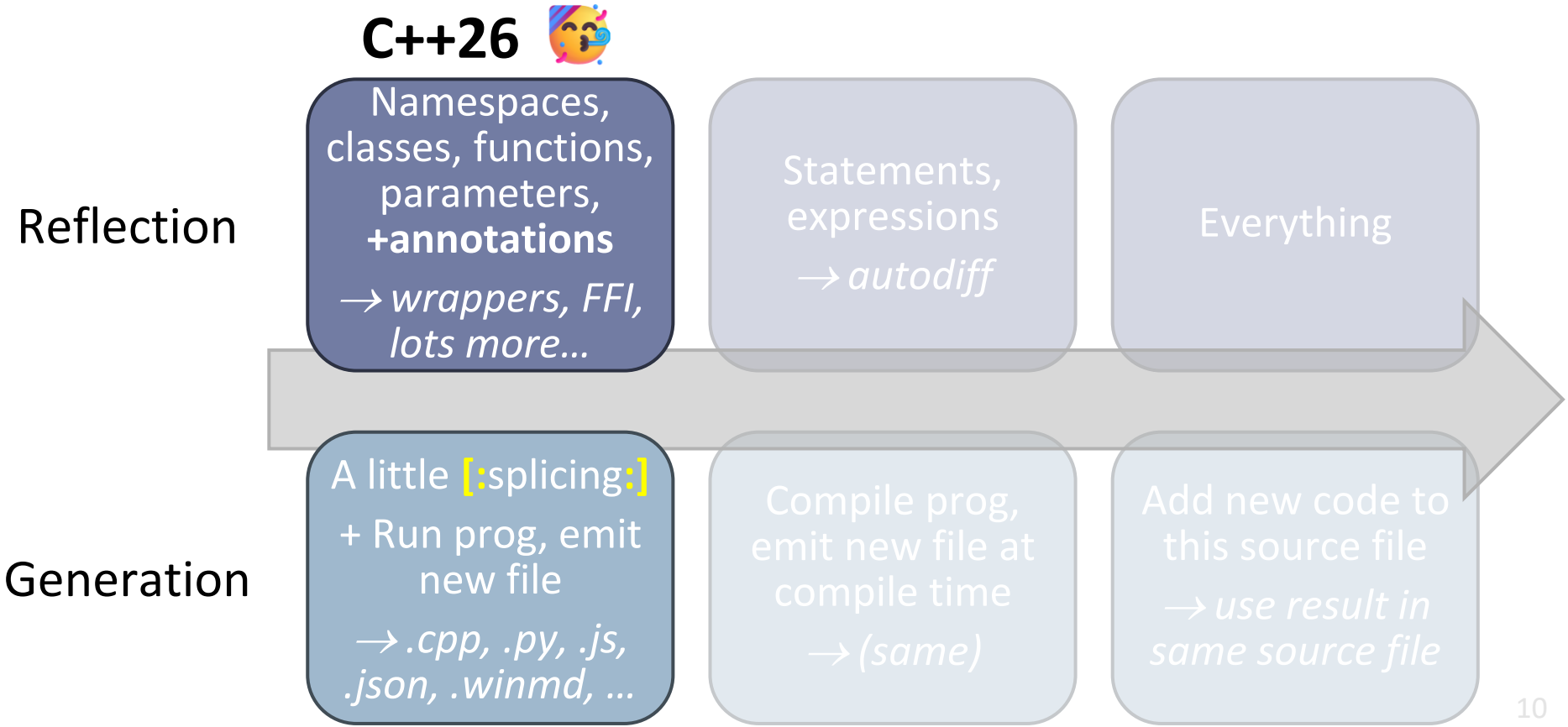
Today Some is good …

Tomorrow … more is better …

Beyond … and “everything” is just about right



The reflection (r)evolution: Roadmap sketch



Warmup: #embed splice

godbolt.org/z/3jzehW1bv

Thanks to Dan Katz!

```
{
    "magic": "XYZZY",
    "inner": { "word": "PLUGH", "number": 1066 }
}
```

```
constexpr const char json_data[] = {
    #embed "test.json"
    , 0
};

auto main() -> int {
    constexpr auto v = [: parse_json(json_data) :];
                     // or: json_to_object<json_data>
    println("v.magic: {}\nv.inner.word: {}\nv.inner.number: {}",
            v.magic, v.inner.word, v.inner.number);
}
```

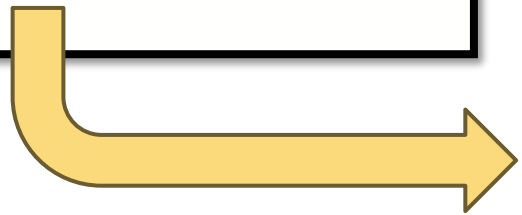
```
Program stdout
v.magic: XYZZY
v.inner.word: PLUGH
v.inner.number: 1066
```

Simple cmdline option parser

godbolt.org/z/P6jeKK3of

Thanks to Matúš Chochlík!

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

auto main(int argc, char *argv[]) -> int
{
    MyOpts opts = parse_options<MyOpts>({ argv, argv+argc});
    println("opts.fname is \"{}\"", opts.fname);
    println("opts.count is {}",     opts.count);
}
```



```
--count 42 --fname

Program returned: 0
Program stdout
opts.fname is "input.txt"
opts.count is 42
```

C++20 ad-hoc polymorphism *godbolt.org/z/MEPvEMbqP*

```
auto v = std::vector<Drawable>{};
v.emplace_back(Sprite{});
v.emplace_back(Icon{});
v.emplace_back(Button{});

auto where = Coordinate{3, 5};
auto sum = 0;
for (auto& d : v) {
    sum += d.Draw(where);
}
std::println( "Sum of Draws: {}\n", sum );

// Copy/assign works, just like std::function
auto d1 = Drawable{v[0]};
auto d2 = Drawable{};
d2 = v[1];
std::println( "d1: {}, d2: {}\n",
              d1.Draw(where), d2.Draw(where) );
```

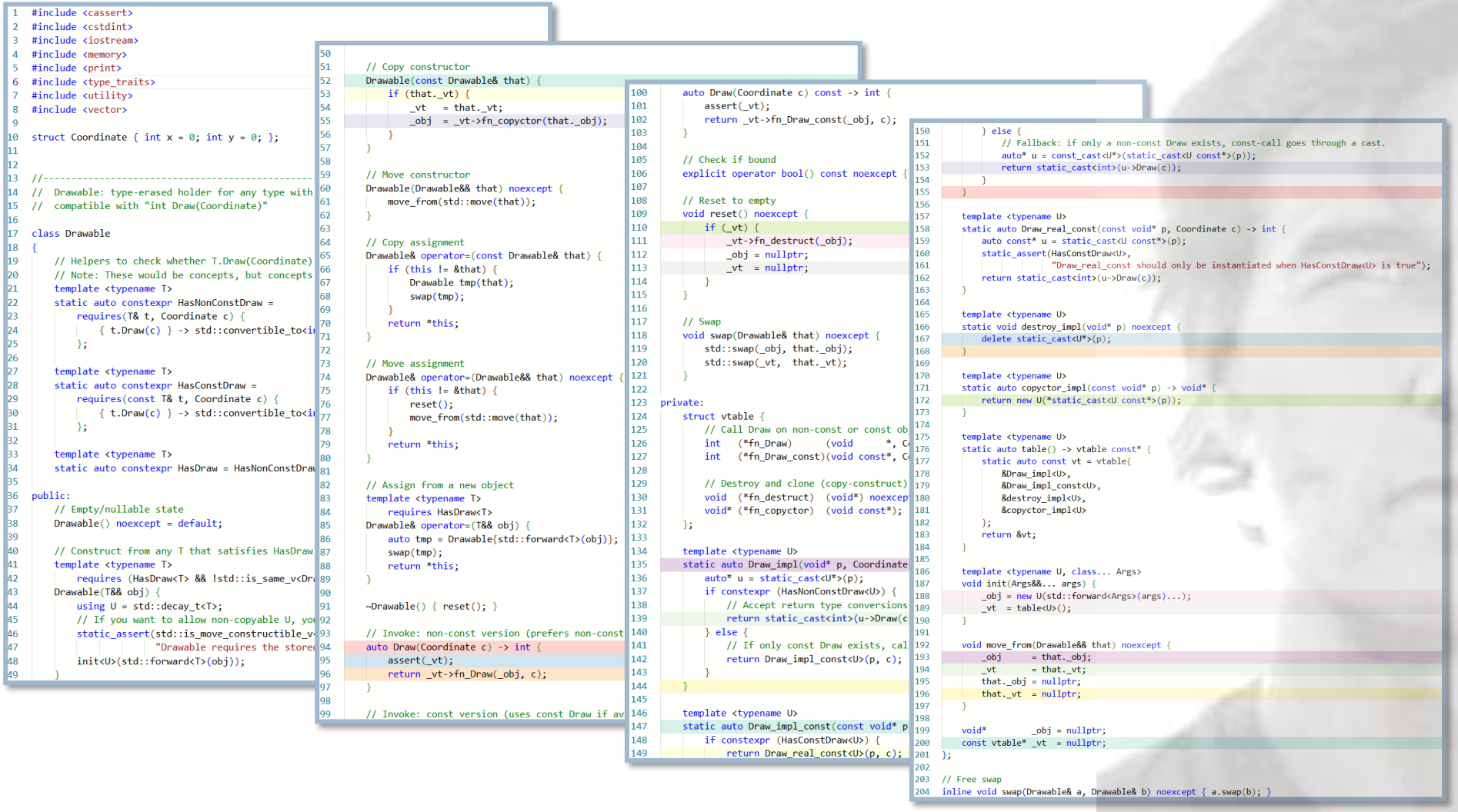
Mission: Drawable

Wrap any object having (approx)

```
int Draw(Coordinate)
```

~200 LOC per case

```
Program returned: 0
  Sprite::Draw at (3,5)
  Icon::Draw at (3,5) [const]
  Button::Draw at (3,5)
  Sum of Draws: 51

  Icon::Draw at (3,5) [const]
  Sprite::Draw at (3,5)
  d1: 1, d2: 42
```



```cpp
#include <cassert>
#include <cstdint>
#include <iostream>
#include <memory>
#include <print>
#include <type_traits>
#include <utility>
#include <vector>

struct Coordinate { int x = 0; int y = 0; };


//-----------------------------------------------
//  Drawable: type-erased holder for any type with
//  compatible with "int Draw(Coordinate)"

class Drawable
{
    // Helpers to check whether T.Draw(Coordinate)
    // Note: These would be concepts, but concepts
    template <typename T>
    static auto constexpr HasNonConstDraw =
        requires(T& t, Coordinate c) {
            { t.Draw(c) } -> std::convertible_to<i
        };

    template <typename T>
    static auto constexpr HasConstDraw =
        requires(const T& t, Coordinate c) {
            { t.Draw(c) } -> std::convertible_to<i
        };

    template <typename T>
    static auto constexpr HasDraw = HasNonConstDra

public:
    // Empty/nullable state
    Drawable() noexcept = default;

    // Construct from any T that satisfies HasDraw
    template <typename T>
        requires (HasDraw<T> && !std::is_same_v<Dr
    Drawable(T&& obj) {
        using U = std::decay_t<T>;
        // If you want to allow non-copyable U, yo
        static_assert(std::is_move_constructible_v
                      "Drawable requires the store
        init<U>(std::forward<T>(obj));
    }
```

```cpp

    // Copy constructor
    Drawable(const Drawable& that) {
        if (that._vt) {
            _vt   = that._vt;
            _obj  = _vt->fn_copyctor(that._obj);
        }
    }

    // Move constructor
    Drawable(Drawable&& that) noexcept {
        move_from(std::move(that));
    }

    // Copy assignment
    Drawable& operator=(const Drawable& that) {
        if (this != &that) {
            Drawable tmp(that);
            swap(tmp);
        }
        return *this;
    }

    // Move assignment
    Drawable& operator=(Drawable&& that) noexcept {
        if (this != &that) {
            reset();
            move_from(std::move(that));
        }
        return *this;
    }

    // Assign from a new object
    template <typename T>
        requires HasDraw<T>
    Drawable& operator=(T&& obj) {
        auto tmp = Drawable{std::forward<T>(obj)};
        swap(tmp);
        return *this;
    }

    ~Drawable() { reset(); }

    // Invoke: non-const version (prefers non-const
    auto Draw(Coordinate c) -> int {
        assert(_vt);
        return _vt->fn_Draw(_obj, c);
    }

    // Invoke: const version (uses const Draw if av
```

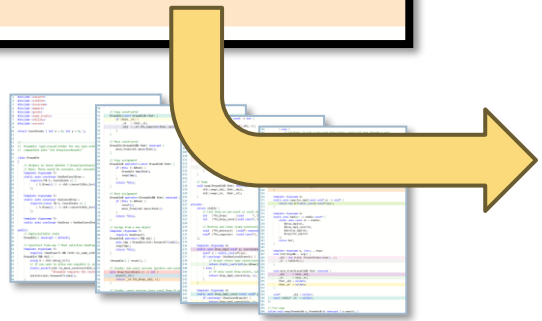
```cpp
    auto Draw(Coordinate c) const -> int {
        assert(_vt);
        return _vt->fn_Draw_const(_obj, c);
    }

    // Check if bound
    explicit operator bool() const noexcept {

    // Reset to empty
    void reset() noexcept {
        if (_vt) {
            _vt->fn_destruct(_obj);
            _obj = nullptr;
            _vt  = nullptr;
        }
    }

    // Swap
    void swap(Drawable& that) noexcept {
        std::swap(_obj, that._obj);
        std::swap(_vt,  that._vt);
    }

private:
    struct vtable {
        // Call Draw on non-const or const ob
        int   (*fn_Draw)       (void       *, C
        int   (*fn_Draw_const)(void const*, C

        // Destroy and clone (copy-construct)
        void  (*fn_destruct)  (void*) noexcep
        void* (*fn_copyctor)  (void const*);
    };

    template <typename U>
    static auto Draw_impl(void* p, Coordinate
        auto* u = static_cast<U*>(p);
        if constexpr (HasNonConstDraw<U>) {
            // Accept return type conversions
            return static_cast<int>(u->Draw(c
        } else {
            // If only const Draw exists, cal
            return Draw_impl_const<U>(p, c);
        }
    }

    template <typename U>
    static auto Draw_impl_const(const void* p
        if constexpr (HasConstDraw<U>) {
            return Draw_real_const<U>(p, c);
```

```cpp
        } else {
            // Fallback: if only a non-const Draw exists, const-call goes through a cast.
            auto* u = const_cast<U*>(static_cast<U const*>(p));
            return static_cast<int>(u->Draw(c));
        }
    }

    template <typename U>
    static auto Draw_real_const(const void* p, Coordinate c) -> int {
        auto const* u = static_cast<U const*>(p);
        static_assert(HasConstDraw<U>,
                      "Draw_real_const should only be instantiated when HasConstDraw<U> is true");
        return static_cast<int>(u->Draw(c));
    }

    template <typename U>
    static void destroy_impl(void* p) noexcept {
        delete static_cast<U*>(p);
    }

    template <typename U>
    static auto copyctor_impl(const void* p) -> void* {
        return new U(*static_cast<U const*>(p));
    }

    template <typename U>
    static auto table() -> vtable const* {
        static auto const vt = vtable{
            &Draw_impl<U>,
            &Draw_impl_const<U>,
            &destroy_impl<U>,
            &copyctor_impl<U>
        };
        return &vt;
    }

    template <typename U, class... Args>
    void init(Args&&... args) {
        _obj = new U(std::forward<Args>(args)...);
        _vt  = table<U>();
    }

    void move_from(Drawable&& that) noexcept {
        _obj       = that._obj;
        _vt        = that._vt;
        that._obj = nullptr;
        that._vt  = nullptr;
    }

    void*          _obj = nullptr;
    const vtable* _vt  = nullptr;
};

// Free swap
inline void swap(Drawable& a, Drawable& b) noexcept { a.swap(b); }
```

Reflection ad-hoc **poly<T>**

godbolt.org/z/aP4dKfqjo

```
struct Coordinate { int x = 0; int y = 0; };

namespace __proto {
    class Drawable {
        int Draw( Coordinate );
    };
}

auto main() -> int {
    std::cout << poly< __proto::Drawable >();
}
```



~300 LOC forever

```
Program returned: 0
   Sprite::Draw at (3,5)
   Icon::Draw at (3,5) [const]
   Button::Draw at (3,5)
   Sum of Draws: 51

   Icon::Draw at (3,5) [const]
   Sprite::Draw at (3,5)
   d1: 1, d2: 42
```

P0707… from 2017 to present

Metaclasses

Document Number:	**P0707 R0**
Date:	2017-06-18
Reply-to:	Herb Sutter (hsutter@microsoft.com)
Audience:	SG7

Declarative class authoring using `consteval` functions + reflection + generation (aka: Metaclasses for generative C++)

Document Number:	**P0707 R5**
Date:	2024-10-12
Reply-to:	Herb Sutter
Audience:	SG7, EWG

Quick refresher example

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

=

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

Note: Slide updated since the talk.

Rationale: See cppfront commit **f7a2af8**

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

“interface” metafunction *godbolt.org/z/Y75Wr3nGe*

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

auto main() -> int {
    std::cout << interface(^^__proto::Widget);
}
```

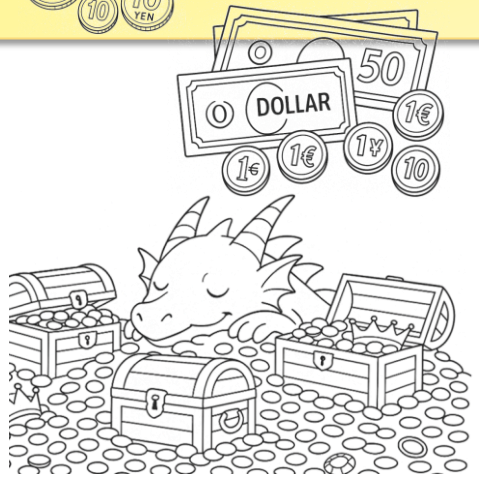
Note: Slide updated since the talk.

Rationale: See cppfront commit **f7a2af8**

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
class Widget {
public:
    virtual int f(  ) =0;
    virtual void g( std::string ) =0;
    virtual ~Widget() = default;
    Widget() = default;
protected:
    Widget(Widget const&) = default;
    void operator=(Widget const&) = default;
};
```

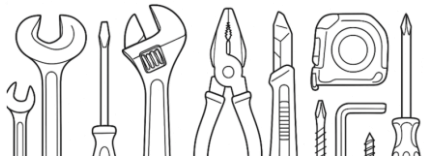
Roadmap

Overview	A few good tools
Today	Some is good …
Tomorrow	… more is better …
Beyond	… and “everything” is just about right



Roadmap

Overview A few good tools

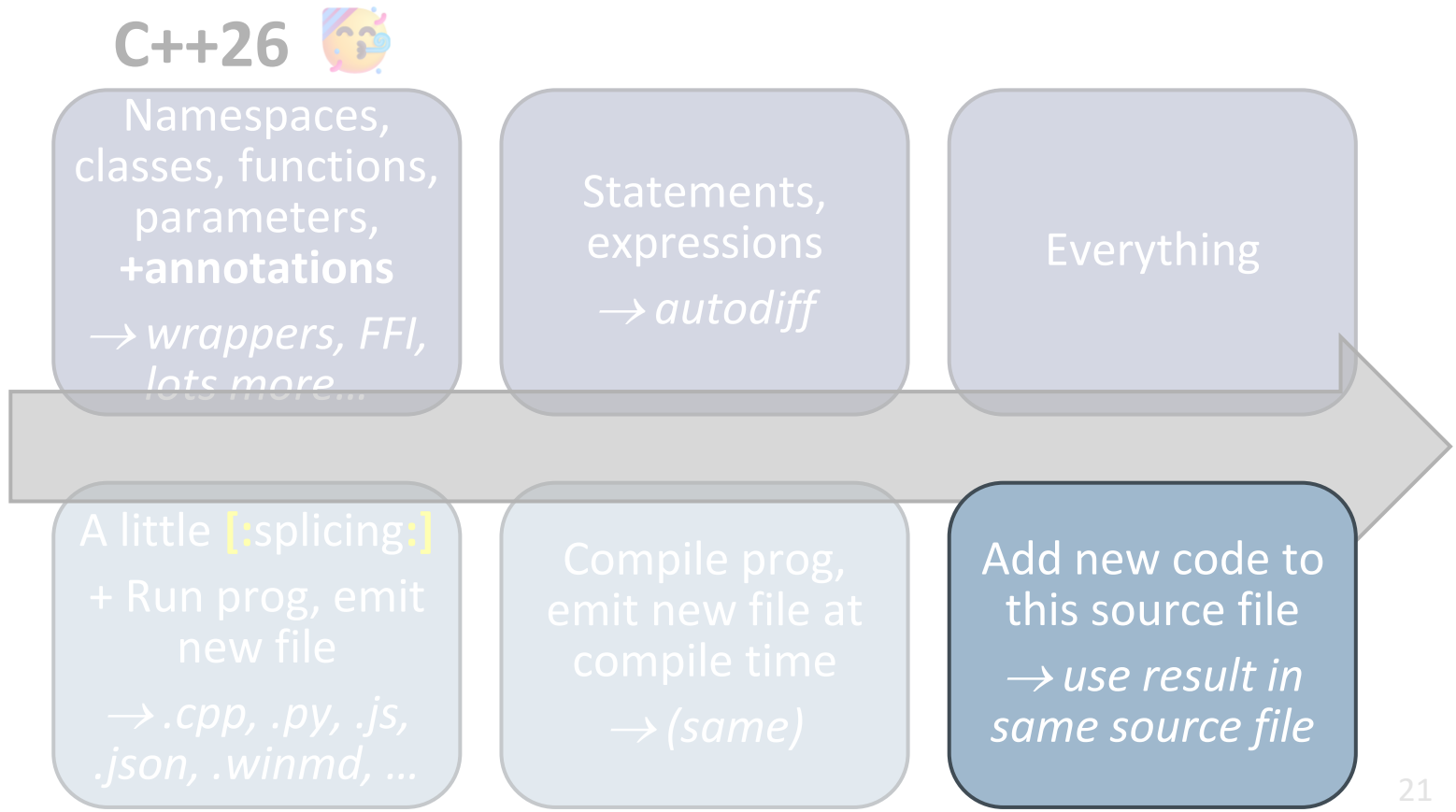


Today Some is good …

Tomorrow … more is better …

Beyond … and “everything” is just about right

The reflection (r)evolution: Roadmap sketch



“interface” now in EDG… *godbolt.org/z/7jv9aj8Mh*

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

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

auto main() -> int {
    auto w = std::unique_ptr<Widget>{ std::make_unique<MyWidget>() };
    std::cout << w->f();     // ok, good
    w->g( "\nxyzzy" );       // ok, good
    // w->h();               // error, good
}
```

```
// P0707 proposed sugar
class(interface) Widget {
  int  f();
  void g(std::string);
};
```

```
// As implemented in cppfront
// (Cpp2 alt syntax → 100% C++)
Widget: @interface type = {
    f: (this) -> int;
    g: (this, s: std::string);
}
```

```
Program stdout
42
xyzzy
```

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 `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`)

`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

`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 op… `strong_ordering`. Also: we…
`copyable`	A type that has copy/move co…
`basic_value`	A `copyable` type that has pub… (generated if not user-written)
`value`	An `ordered basic_value`. Al… `partially_ordered_value`
`struct`	A `basic_value` with all public…
`enum`	An `ordered basic_value` wit…
`flag_enum`	An `ordered basic_value` with…
`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

Prediction: "class(M)" will become pervasive

… and we'll never need =default or =delete to control generated functions ever again

because every M is a Word of Power = a collection of defaults, generated functions, and constraints

From P0707R0 (2017) Abstract

Document Number:	**P0707 R0**
Date:	2017-06-18

(major, this paper) Provide a new abstraction authoring mechanism so programmers can write new kinds of user-defined abstractions that encapsulate behavior. In current C++, the function and the `class` are the two mechanisms that encapsulate user-defined behavior. In this paper, `$class` metaclasses enable defining categories of `class`es that have common defaults and generated functions, and formally expand C++'s type abstraction vocabulary beyond `class`/`struct`/`union`/`enum`.

Also, §3 includes a set of common metaclasses, and proposes that several are common enough to belong in `std::`. Each subsection of §3 is equivalent to a significant "language feature" that would otherwise require its own EWG paper and be wired into the language, but here can be expressed instead as just a (usually tiny) library that can go through LEWG. For example, this paper begins by demonstrating how to implement Java/C# `interface` as a 10-line C++ `std::` metaclass – with the same expressiveness, elegance, and efficiency of the built-in feature in such languages, where it is specified as ~20 pages of text.

Simplifying C++ evolution

Reflection reduces need for certain kinds of new language features

By making it possible to express them **well** as compile-time libraries

"Well" = equivalent usability (to write and debug), flexibility, performance

Benefits of compile-time libraries vs. language features:

- 🩹 Code is testable (and fixable)
- 🚚 Code is portable (works on all compilers right away… once we have reflection)
- 🚤 Code is faster to deliver (GitHub, package managers, …)
- ✂️ Code is easier to customize (adapt, fork, …)

A “starter” list of ideas…

Think outside the box…

- generate a **JSON** serializer/deserializer for a C++ type
- generate **Python/JS/… code** to use a C++ type
- generate a **.WINMD binary metadata** file for a C++ type
- customize a class’s **object layout** (e.g., `class(compressed)`)
- generate a **unit test function** to execute all functions in a namespace that are marked as unit tests
- make a class’s **interface async** (e.g., `class(async)`, such as changing return values: `T` → `std::future<T>`)

… all in portable, testable, shareable Standard C++ code

no side compilers, no proprietary language extensions

prediction (prove me wrong!): we are now on a direct path to be able to sunset Qt moc, C++/CLI, C++/CX, etc. in the next decade

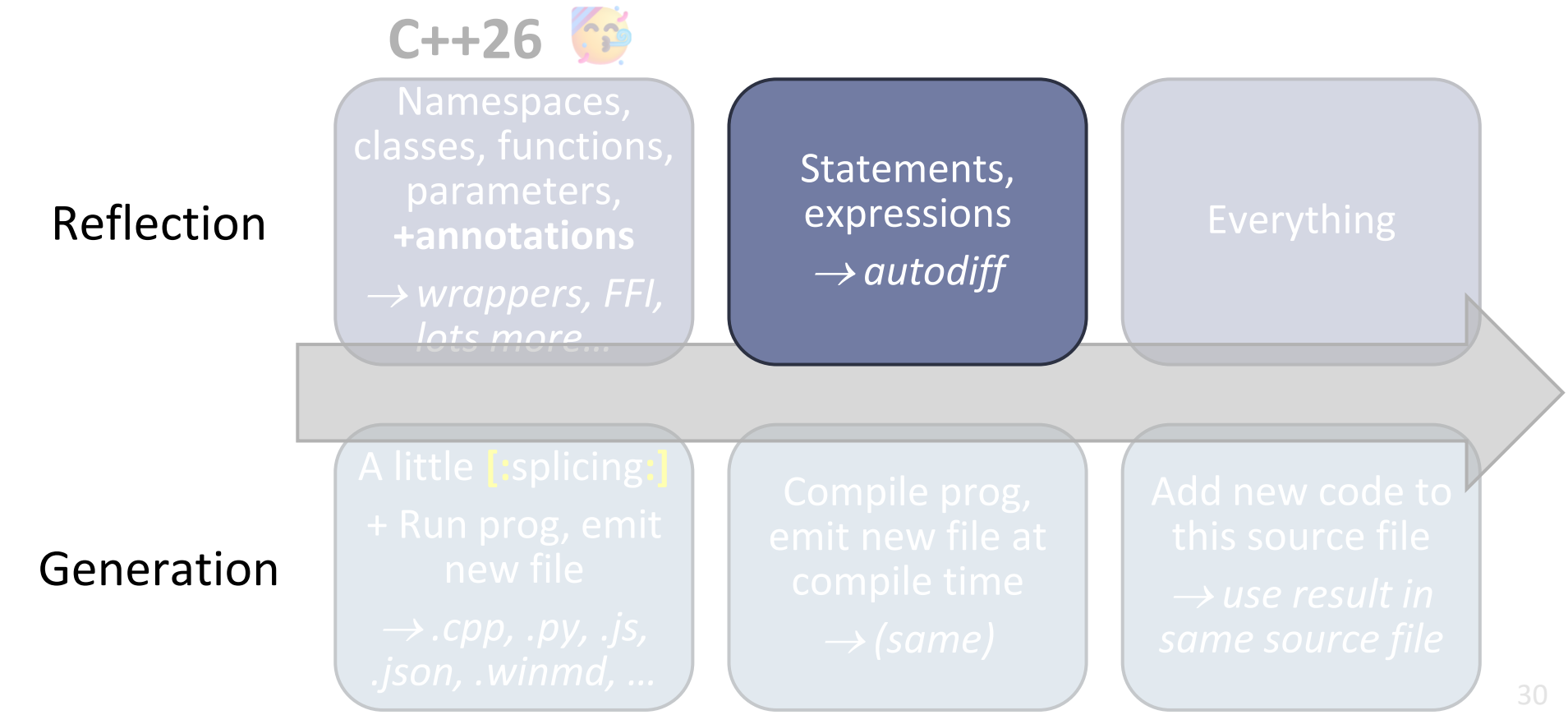
Key to remember:

If you can write the code you can generate the code

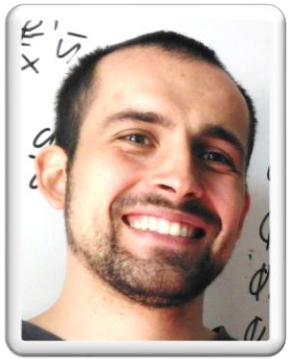
Q: What would *you* do if you had (the equivalent of) an accurate always-up-to-date C++ parser?

A: We're all about to find out together over the next decade!

The reflection (r)evolution: Roadmap sketch



$$\frac{dy}{dx}$$



Automatic differentiation (autodiff, AD)

Option 1: Symbolic differentiation, Wolfram/Maple

- ✓ Accurate, exact up to machine precision
- ☒ Expression swell, no loops/branches/functions

Option 2: Numerical differentiation, $(f(x+h)-f(x))/h$

- ✓ Computable
- ☒ Doesn't scale, truncation vs rounding error

Option 3: Transform source code

- ✓ Accurate: Equal to symbolic differentiation
- ✓ Computable: Use chain rule on individual ops
- ✓ Supports loops/branches/functions: Single pass propagates original values **+ their derivatives**

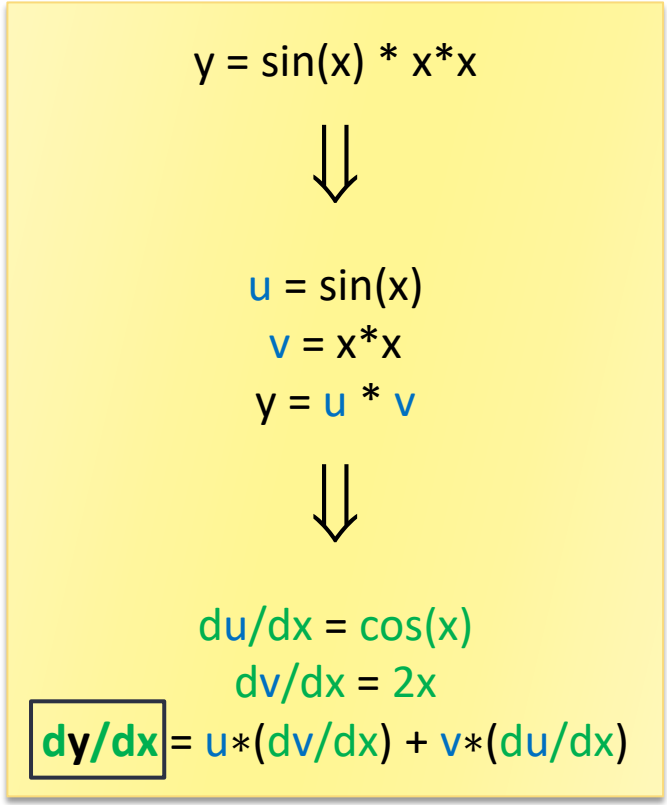
Automatic differentiation (autodiff, AD)

Option 3: Transform source code

- ✓ Accurate: Equal to symbolic differentiation
- ✓ Computable: Use chain rule on individual ops
- ✓ Supports loops/branches/functions: Single pass propagates original values **+ their derivatives**

Bonus: Got multiple inputs/outputs?
Compute **partial derivatives** on single pass too (i.e., Jacobian matrix of all first-order partials, without computing the matrix itself)

Bonus: Need second (third, etc.) derivatives?
Compute them too, still single pass



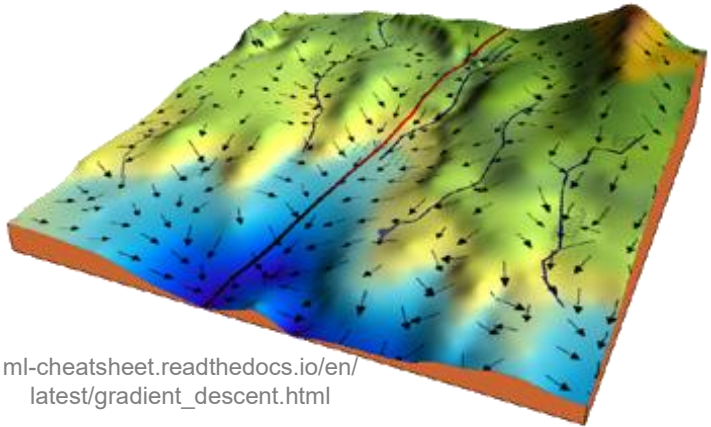
Why should C++ developers care?

Gradients (inputs' partial derivatives) power modern computing

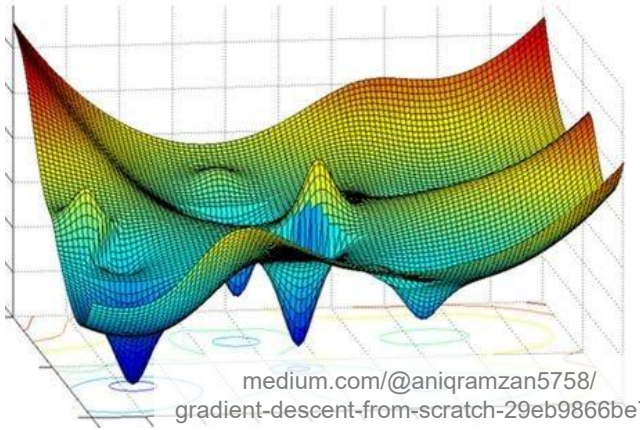
- **Machine learning & deep learning: find global/better minimum for Bs/Ts of inputs**
- Also: Scientific computing, optimization, physics engines
- **Key: Function can include (possibly complex) code**, not just pure math-text notations

AD replaces hand-written derivatives

- Faster development and maintenance, fewer bugs — C++ bonus: faster execution



ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html



medium.com/@aniqramzan5758/gradient-descent-from-scratch-29eb9866be79

Why should C++ developers care?

C++ AD currently limited to libraries/extensions

- Operator overloading: CoDiPack, CppAd, Adept, …
  - Custom types: doesn't Just Work™ with your code
  - Less efficient: template libraries, harder to optimize
  - Less debuggable: expression templates

- Custom transformation: Tapenade (C), Clad (Clang)
  - Require custom compiler/extension

Portable reflection: 100% ordinary code

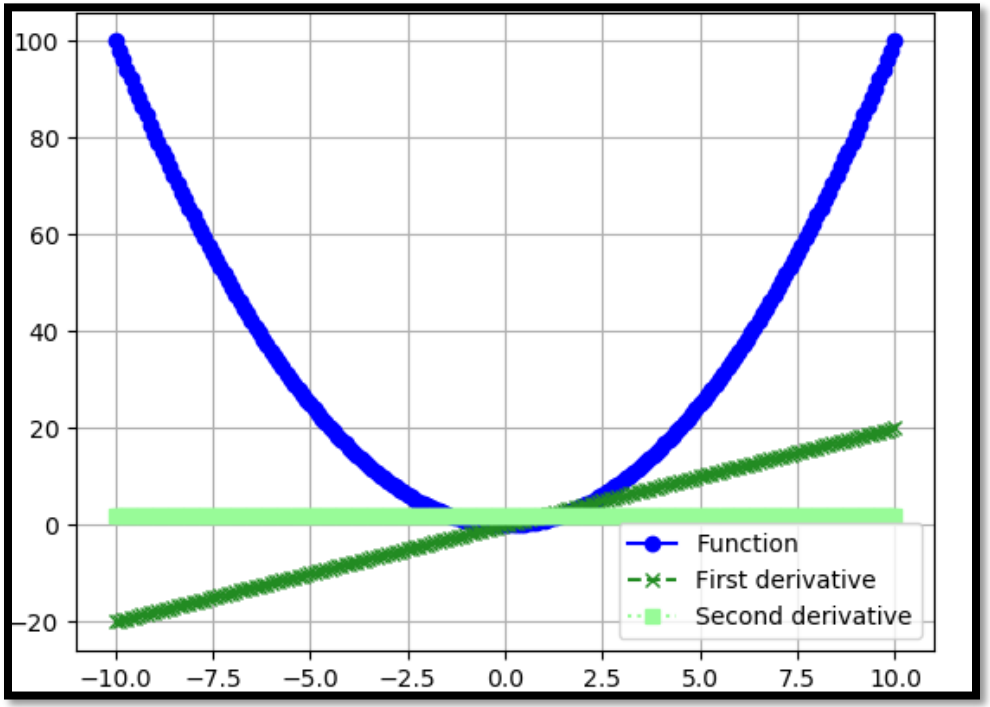
Everything just lights up (e.g., optimizers, all compilers)

Everything is easier: implementation, iteration, debugging, understanding, maintenance, adaptation, experimentation, performance, and an API vs. walking internal AST nodes…

Reflection

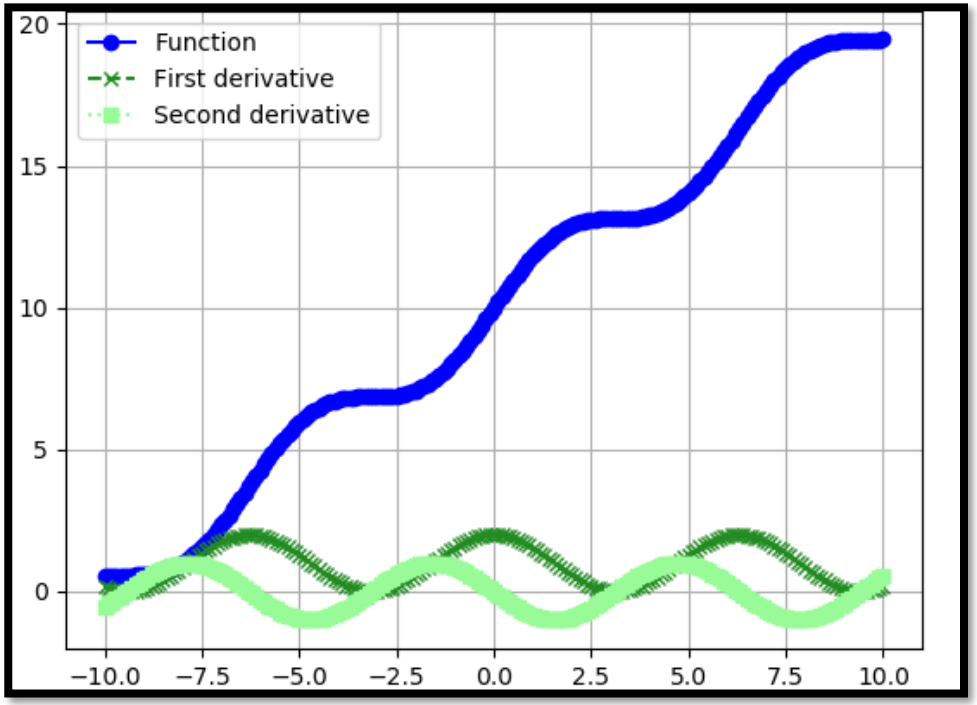
Example 1: y = x * x

```
ad_test: @autodiff<"order=2"> type = {
  f: (x: double) -> (y: double) = {
    y = x * x;
  }
}
```



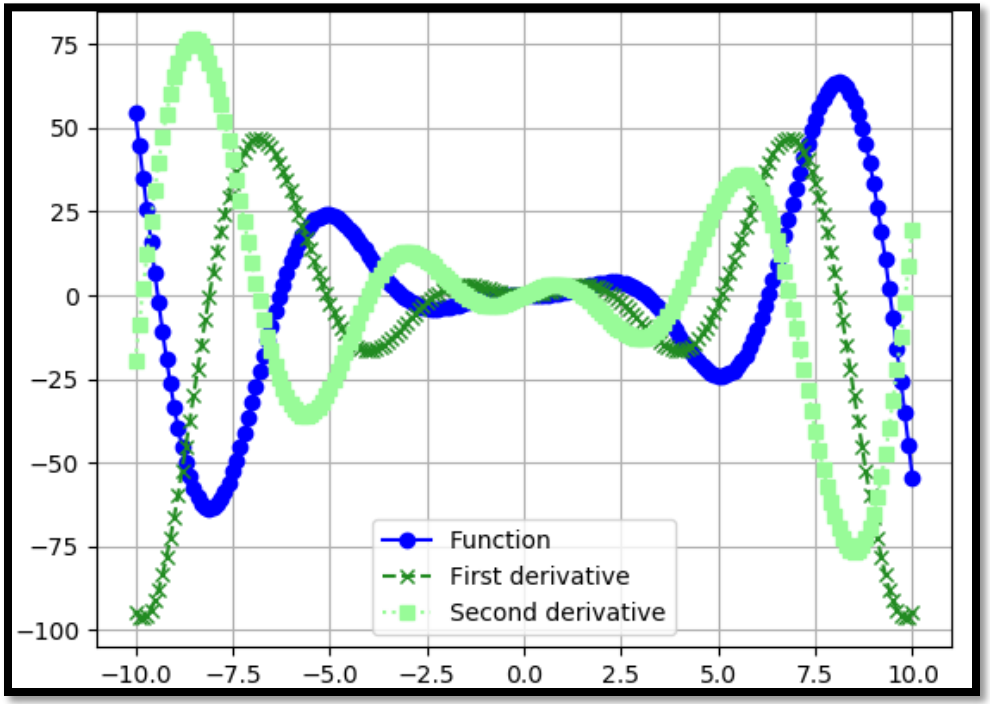
Example 2: y = x + sin(x) + 10

```
ad_test: @autodiff<"order=2"> type = {
  f: (x: double) -> (y: double) = {
    y = x + sin(x) + 10;
  }
}
```



Example 3: sin(x) * x*x

```
ad_test: @autodiff<"order=2"> type = {
  f: (x: double) -> (y: double) = {
    y = sin(x) * x*x;
  }
}
```

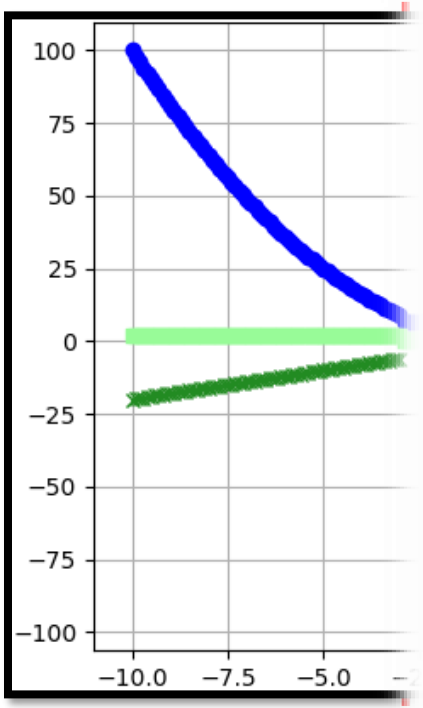


Example 1+2+3: Discontinuity

```
ad_test: @autodiff<"order=2"> type = {
  f: (x: double) -> (y: double) = {
    if x < -3 {
      y = x * x;
    }
    else if x < 3 {
      y = x + sin(x) + 10;
    }
    else {
      y = sin(x) * x*x;
    }
  }
}
```

Example 1+2+3: Discontinuity

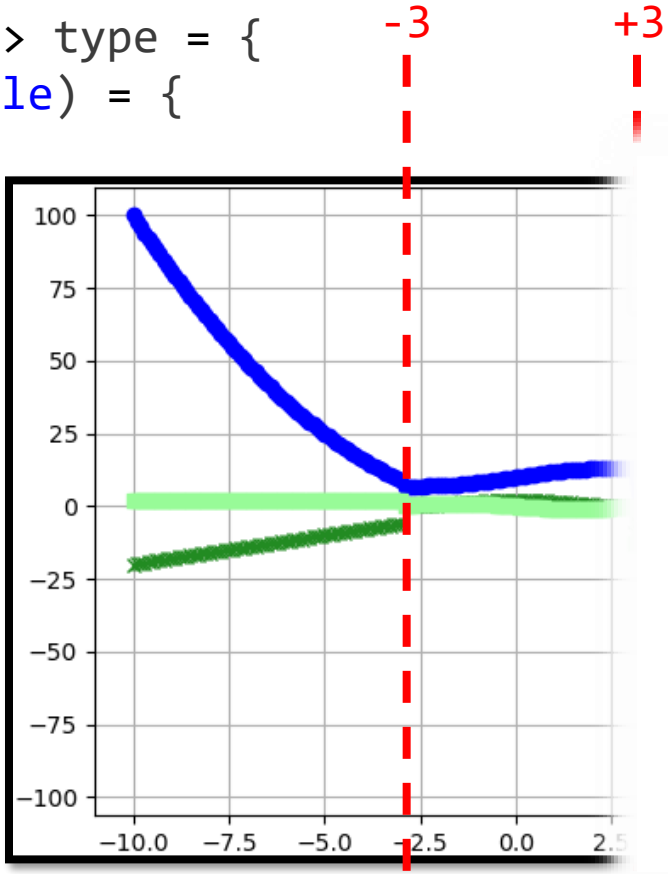
```
ad_test: @autodiff<"order=2"> type = {
  f: (x: double) -> (y: double) = {
    if x < -3 {
      y = x * x;
    }
    else if x < 3 {
      y = x + sin(x) + 10;
    }
    else {
      y = sin(x) * x*x;
    }
  }
}
```



?

Example 1+2+3: Discontinuity

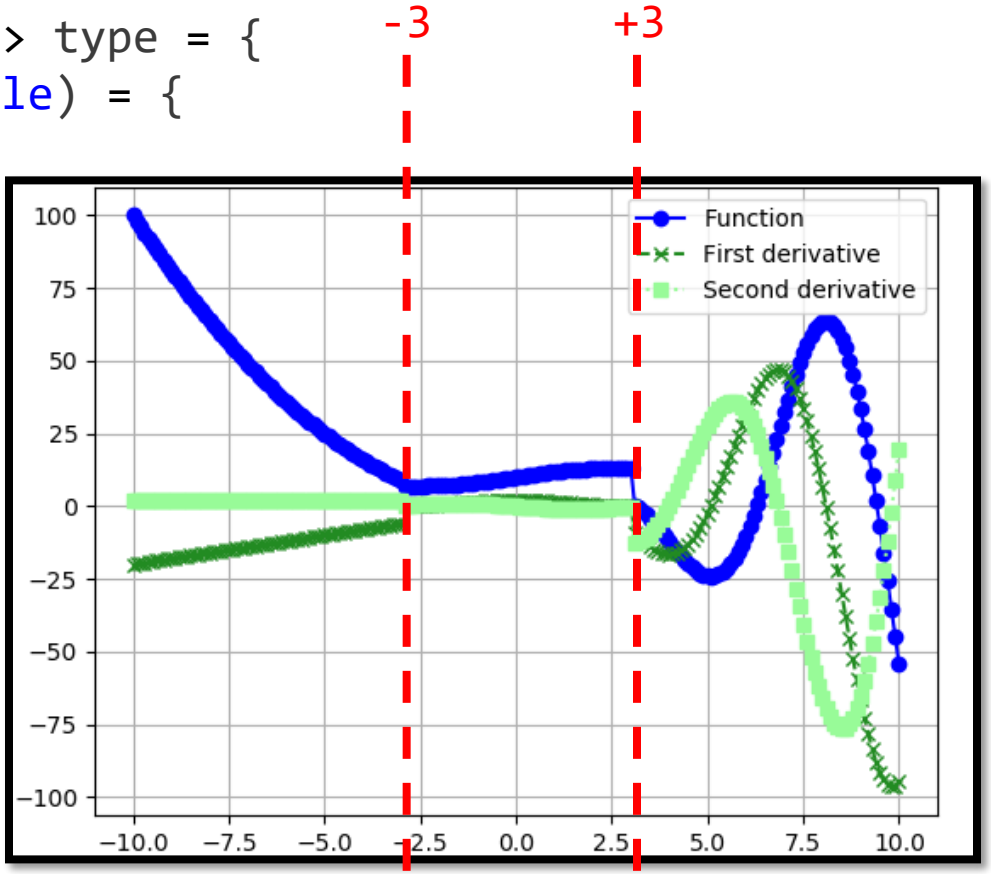
```
ad_test: @autodiff<"order=2"> type = {
  f: (x: double) -> (y: double) = {
    if x < -3 {
      y = x * x;
    }
    else if x < 3 {
      y = x + sin(x) + 10;
    }
    else {
      y = sin(x) * x*x;
    }
  }
}
```



?

Example 1+2+3: Discontinuity

```
ad_test: @autodiff<"order=2"> type = {
  f: (x: double) -> (y: double) = {
    if x < -3 {
      y = x * x;
    }
    else if x < 3 {
      y = x + sin(x) + 10;
    }
    else {
      y = sin(x) * x*x;
    }
  }
}
```

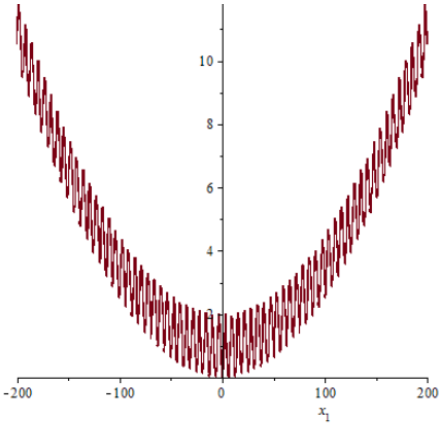


Griewank function

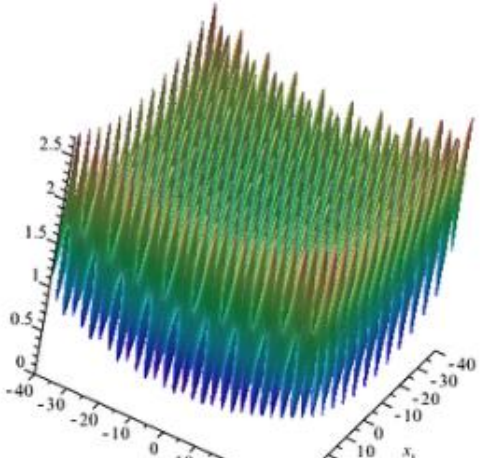
$$f(x) = 1 + \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} P_i(x_i) \text{ with } P_i(x_i) = \cos\left(\frac{x_i}{\sqrt{i}}\right)$$

A handy stress-test to make sure global optimizers don't "get stuck"

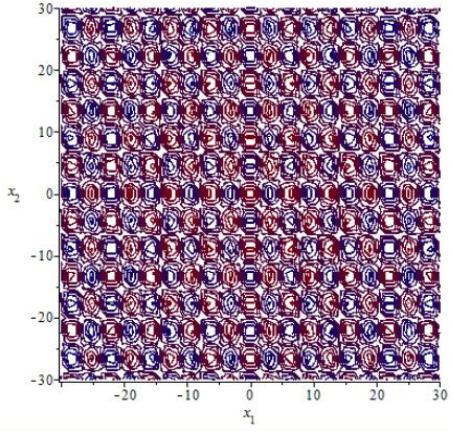
- Shallow bowl (quadratic) ⇒ one true global minimum at x=0
- Dense ripples (product of cosines) ⇒ many local minima to trap naïve methods
- Cheap to compute, scales to any number of dimensions



2D Plot of the one dimensional Griewank function



3D plot of the two dimensional Griewank function



Contour plot of the two dimensional Griewank function

唐戈 and Gisling, via en.wikipedia.org/wiki/Griewank_function

Griewank function

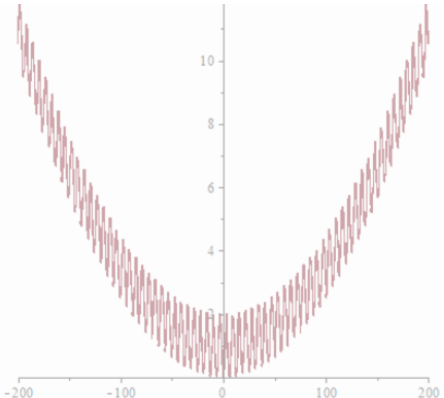
differentiating out of context

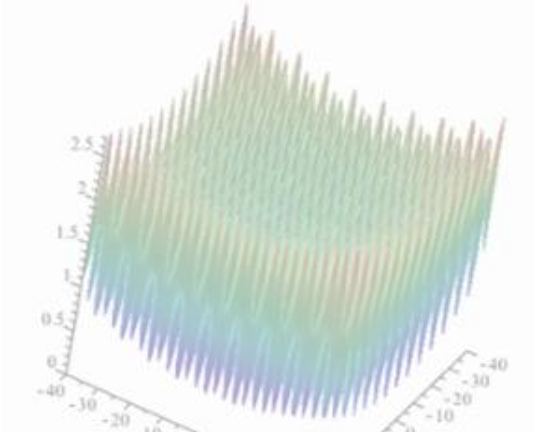
$$f(x) = 1 + \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} P_i(x_i) \text{ with } P_i(x_i) = \cos\left(\frac{x_i}{\sqrt{i}}\right)$$

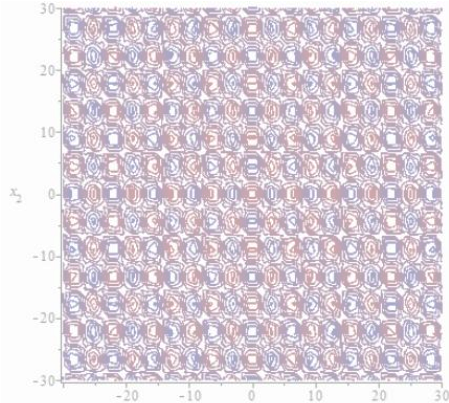
loops

function calls

passive values







Some C++ world's firsts (with working live demos)

2017: World's first authoring C++ classes using reflection

Thanks **Andrew Sutton** & team! (Lock3 Software)

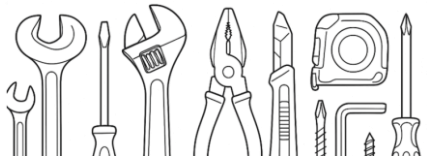
Today: World's first C++ automatic differentiation using reflection (via cppfront)

2,400 Cpp2 LOC

Thanks **Max Sagebaum**! (Chair for Scientific Computing, University of Kaiserslautern-Landau (RPTU))

Roadmap

Overview A few good tools



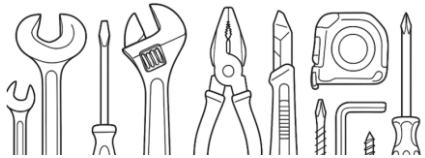
Today Some is good …

Tomorrow … more is better …

Beyond … and “everything” is just about right

Roadmap

Overview A few good tools



Today Some is good …

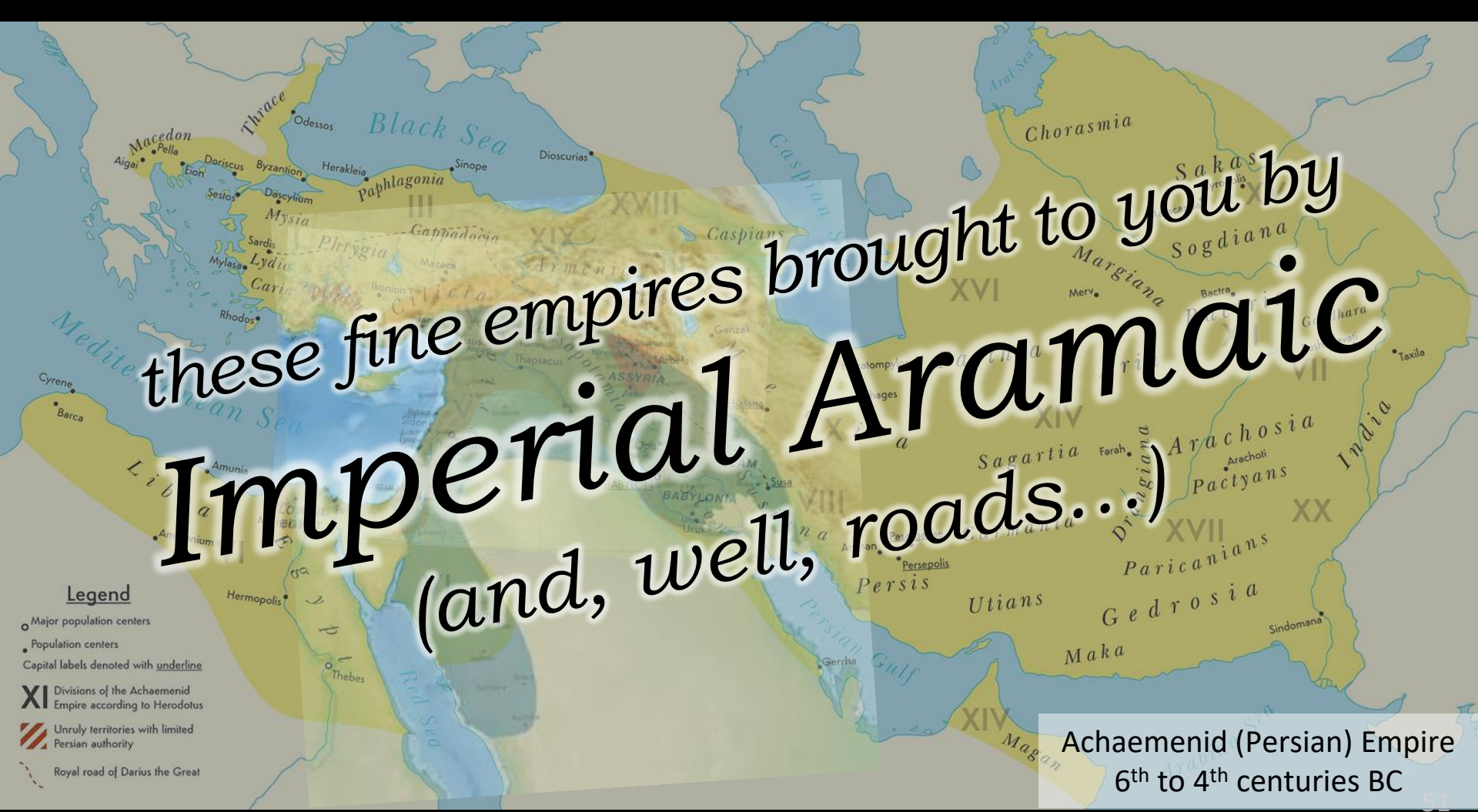
Tomorrow … more is better …

Beyond … and “everything” is just about right

Finally, consider a major unsolved industry problem

Neo-Babylonian Empire
7th to 6th centuries BC

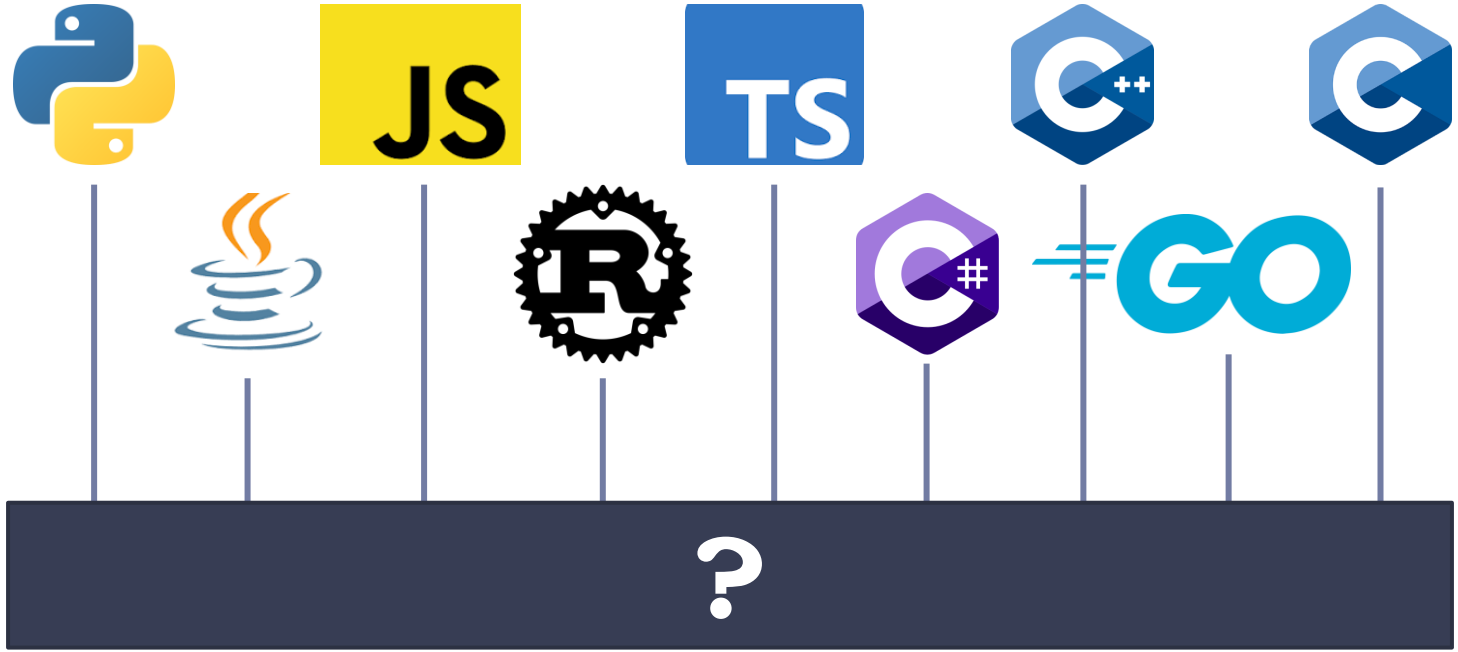
Achaemenid (Persian) Empire
6th to 4th centuries BC


these fine empires brought to you by
Imperial Aramaic
(and, well, roads...)
Legend
Major population centers
Population centers
Capital labels denoted with underline
XI Divisions of the Achaemenid Empire according to Herodotus
Unruly territories with limited Persian authority
Royal road of Darius the Great
Achaemenid (Persian) Empire
6th to 4th centuries BC

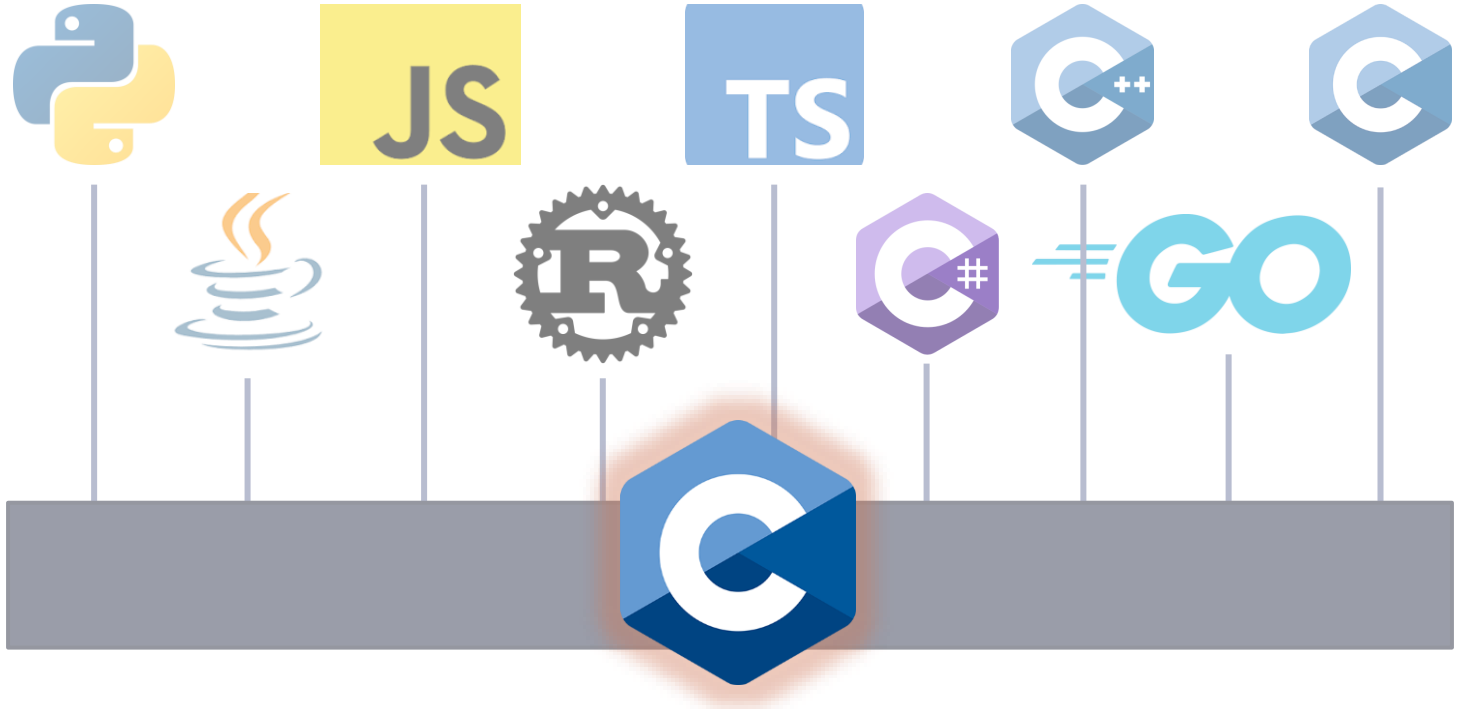
Roman/Byzantine Empire
1st century BC to
5th/15th centuries AD


these fine empires brought to you by
Koine Greek
(and Latin, and more roads…)
Roman/Byzantine Empire
1st century BC to
5th/15th centuries AD

Q: What's the lingua franca of programming?



Q: What's the lingua franca of programming?



A weakness == an opportunity

C is the lingua franca FFI systems language

- Every API consumable from most languages (e.g., OS APIs)
- Our civilization's ABI-stable "universal interop glue"

Q: What's, um, wrong with that?

- A: Low abstraction + low safety
- The world is built on "unencapsulated structs + raw pointers + manual lifetimes"

Q: Why not "encapsulated classes + methods"?

- What a wonderful new idea! We should try that!
- See: ***COM***, ***CORBA***, ***XPCOM***, ***Objective-C runtime***, ***GObject***, ***.NET runtime / CLR***, ***Java Native Interface***, ***JNI*** *(mostly to call C rather than reverse)*, ***UNO***, ***SOAP/WSDL***, *Google* ***protocol buffers + gRPC***, ***SWIG***, …

I see only one "just maybe" hope in my lifetime…

Let’s consider this major unsolved industry problem

… where I think C++ reflection is our best chance at a solution

Disclaimers: The following slides and demos…

are a proof of concept only
generate 100% ordinary portable Standard C++ code
use cppfront reflection as a proxy for post-C++26 standard reflection

P2911R1 - Python Bindings with Value-Based Reflection

Authors: Adam Lach, Jagrut Dave
Last Updated: Sep 18, 2023

Function Parameter Reflection in Reflection for C++26

Document #: P3096R12
Date: 2025-06-20
Reply-to: Adam Lach
Dan Katz
Walter Genovese

Using Reflection to Generate C++ Python Bindings

Callum Piper

14:00-14:20, Thursday, 3rd April 2025 - Empire

A future I wonder about

Use C++ type from Python

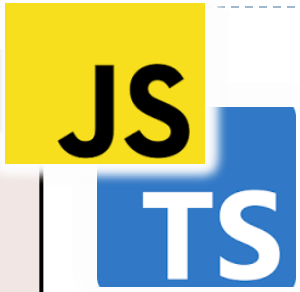
```
import widgetlib

w = widgetlib.widget(10)
print(w.add(5))       # prints 15
print(w.add(i=7))     # prints 22
print(w.shout("hi")) # prints hi!
```

C++ (near future?)

```
class(python) widget {
public:
    widget(int i);
    int add(int i);
    string shout(string s);
    // ...
};
```

A future I wonder about

Use C++ type from JavaScript/TypeScript




```
import createModule from './widget.js';

const Module = await createModule();
const w = new Module.Widget(10);
console.log(w.add(5));      // 15
console.log(w.add(7));      // 22
console.log(w.shout("hi")); // hi!
w.delete();
```

C++ (near future?)

```
class(javascript) widget {
public:
    widget(int i);
    int add(int i);
    string shout(string s);
    // ...
};
```

A future I wonder about

Use C++ type from Go (golang)

```
package main
import "fmt"

func main() {
    w := NewWidget(10)
    defer w.Close()
    fmt.Println(w.Add(5))       // 15
    fmt.Println(w.Add(7))       // 22
    fmt.Println(w.Shout("hi")) // hi!
}
```

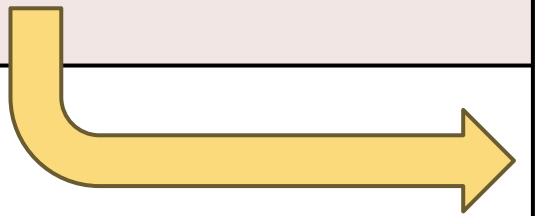
C++ (near future?)

```
class(golang) widget {
public:
    widget(int i);
    int add(int i);
    string shout(string s);
    // ...
};
```

A future I wonder about

Use C++ type from C

```c
int main() {
    widget_handle* w = widget_create(10);
    if (!w) { fprintf(stderr, "can't create widget\n"); return 1; }
    printf("+5 = %d\n", widget_add(w, 5));       // 15
    printf("+7 = %d\n", widget_add(w, 7));       // 22
    printf("hi = %s\n", widget_shout(w, "hi")); // hi!
    widget_destroy(w);
    return 0;
}
```



C++ (near future?)

```c
class(c_lang) widget {
public:
    widget(int i);
    int add(int i);
    string shout(string s);
    // ...
};
```

Early user notes: Alan Gray @ 12d

Use case	C# wrappers for 5,000+ C++ types
Current tool	C++ → generated .i file → SWIG → C#
Issues	templates, extern template, constexpr, modules, requires, nested class accessibility, …
Attempt 1	Tried C++/CLI, but that only supported C++20 at the time 😞
Experiment	**Tried rewriting in Dan Katz' prototype … with Dan's help!** 3 days: Get Clang P2996 and get it working 1 all-nighter: Parsing all examples that had issues in SWIG

"… generating C# wrapper prototypes transforming std::string, std::wstring, in-house Text class, BStr with charset correctly into ref string result, out string result etc etc.

The most stunning part is we are just sprinkling a few consteval keywords, then doing what we do best — writing C++ code."

Early user notes: Alan Gray @ 12d

*"Also, reflection could **generate unit tests** to know the compiler has parsed code into the correct internal representation. So the compiler testing itself.*

*Everything points to Reflection being able to accomplish what we need and in a way very **familiar to any modern C++ programmer**.*

I have lost count of the dodgy C++ parsers I have written over the last 30 years to achieve source code generators.

***Outstanding questions** will be how efficiently Reflection can handle 50 MLOC, boundary cases, integration into tooling/build systems etc etc."*

Use C++ type from Python

```
import widgetlib

w =
print
print
print
```

Use C++ type from JavaScript/TypeScript

```
import createModule from './widget.js'

const Module = await createModule();
const w = new Module.Widget(10);
console.log(w.add(5));       // 15
console.log(w.add(7));       // 22
console.log(w.shout("hi")); // hi!
w.delete();
```

Use C++ type from Go (golang)

```
package main
import "fmt"

unc main() {
    w := NewWidget(10)
    defer w.Close()
    fmt.Println(w.Add(5))
    fmt.Println(w.Add(7))
    fmt.Println(w.Shout("hi
}
```

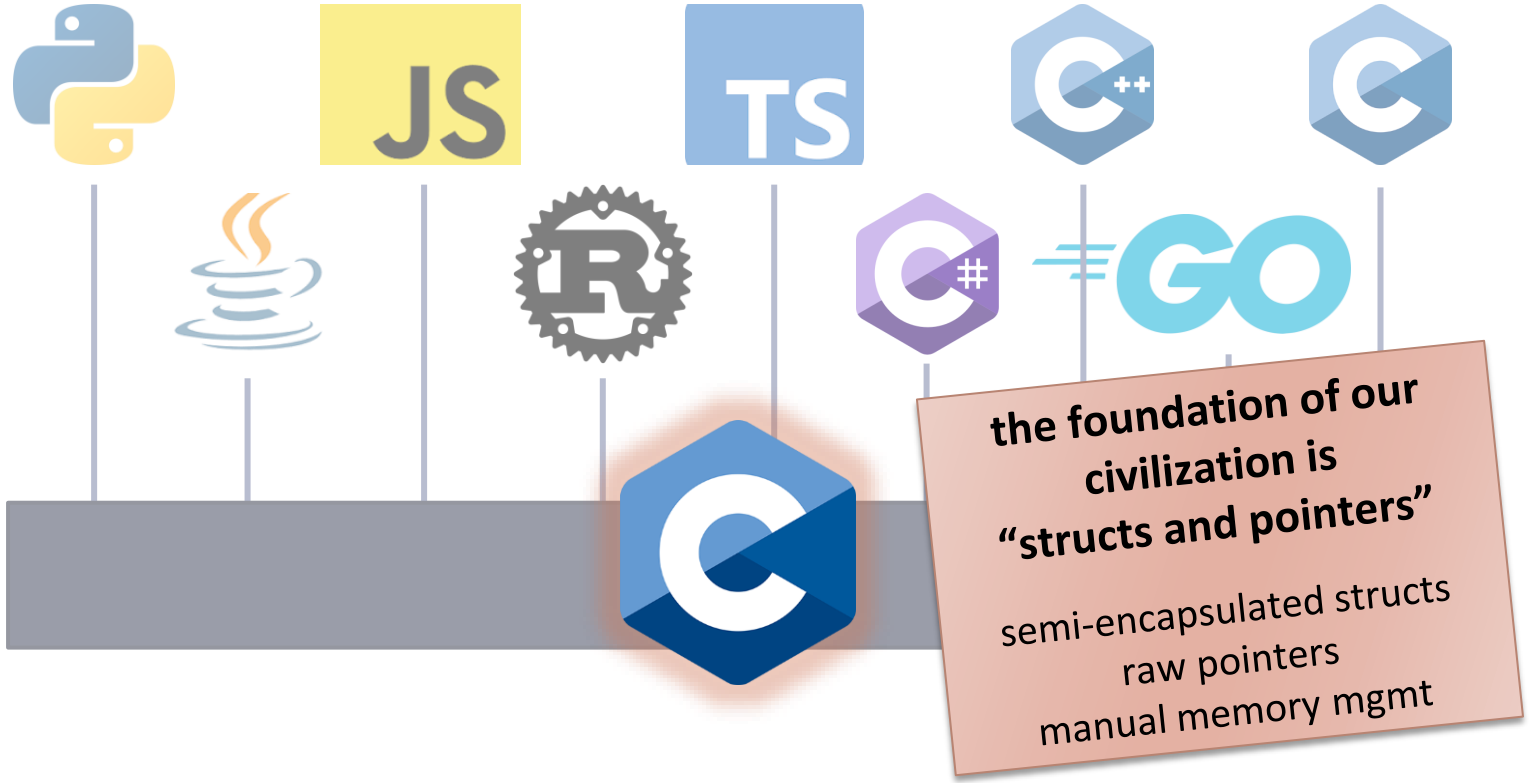
Use C++ type from C

```
int main() {
    widget_handle* w = widget_create(10);
    if (!w) { fprintf(stderr, "can't create widg
    printf("+5 = %d\n", widget_add(w, 5));
    printf("+7 = %d\n", widget_add(w, 7));
    printf("hi = %s\n", widget_shout(w, "hi"));
    widget_destroy(w);
    return 0;
}
```

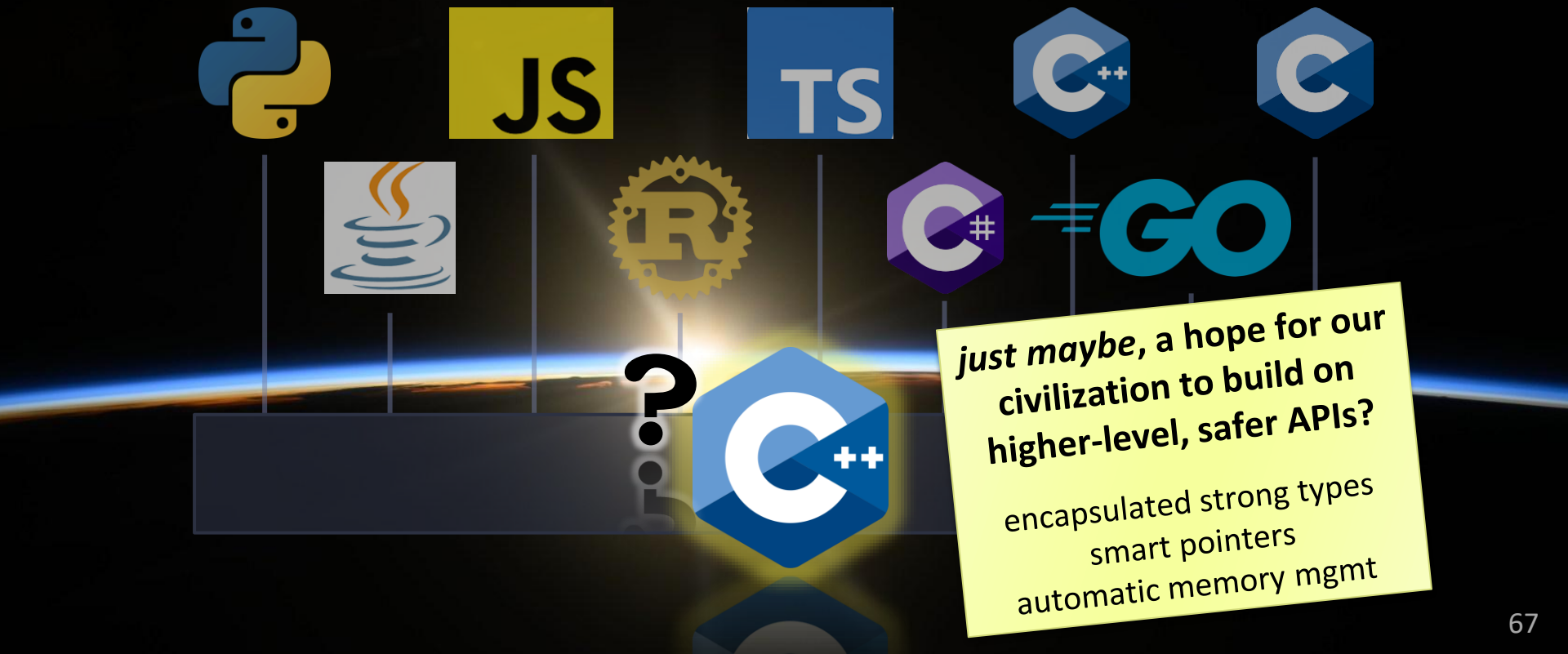
C++ (near future?)

```
class(api) widget {

    // ...

};
```

Q: What's the lingua franca of programming?



Q: What *could be* our lingua franca?



in Sofia, three months ago…

Dan Katz: “Reflection is on track to make C++26 this week!”

Hana Dusíková: (deadpan, little shrug) “**Whole new language.**”


And I, for one,
welcome our new
reflection overlords

A little confession…

Metaclasses

Document Number:	**P0707 R0**
Date:	2017-06-18
Reply-to:	Herb Sutter (hsutter@microsoft.com)
Audience:	SG7

Declarative class authoring using `consteval` functions + reflection + generation (aka: Metaclasses for generative C++)

Document Number:	**P0707 R5**
Date:	2024-10-12
Reply-to:	Herb Sutter
Audience:	SG7, EWG

A little confession…

1957: Lockheed L-1649 Starliner enters service

- ✓ **+10%** luxurious, **+10%** range vs. Douglas DC-7C
- ✓ Compatible with existing infrastructure

Perfection of the turboprop airplane

1958: Both L-1649 and DC-7C cancel production

Lockheed Starliner (1956-58)

A little confession…

1957: Lockheed L-1649 Starliner enters service

- ✓ **+10%** luxurious, **+10%** range vs. Douglas DC-7C
- ✓ Compatible with existing infrastructure

Perfection of the turboprop airplane

1958: Both L-1649 and DC-7C cancel production

Lockheed Starliner (1956-58)

1958: Boeing 707 starts production

- ✓ **2**× speed, **2**× passengers,
  quiet (less passenger-fatiguing noise/vibration)
- ✓ **Compatible** with existing infrastructure

Still an airplane, not "something incompatible"
New era of innovation
Fully replaced Starliner by 1962

Boeing 707 (1958-79)

A little confession…

1958: Boeing 707 starts production

- ✓ **2**× speed, **2**× passengers,
  quiet (less passenger-fatiguing noise/vibration)
- ✓ **Compatible** with existing infrastructure

Still an airplane, not “something incompatible”
New era of innovation
Fully replaced Starliner by 1962

Boeing 707 (1958-79)

A little confession…

Metaclasses

Document Number:	**P0707 R0**
Date:	2017-06-18
Reply-to:	Herb Sutter (
Audience:	SG7

Boeing 707 (1958-79)

Reflection

C++'s Decade-Defining Rocket Engine

HERB SUTTER

CITADEL | Securities