NOT GETTING LOST IN TRANSLATION

Daniela Engert - BinaryCore 2024

Outline

- The beginnings
- Let's talk
- A new library
- A new world
- What now
- An experiment
- Move on

ABOUT ME

- Electrical engineer
- Build computers and create software for more than 40 years
- Develop hardware and software in the field of applied digital signal processing for 30 years
- Member of the C++ committee (learning novice) for 4 years (EWG, SG15)

- employed by 

TIMELINE

2009

Boost::format

German only

END

STARTUP CULTURE

Features! More features!

Get a name!

Get the architecture right!

BOOST::FORMAT

Why?

- it's compatible to C's printf() family
- it's different from C's printf() family
- it supports advanced formatting specifiers
- it "creatively" uses operators to separate the formatting specification from the arguments
- it supports formatting of user-defined types
- it supports stream manipulators

Example:

```
#include <boost/format.hpp>

std::cout << boost::format("formatting specifier") % argument1 % argument2;
```

let's talk

TIMELINE


Boost::format + Boost::locale
Converted to English (core + GUI)
2 Translations (German + Czech)
2009
2014
Boost::format
German only

STRING TRANSLATION

Questions

- which translation system?
- is it mature?
- does it have an ecosystem?
  - editors
  - tools
  - build system
- does it fit? Does it feel "natural"?

- can it be used by end users?
  - domain experts
  - not even remotely firm in IT

TRANSLATION FORMAT

~~Binary~~

~~XML~~

~~JSON~~

Text

GETTEXT

pretty standard, available on all major platforms

two major implementations

- GNU gettext
- Boost::locale

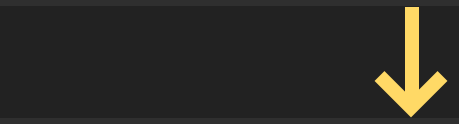
both rely on C locales (or their C++ equivalents) and message catalogs

P2918 brings GNU gettext as a striking example to support runtime format strings!

TOOLS

step 1: mark translatable text

```
#include <boost/format.hpp>

std::cout << boost::format("formatting specifier") % argument1 % argument2;
```



```
#include <boost/format.hpp>
#include <boost/locale/message.hpp>
using boost::message::translate

std::cout << boost::format(translate("formatting specifier")) % argument1 % argument2;
```

step 2: scan source files with 'xgettext' for marked text

```
$(Keyword) is e.g. "translate"

xgettext --keyword --keyword=$(Keyword):1,1t --keyword=$(Keyword):1c,2,2t --keyword=$(Keyword):1,2,3t
--keyword=$(Keyword):1c,2,3,4t a.cpp b.cpp c.hpp ...
```

TOOLS

Result: a so-called PO template file

- language agnostic
- transient

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"
"Plural-Forms: nplurals=INTEGER; plural=EXPRESSION;\n"
#: main.cpp
msgctxt "(optional context)"
msgid "singular"
msgid_plural "(optional plural)"
msgstr[0] ""
msgstr[1] ""
```

MO' TOOLS

step 3: each template is then processed into individual translation files, one for each country, region, specialties

- msguniq
- msgmerge
- msgattrib

The result is a collection of PO (portable object) files

- specific to language, country, region, company
- stable
- checked into repositories

you can have multiple of them, e.g. one for each subsystem

PO FILE CONTENT

```
#
msgid ""
msgstr ""
"Project-Id-Version: Example 1.0\n"
"Report-Msgid-Bugs-To: somebody@example.com\n"
"POT-Creation-Date: \n"
"PO-Revision-Date: 2024-04-30 11:38+0200\n"
"Last-Translator: EMAIL@ADDRESS\n"
"Language-Team: Czech\n"
"Language: cs\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Plural-Forms: nplurals=3; plural=(n==1 ? 0 : n>=2 && n<=4 ? 1 : 2);\n"
"X-Generator: Poedit 3.4.2\n"

# This is only a example
#: main.cpp
msgctxt "Disambiguation"
msgid "Singular"
msgid_plural "Plural"
msgstr[0] "Singulární"
msgstr[1] "Duální"
msgstr[2] "Plurál"
```

EVEN MORE TOOLS

step 4: (possibly) combine multiple PO files and compile them to the target directory, in the target format

- msgmerge
- msgfmt

This checks the validity of format specifiers and placeholders!

The result is a collection of MO (machine object) files

- specific to language, country, region, company
- binary
- optimized for consumption
- distributed to the customer's machine

The format is described in the manual:
https://www.gnu.org/software/gettext/manual/gettext.html#MO-Files

MO FILE CONTENT

- Header
- two string offset tables
  - original (contains context)
  - translated
- optional, unspecified hashing table
- all language forms (i.e. plurals) are NUL separated!

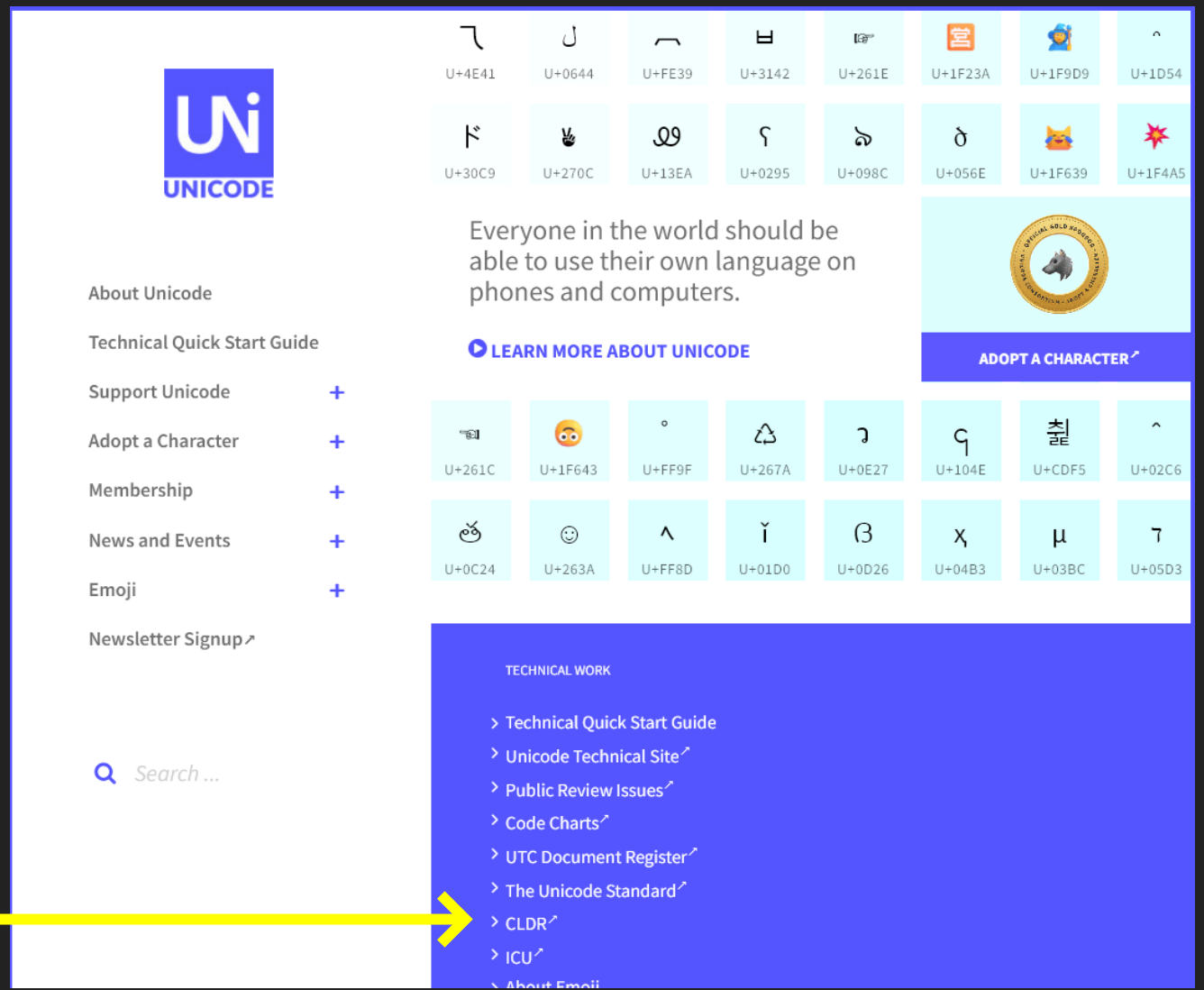
Every entry (besides strings) is a 4-byte unsigned integer

UNICODE

Unicode covers more than glyphs, code points, and their encodings!

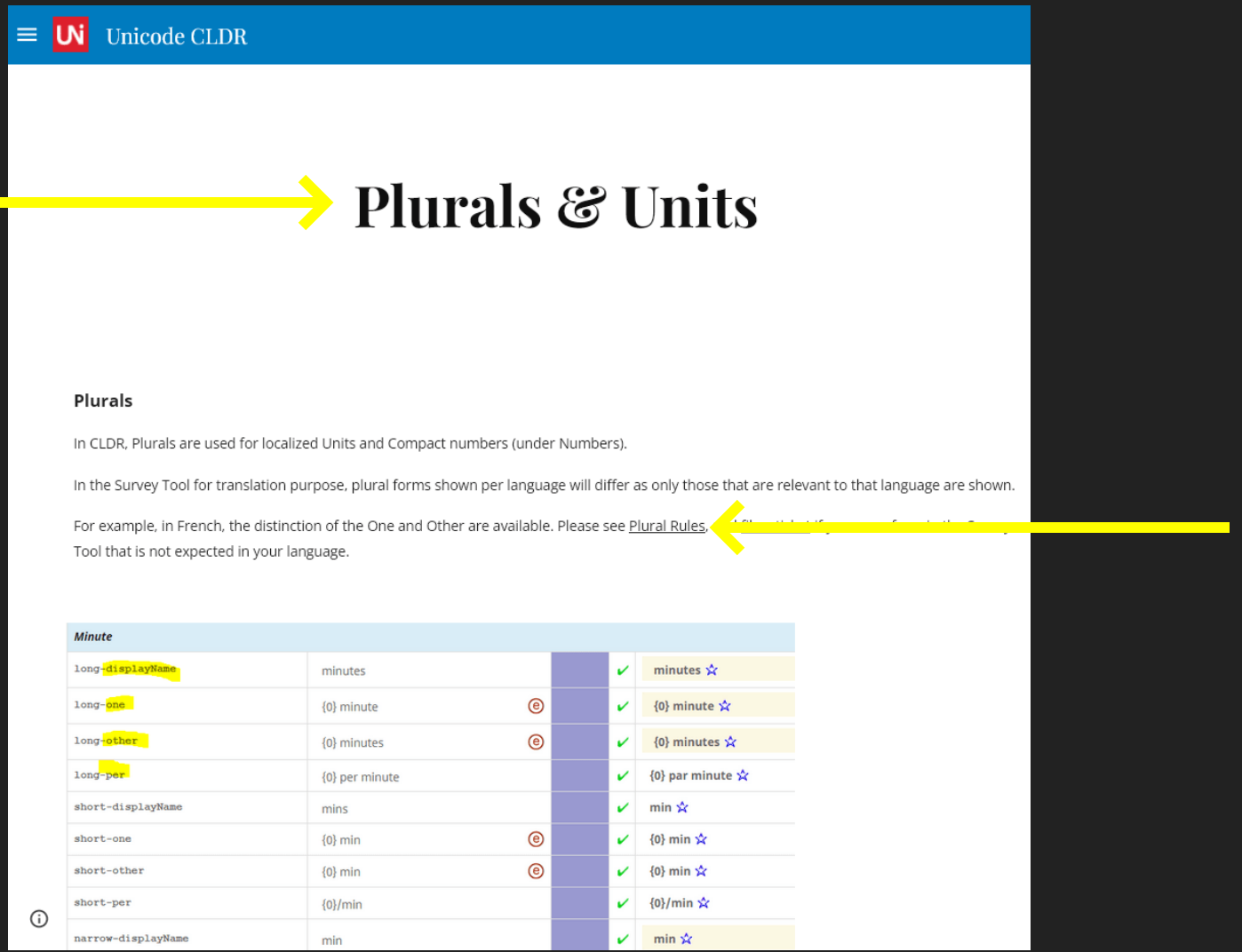
CLDR

Common
Language
Data
Repository



CLDR

The CLDR defines machine readable rules for e.g. translation



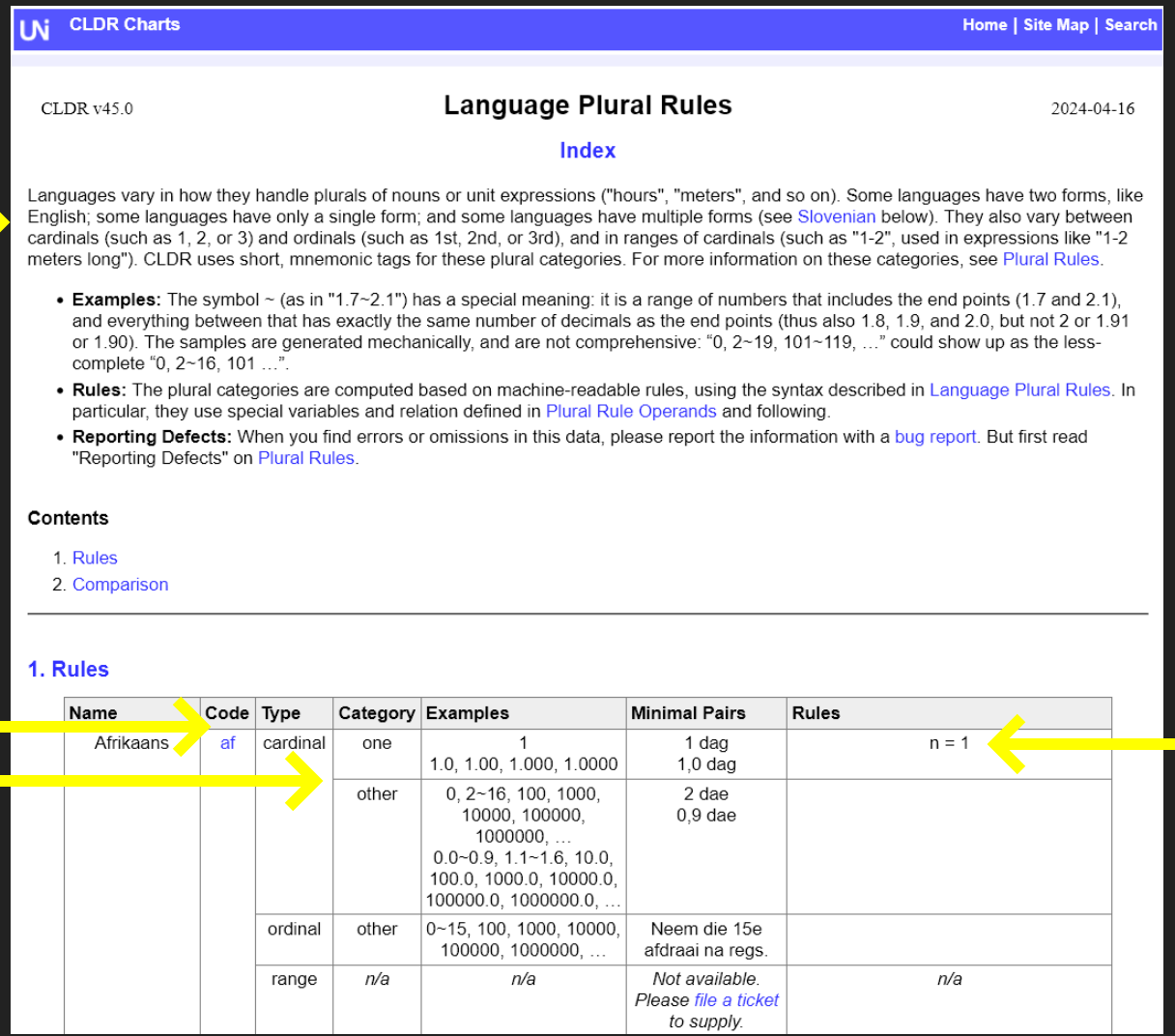
LANGUAGE PLURAL RULES

Languages vary in plural forms

ISO 639

ISO 3166,

as POSIX



CLDR Charts — Home | Site Map | Search

CLDR v45.0

Language Plural Rules

2024-04-16

Index

Languages vary in how they handle plurals of nouns or unit expressions ("hours", "meters", and so on). Some languages have two forms, like English; some languages have only a single form; and some languages have multiple forms (see Slovenian below). They also vary between cardinals (such as 1, 2, or 3) and ordinals (such as 1st, 2nd, or 3rd), and in ranges of cardinals (such as "1-2", used in expressions like "1-2 meters long"). CLDR uses short, mnemonic tags for these plural categories. For more information on these categories, see Plural Rules.

- **Examples:** The symbol ~ (as in "1.7~2.1") has a special meaning: it is a range of numbers that includes the end points (1.7 and 2.1), and everything between that has exactly the same number of decimals as the end points (thus also 1.8, 1.9, and 2.0, but not 2 or 1.91 or 1.90). The samples are generated mechanically, and are not comprehensive: "0, 2~19, 101~119, …" could show up as the less-complete "0, 2~16, 101 …".
- **Rules:** The plural categories are computed based on machine-readable rules, using the syntax described in Language Plural Rules. In particular, they use special variables and relation defined in Plural Rule Operands and following.
- **Reporting Defects:** When you find errors or omissions in this data, please report the information with a bug report. But first read "Reporting Defects" on Plural Rules.

Contents

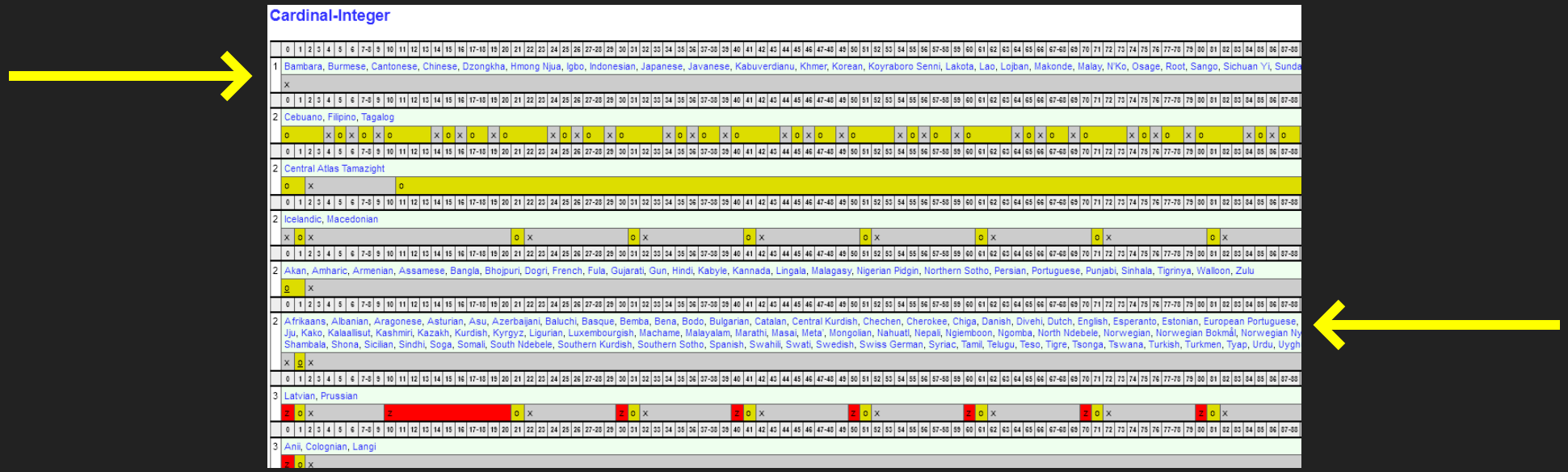
1. Rules
2. Comparison

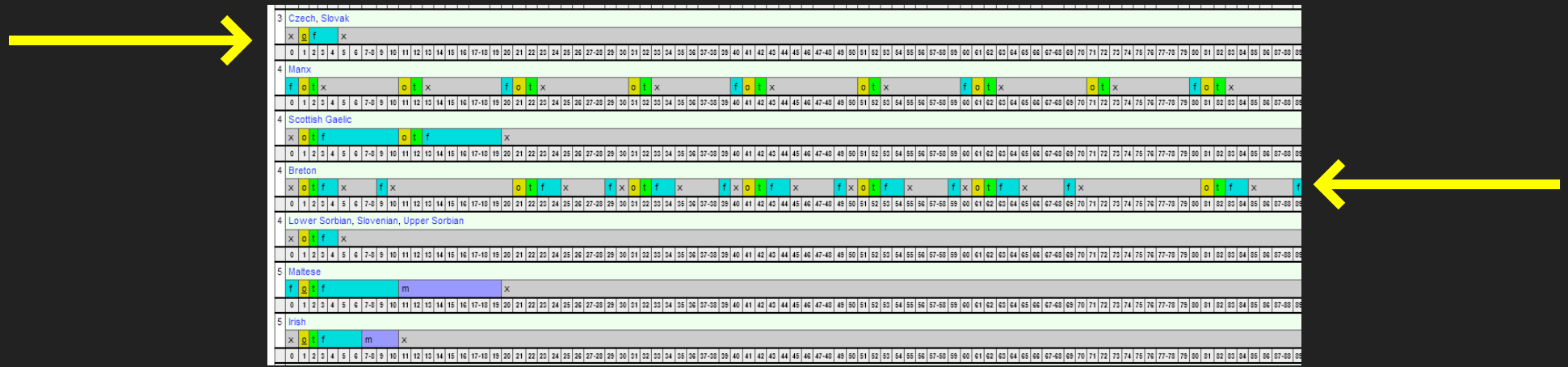
1. Rules

Name	Code	Type	Category	Examples	Minimal Pairs	Rules
Afrikaans	af	cardinal	one	1 1.0, 1.00, 1.000, 1.0000	1 dag 1,0 dag	n = 1
			other	0, 2~16, 100, 1000, 10000, 100000, 1000000, … 0.0~0.9, 1.1~1.6, 10.0, 100.0, 1000.0, 10000.0, 100000.0, 1000000.0, …	2 dae 0,9 dae	
		ordinal	other	0~15, 100, 1000, 10000, 100000, 1000000, …	Neem die 15e afdraai na regs.	
		range	*n/a*	*n/a*	*Not available. Please file a ticket to supply.*	*n/a*

PLURAL FORMS

from none, to simple, to complicated



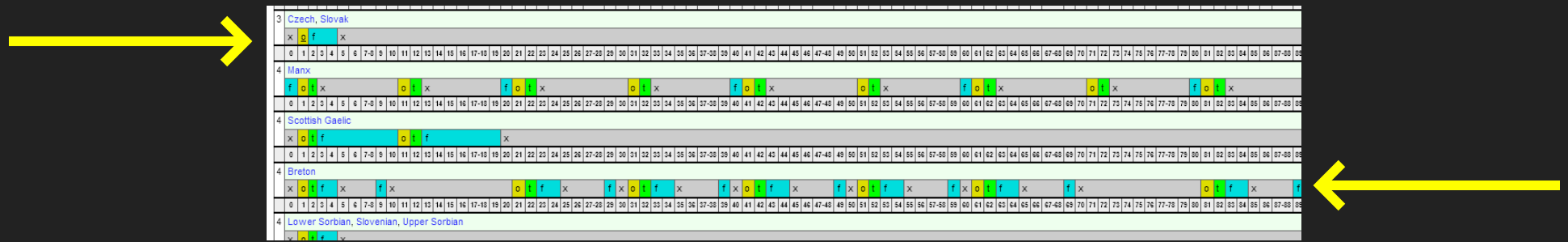


PLURAL FORMS

GNU gettext tries to cover many of these patterns in single-line C expressions.

Most implementations follow the CLDR rules because they cover more languages.

```
1  "Language: ja"
2  "Plural-Forms: nplurals=1; plural=0;"
3
4  "Language: de"
5  "Plural-Forms: nplurals=2; plural=(n != 1);"
6
7  "Language: cs"
8  "Plural-Forms: nplurals=3; plural=(n==1 ? 0 : n>=2 && n<=4 ? 1 : 2);"
9
10 "Language: br"
11 "Plural-Forms: nplurals=5; plural=(n%10==1 && n%100!=11 && n%100!=71 && n%100!=91 ? 0 : n%10==2 && n%100!=12 &&
   n%100!=72 && n%100!=92 ? 1 : ((n%10>=3 && n%10<=4) || n%10==9) && (n%100<10 || n%100>19) && (n%100<70 || n%100>79)
   && (n%100<90 || n%100>99) ? 2 : n!=0 && n%1000000==0 ? 3 : 4);"
```



BOOST.LOCALE

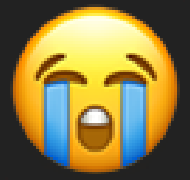
Among other things, it implements the message facet of std::locale

```
template <typename Char>
class basic_message {
    lots of constructors ...
    lots of conversions ...
    lots of write operations to streams ...
    ...
private:
    lots of translation and character encoding operations ...

    basic_string_view<Char> Singular;
    basic_string_view<Char> Plural;
    basic_string_view<Char> Context;
    long long               Cardinal;
};

lots of free function that create instances of basic_message
 - translate( ... )
 - gettext(), ngettext(), dgettext(), dngettext(), pgettext(), dpgettext()
```

brings additional overloads and helpers for Boost.Format

BOOST.LOCALE

it does too much!



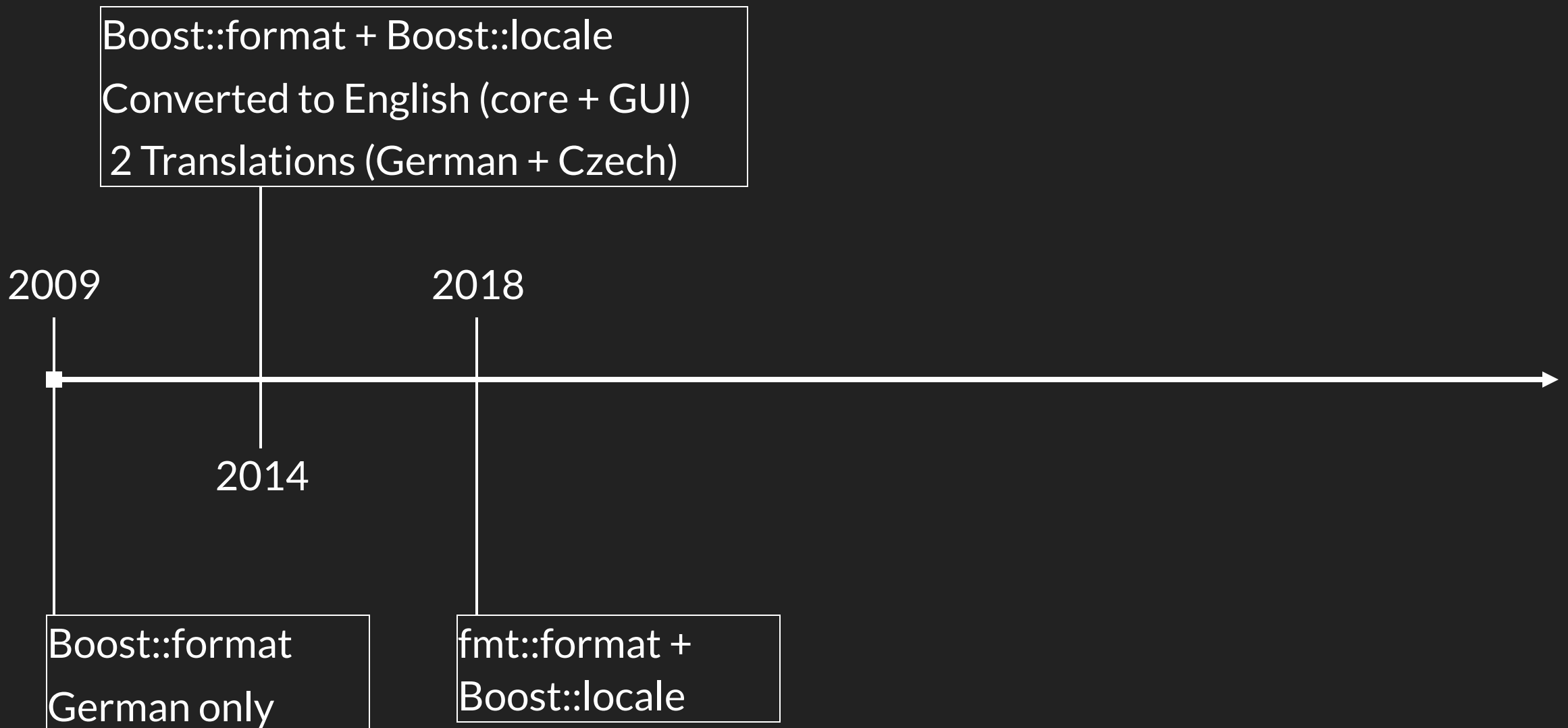
every return operation

- does a translation
- does an encoding conversion

therefore it requires a string allocation

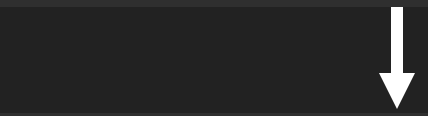
a new library

TIMELINE


Boost::format + Boost::locale
Converted to English (core + GUI)
2 Translations (German + Czech)
2009
2014
2018
Boost::format
German only
fmt::format +
Boost::locale

{FMT}

```cpp
#include <boost/format.hpp>

std::cout << boost::format("formatting specifier") % argument1 % argument2;
```



```cpp
#include <fmt/format.hpp>

std::cout << fmt::format("formatting specifier", argument1, argument2);
```

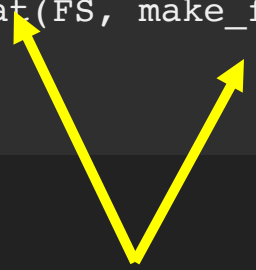
```cpp
template <typename Ch>
format(const something-related-to-Ch & FS) -> boost::basic_format<Ch>;
     +
template <typename Ch, typename T>
operator%(boost::basic_format<Char> &, const T &) -> boost::basic_format<Ch>;

vs.

template <typename T1, typename T2>
format(std::string_view FS, T1 && Arg1, T2 && Arg2) -> std::string;

template <typename T1, typename T2>
format(std::wstring_view FS, T1 && Arg1, T2 && Arg2) -> std::wstring;
```

{FMT} HAS TWO INTERFACES!

```cpp
#include <fmt/format.hpp>

// Interface 1

template <typename... Types>
format(std::string_view FS, Types &&... Args) -> std::string;


// Interface 2

vformat(std::string_view FS, format_args FA) -> std::string;


// Express interface 1 in terms of interface 2

template <typename... Types>
format(std::string_view FS, Types &&... Args -> std::string {
    return vformat(FS, make_format_args(Args));
}
     +
template <typename Context = format_context, typename... Args>
make_format_args(const Args &... args) -> format-arg-store <Context, Args...>;
```

TYPE ERASURE

```cpp
// Typeful interface

template <typename... Types>
format(std::string_view, Types &&... Args) -> std::string;


// Type-erased interface

vformat(std::string_view, format_args) -> std::string;


// the type-full interface is a thin wrapper around the type-erased interface

template <typename... Types>
format(std::string_view FS, Types &&... Args -> std::string {
    return vformat(FS, make_format_args(Args));
}
```



Type erasure, or better: type classification

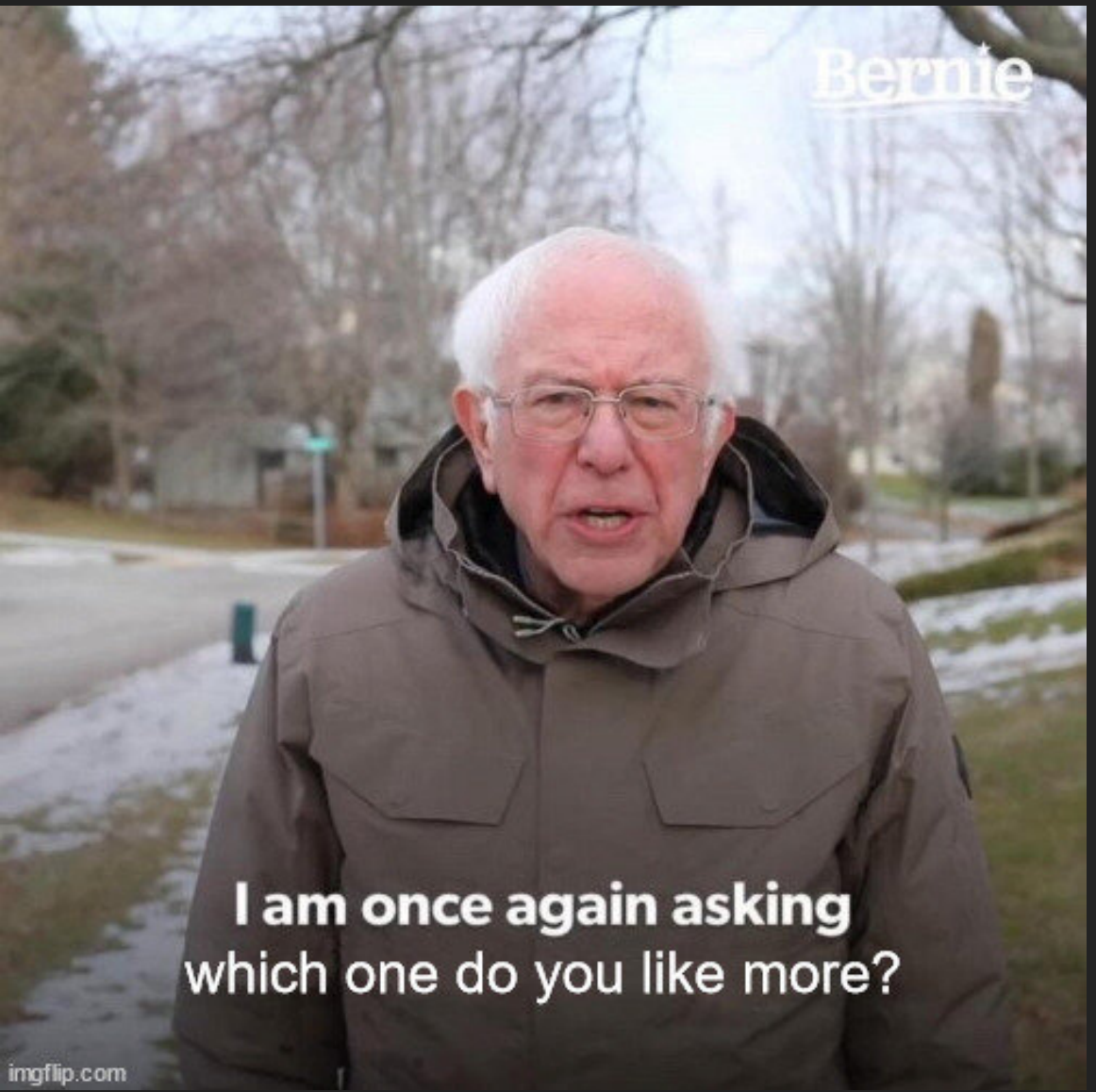
TYPE ERASURE

```cpp
#include <boost/format.hpp>

template <typename Ch>
format(const something-related-to-Ch &) -> boost::basic_format<Ch>;
```

```cpp
template <typename Ch, typename Traits, typename Allocator>
class basic_format {
    ...
    template <typename T>
    operator%(const T &) -> basic_format;
    ...

private:
    ...
    std::vector<format_item_t> items_; // each '%' directive leads to a format_item
    ...
};
```

Type erasure 

CHOICES...



Ficticiously replace ' operator%()' with 'operator,()'

```
#include <boost/format.hpp>

format("formatting specifier"), argument1, argument2;
```

```
#include <fmt/format.hpp>

format("formatting specifier", argument1, argument2);
```

TYPE ERASURE

{fmt} is using all kinds of type-based metaprogramming to achieve its goals:

- templates (type calculations)
- partial template specializations (i.e. "pattern matching")
- overload resolution and implicit conversion sequences of unimplemented functions (conversion-based metaprogramming)
- SFINAE ("substitution failure is not an error") to control overload sets and viability of functions

RULE OF Chiel Douwes 💜

COST OF OPERATIONS

- SFINAE
- Instantiating a function template
- Instantiating a type
- Calling an alias
- Adding a parameter to a type
- Adding a parameter to an alias call
- looking up a memoized type

AKA THE RULE OF CHIEL

WHAT IS A FUNCTION?

for every x from the definition domain,
there is a y from the value domain, with

let $y = f(x)$

$x \rightarrow y$

x can be from any domain,
so does y, it can be from a different domain

WHAT IS A FUNCTION?

the operator *f* describes any kind of mapping

anything,

not necessarily just one element

$$f: x \longrightarrow \begin{cases} y_0 & \text{if } x \text{ matches predicate 0} \\ y_1 & \text{if } x \text{ matches predicate 1} \\ y_2 & \text{if } x \text{ matches predicate 2} \end{cases}$$

anything else,

necessarily exactly one element

WHAT IS A FUNCTION?

operator f may have any number of arguments with a definite number of return values

$f()$	✅	$f() \longrightarrow (y_0, y_1, y_2, \ldots)$	❌
$f(x)$	✅	$f(x) \longrightarrow (y)$	✅
$f(x_0, x_1)$	✅	$f(x_0, x_1) \longrightarrow ()$	✅
$f(x_0, x_1, x_2)$	✅	$f(x_0, x_1, x_2) \longrightarrow (\ldots)$	❌
$f(x_0, x_1, x_2, \ldots)$	✅	$f(x_0, x_1, x_2, \ldots) \longrightarrow (y_0, y_1, y_2)$	✅
$f(\ldots)$	✅	$f(\ldots) \longrightarrow (y_0, y_1)$	✅

WHAT IS A FUNCTION?

names are unimportant as long as they are unique

f() ✅

g() ✅

ħ() ✅

aŭnt_Aňňŋi≠() ✅

💭() ✅

some unutterable name

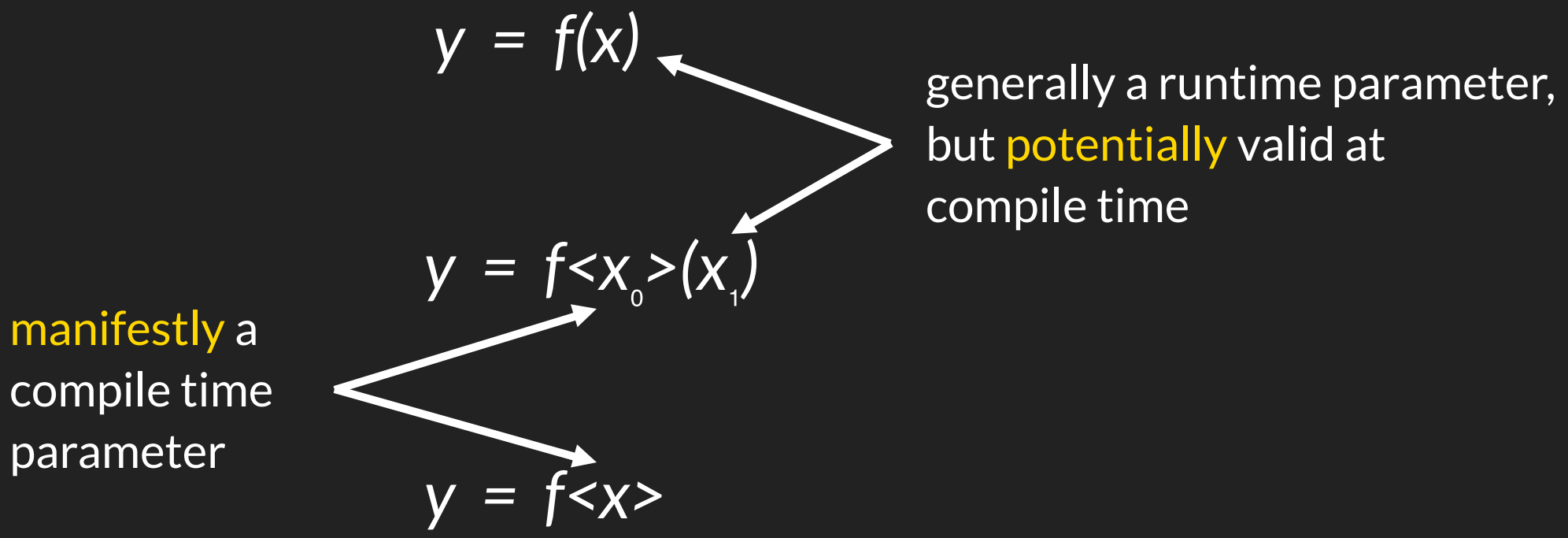
WHAT IS A FUNCTION?

spellings are unimportant, but they may have context

$y = f(x)$	✅	$x \longrightarrow y$
$y = f\{x\}$	✅	$x \longrightarrow y$
$y = f[x]$	✅	$x \longrightarrow y$
$y = f«x»$	✅	$x \longrightarrow y$
$y = f\langle x \rangle$	✅	$x \longrightarrow y$
$y = f\langle x_0 \rangle(x_1)$	✅	$x_0, x_1 \longrightarrow y$

WHAT IS A FUNCTION?

the context is important, in which parameters are valid



WHAT IS A FUNCTION?

functions can return any number of things of any kind

$$y = f\langle x \rangle$$

$$y = f\langle x_0, x_1, \ldots \rangle$$

x: type parameter, non-type parameter (integral, pointer, reference, enumerations, classes, templates, placeholders)

see http://eel.is/c++draft/temp.param and http://eel.is/c++draft/temp.arg

y: pretty much any C++ object, reference, and alias that you can e.g. use as class member, but also values

WHAT IS A FUNCTION?

the type traits from the standard library are your friend

std::is_object<T>	✅	T	$\longrightarrow$	*bool*
std::is_convertible<From, To>	✅	T_0, T_1	$\longrightarrow$	*bool*
std::remove_cvref<T>	✅	T	$\longrightarrow$	*T '*
std::common_type<T_0, T_1>	✅	T_0, T_1	$\longrightarrow$	*U*
std::invoke_result<f<Ts...>>	✅	*f, Ts...*	$\longrightarrow$	*U*
std::void_t<Ts...>	✅	*Ts...*	$\longrightarrow$	*void*
std::vector<T>	✅	*U, T*	$\longrightarrow$	*V*

WHAT IS A FUNCTION?

there are multiple ways to say the same

```
template<T>
struct S { enum bool { value = ...}; };
```

✅ $T \longrightarrow bool$

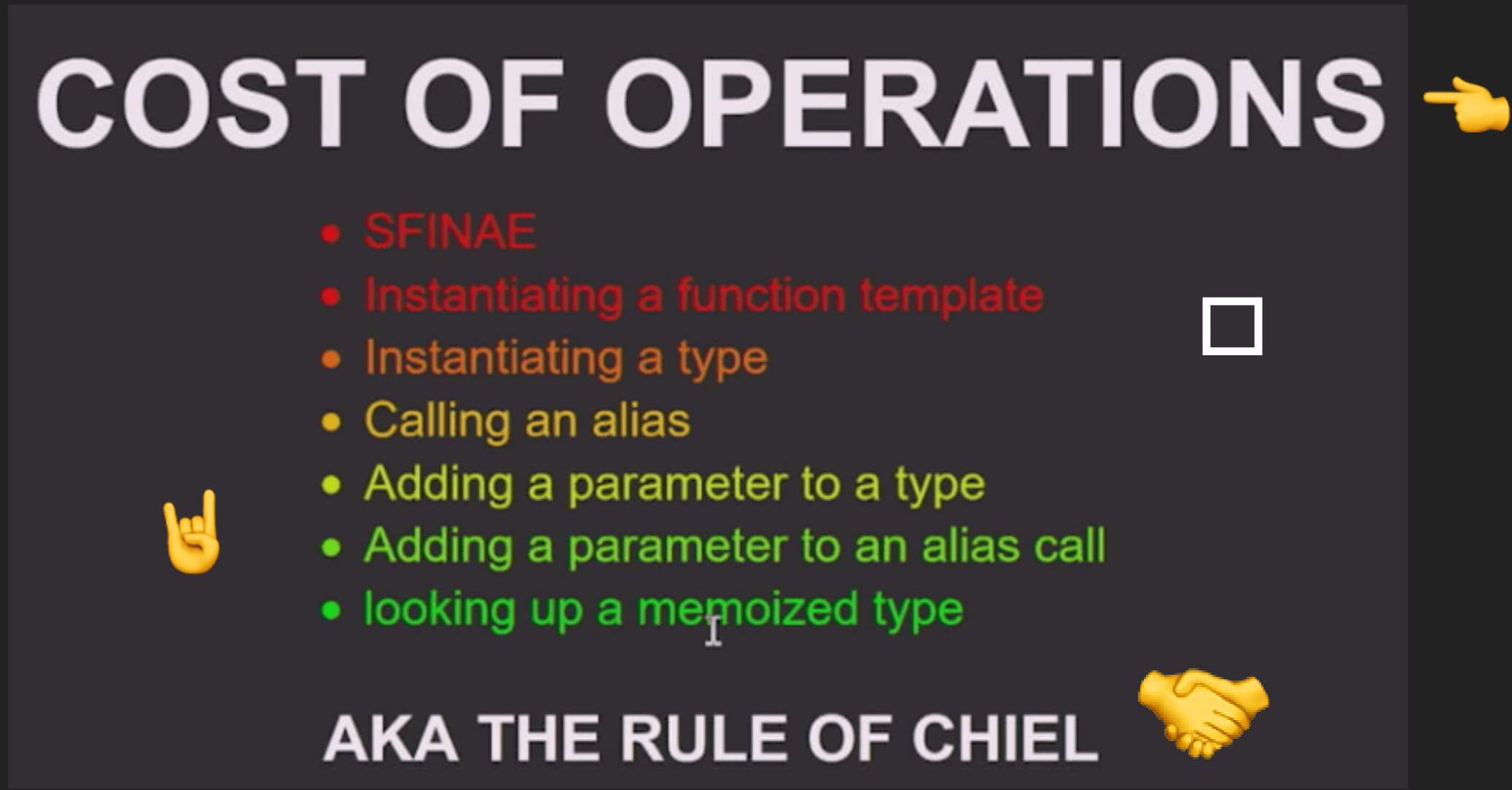
```
template<T>
constexpr bool value = ...;
```

✅ $T \longrightarrow bool$

```
template<T>
std::bool_constant value(T) noexcept { ... };
```

✅ $T \longrightarrow bool$

ONCE MORE...


COST OF OPERATIONS
SFINAE
Instantiating a function template
Instantiating a type
Calling an alias
Adding a parameter to a type
Adding a parameter to an alias call
looking up a memoized type
AKA THE RULE OF CHIEL

ONE MORE THING...

Boost.MP11

~~Boost.MPL~~

Kvasir

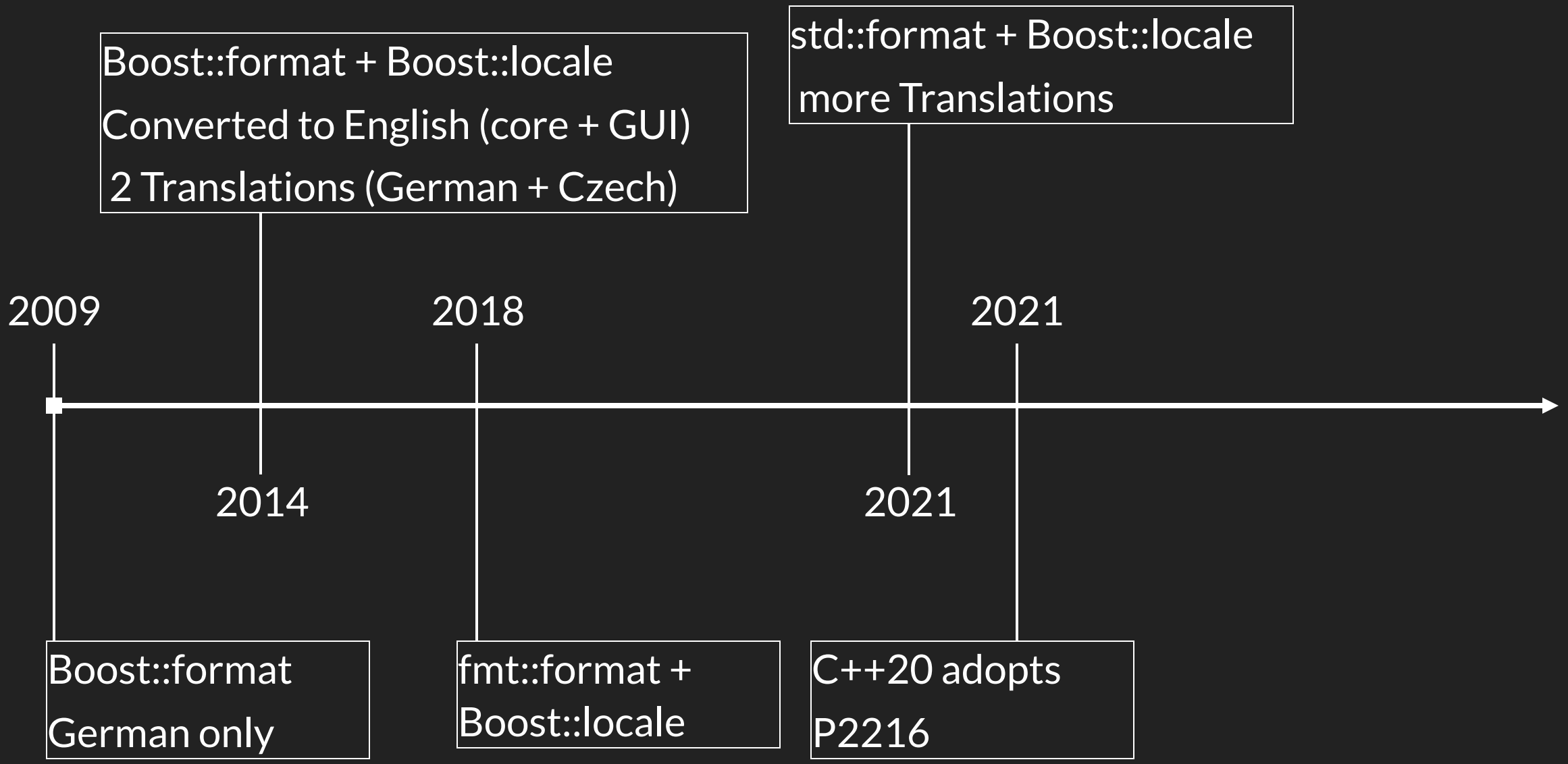
use libraries!

Metal

Brigand

a new world

TIMELINE


Boost::format + Boost::locale
Converted to English (core + GUI)
2 Translations (German + Czech)
std::format + Boost::locale
more Translations
2009
2018
2021
2014
2021
Boost::format
German only
fmt::format +
Boost::locale
C++20 adopts
P2216

FROM C++20 TO C++20

Original C++20 (Prague 2020, N4860)

```
template<typename ... Types>
auto std::format(std::string_view Fmt, const Args &... Args) -> std::string {
    return std::vformat(Fmt, std::make_format_args(Args ...));
}
```

Today's C++20 (April 2024, N4981)

```
template<typename ... Types>
auto std::format(std::format_string<Types...> Fmt, Args &&... Args) -> std::string {
    return std::vformat(Fmt.get(), std::make_format_args(Args ...));
}

template <typename ... Types>
struct format_string {
    consteval format_string(std::string_view Str) : Str_(Str) {}

    constexpr std::string_view get() const noexcept { return Str_; }

private:
    std::string_view Str_;
};
```

CONSTEVAL

P2216

2020:

- the std::string_view was potentially evaluated (and constructed) at compile time
- in most cases compile time evaluation succeeds

2021:

- the std::format_string is necessarily evaluated and constructed at compile time
- and so is the std::string_view that it is constructed from

- the arguments to the constructor of std::string_view must be known at compile-time!
- for more, please see http://eel.is/c++draft/expr.const#16

P2216

```cpp
template<typename... Types>
auto std::format(std::format_string<Types...> Fmt, Args &&...) -> std::string {}

template <typename... Types>
struct format_string {
    consteval format_string(std::string_view Str) : {
        constexpr size_t num_args = sizeof...(Types);
        constexpr basic_format_arg_type arg_types[num_args > 0 ? num_args : 1] = {
            std::get_format_arg_type<Types>()...
        };

        parse_format_string(Str, format_checker<std::remove_cvref_t<Types>...>{Str, arg_types});
    }
};
```

format string syntax and argument type checking can - and will - be done at compile time

no more exceptions at runtime!

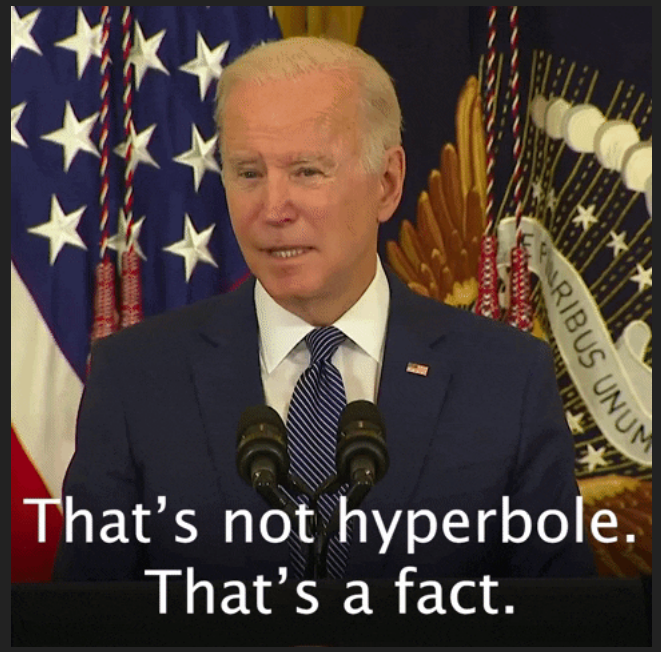
WHAT IS CONSTANT EVALUATION?

During compilation, the compiler has to remember everything it has seen so far:

- identifiers
- entities
- declarations
- definitions
- ...
- templates
- all template instantiations so far

Pretty much everything.

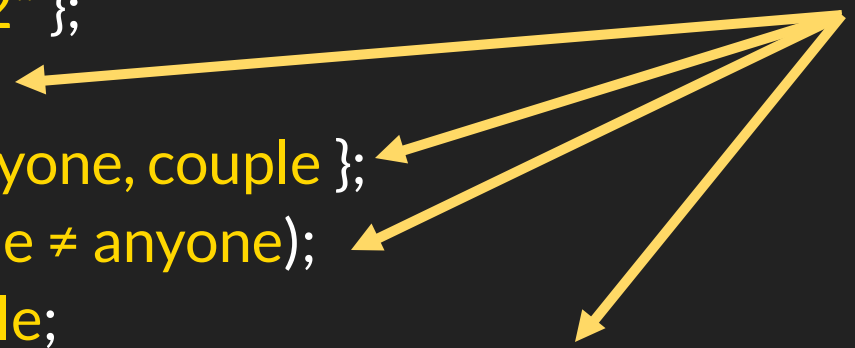
BTW, this is why we have modules now 🎉



WHAT IS CONSTANT EVALUATION?

But what about those?

- int i = 1 + 2 + ¾;
- bool b{ 42 == L"42" };
- char a[] = "huh?";
- enum e { none, anyone, couple };
- static_assert(none ≠ anyone);
- const int c = couple;
- std::vector v = { none * 2, anyone * sizeof(a), couple * c };
- ...

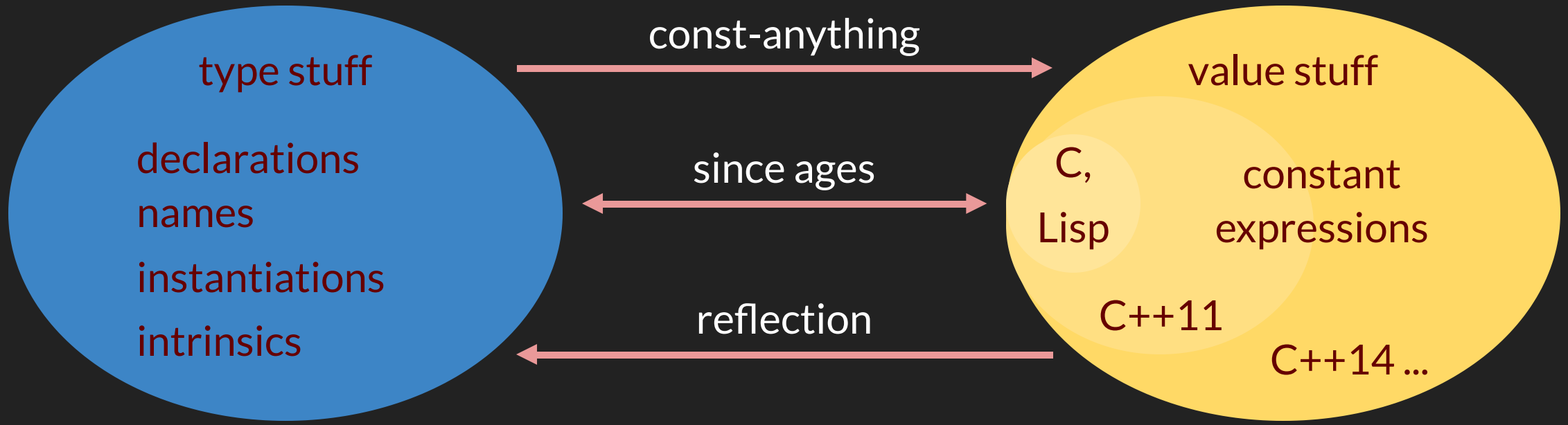


constant required,

"constant expression"

SPLIT BRAIN

There are actually two subsystems in the compiler that handle compile-time entities



entity table

constant evaluator

CONSTANT EXPRESSION

Wherever a "constant" is required , a

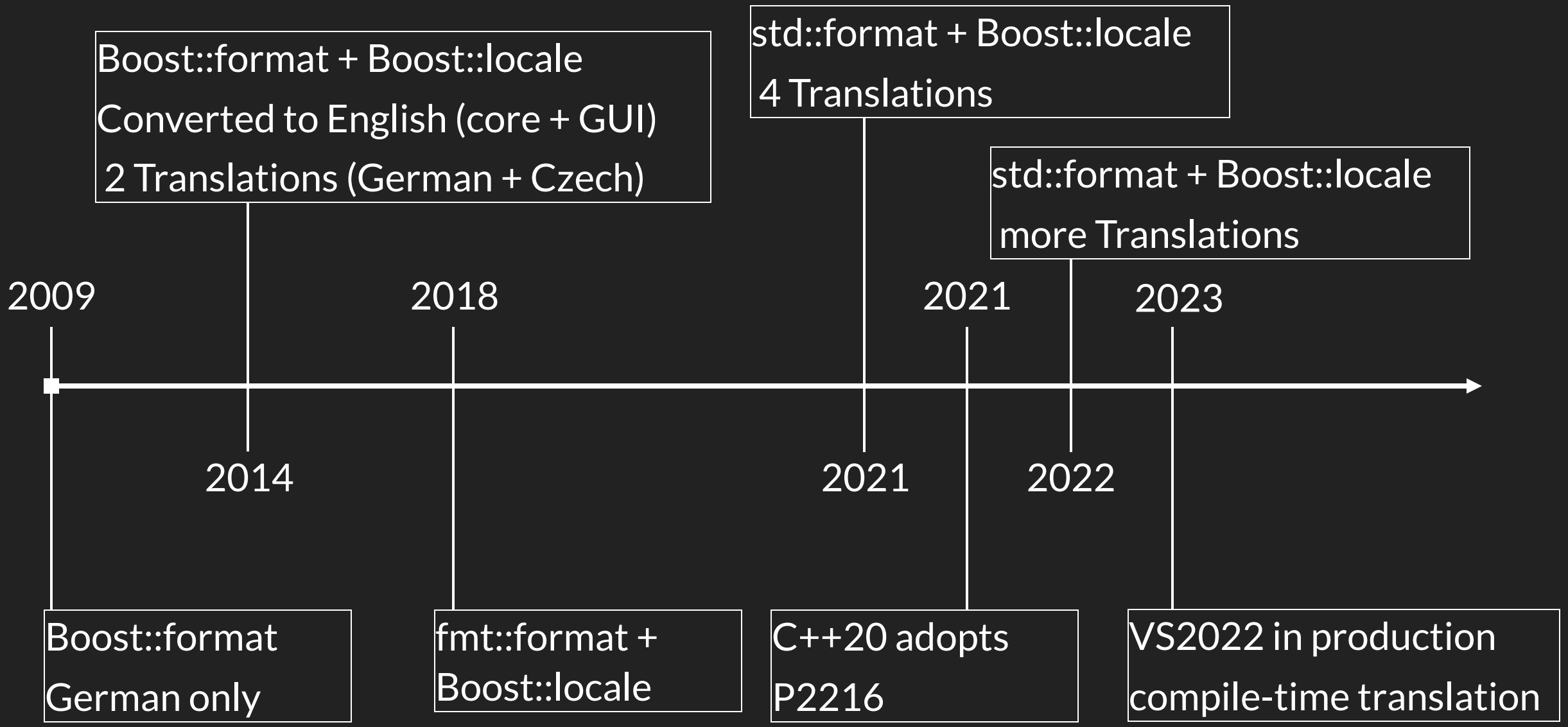
new constant evaluation

is started

even if you are already within a constant evaluation!

what now?

TIMELINE


Boost::format + Boost::locale
Converted to English (core + GUI)
2 Translations (German + Czech)
std::format + Boost::locale
4 Translations
std::format + Boost::locale
more Translations
2009
2014
2018
2021
2021
2022
2023
Boost::format
German only
fmt::format +
Boost::locale
C++20 adopts
P2216
VS2022 in production
compile-time translation

P2216 FALLOUT

~~Boost::locale~~

at least,

the translation frontend with
runtime translation is no longer a thing

WHAT NOW?

a closer look

```
1 struct format_string {
2     consteval
3     format_string(const convertible_to<std::string_view> & Str) : Str_(Str) {}
4
5     constexpr
6     std::string_view get() const noexcept { return Str_; }
7 };
```

MO' OVERLOADS

```
template <typename... Types>
auto format(const format_string_translator<Types...> XFmt, Types &&... Args) -> std::string {
    if constexpr (sizeof...(Args) > 0) {
        const auto Quantity = XFmt.Quantity( ... something with Args );
        return std::vformat(XFmt.get(Quantity),
                            make_format_args(
                                wrapped<std::remove_cvref_t<Types>>::translate(Args)...
                            ));
    } else {
        return std::vformat(XFmt.get(), {});
    }
}
```

FORMAT_STRING_TRANSLATOR

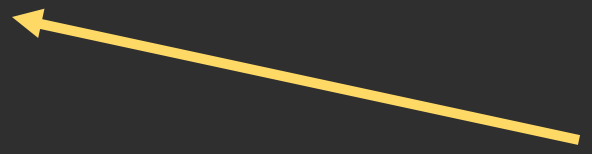
```
1  template <typename... Types>
2  auto format(const format_string_translator<Types...>, Types &&...) -> std::string;
3
4  template <typename... Types>
5  struct format_string_translator : basic_translator {                    // empty class!
6      using base = format_string<Types...>;
7
8      static constexpr auto numPluralArguments = (isMarkedAsPlural<Types> + ... + 0);
9      static_assert(numPluralArguments <= 1, "Oops, more than one plural argument was found!");
10
11     consteval format_string_translator(const tTranslate & Tr)      // almost all work is done here
12     : basic_translator(Tr) {
13         base{ Tr.Singular() }, base{ Tr.Plural() }, checkPlural(Tr.lenPlural_, numPluralArguments);
14     }
15 };
16
17 consteval void checkPlural(std::size_t gotPluralFormat, std::size_t havePluralArguments) {
18     if (gotPluralFormat and not havePluralArguments)
19         throw "Sorry, a plural format string is present, but no plural argument was found!";
20     if (havePluralArguments and not gotPluralFormat)
21         throw "Sorry, a plural argument was found, but no plural format string is present!";
22 }
23
24 template <typename T>
25 constexpr bool isMarkedAsPlural = std::is_same_v<plural, T>;
```

BASIC_TRANSLATOR

```
template <typename... Types>
auto format(const format_string_translator<Types...>, Types &&...) -> std::string;

template <typename... Types>
struct format_string_translator : basic_translator;  // knows argument types, does syntax checking

struct basic_translator {   // knows only string digest, and how to translate from actual cardinal
    consteval basic_translator(const tTranslate & Tr)
    : Translator_{ markPluralsPresent(Tr.Digest_, Tr.lenPlural_ > 0) } {}

    std::string_view get(plural::type N) const noexcept {
        return havePlurals(Translator_.Digest_) ? Translator_.multiple(N) : Translator_.single();
    }

    std::string_view get() const noexcept { return Translator_.single(); }

private:
    tBaseTranslate Translator_;                 // strips off all strings, knows only the digest
};

constexpr bool havePlurals(const uint64_t Digest) { ... }

consteval uint64_t markPluralsPresent(const uint64_t Digest, const bool havePlural) { ... }
```

BASIC_TRANSLATOR

```
1  template <typename... Types>
2  auto format(const format_string_translator<Types...>, Types &&...) -> std::string;
3
4  template <typename... Types>
5  struct format_string_translator : basic_translator;  // knows argument types, does syntax checking
6
7  struct basic_translator {}; // knows only string digest, and how to translate from actual cardinal
8
9  struct tBaseTranslate {                              // perform translation from compile-time digest
10     constexpr auto single() const noexcept -> std::string_view {
11         const auto & Maps = ...          // from backend;
12         return lookup(Maps, Digest_);    // middle: wed frontend and backend □
13     }
14
15     constexpr auto multiple(plural::type N) const noexcept -> std::string_view {
16         const auto & Maps = ...                                // from backend;
17         return lookup(Maps, Digest_, std::uint64_t{ N });  // middle: wed frontend and backend □
18     }
19
20     ... more overloads of 'single()' and 'multiple(N)'
21
22     std::uint64_t Digest_;   // constructed by frontend at compile-time
23 };
```



the only runtime code

3 QUESTIONS

```
1 template <typename... Types>
2 auto format(const format_string_translator<Types...> XFmt, Types &&... Args) -> std::string {
3     const auto Quantity = XFmt.Quantity( ... something with Args);
4     return std::vformat(XFmt.get(Quantity),
5                         make_format_args(wrapped<std::remove_cvref_t<Types>>::translate(Args)...));
6 }
```

1️⃣,2️⃣

3️⃣

1️⃣ which argument holds the cardinal that determines the language form?

2️⃣ how to access its current value?

3️⃣ how to figure out how a given argument is translated? Or is it at all?

P2663 (PACK INDEXING) + P2996 (REFLECTION)

```
1 template <typename... Types>
2 auto format(const format_string_translator<Types...> XFmt, Types &&... Args) -> std::string {
3     constexpr auto Index = std::ranges::find_if({ Args^... }, isPluralType) - &Args...[0]^;
4     return std::vformat(XFmt.get(Args...[Index]),
5                         make_format_args( [... something with Args^... ...]));
6 }
```

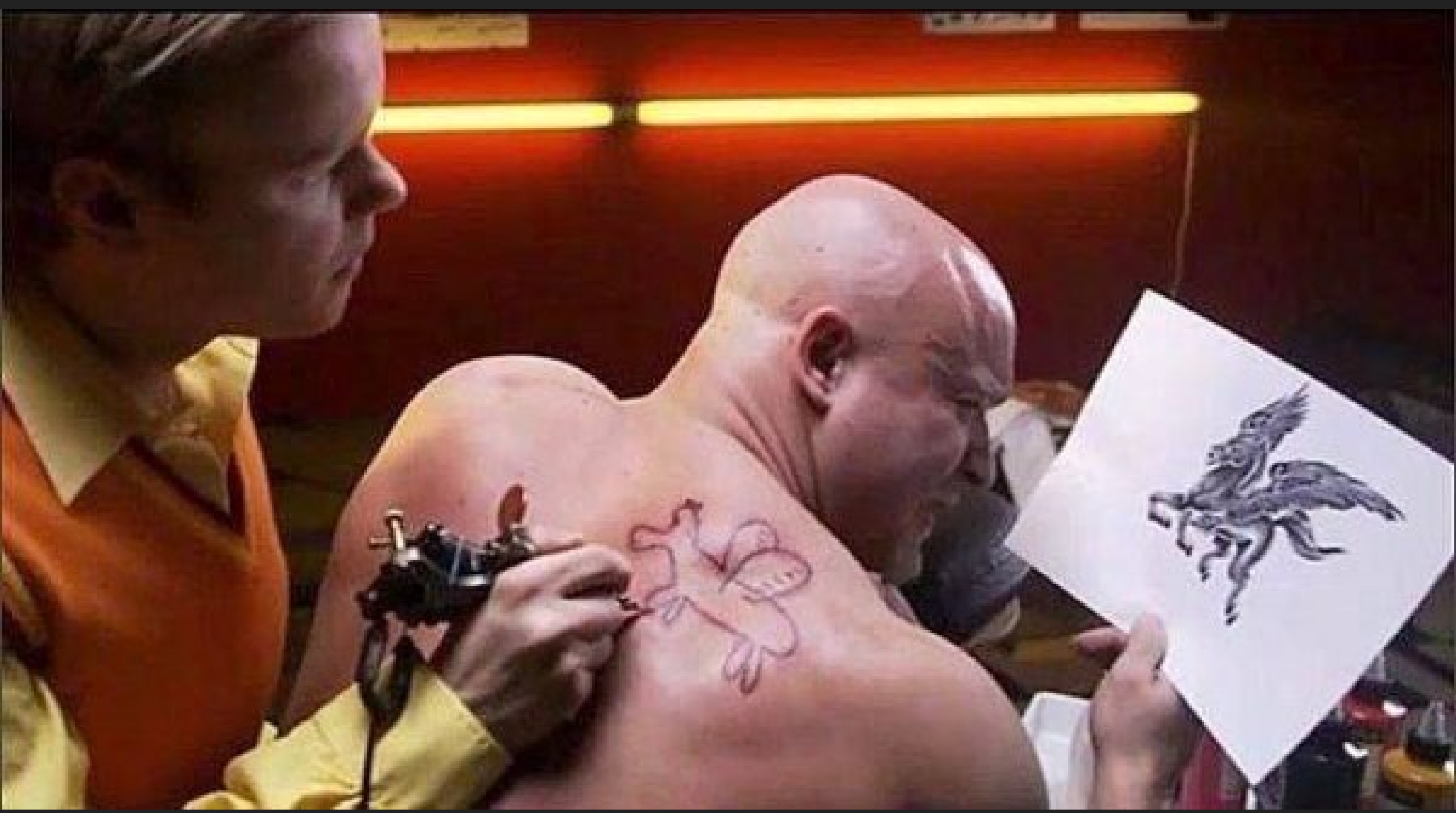
2 1 3

1️⃣ which argument holds the cardinal that determines the language form?

2️⃣ how to access its corrent value?

3️⃣ how to figure out how a given type is translated? Or is it at all?

Unfortunately, this is 2024. C++26 is not a thing yet 🤯



WHAT IS THE PLURAL INDEX?

For homogenous sequences you need two things:

- the proper algorithm, e.g. how to go over the elements in a sequence
- the proper predicate, e.g. how to determine a property of a element

use the sequence algorithms from the standard library!

But what about heterogenous sequences, like e.g. type-lists?

HETEROGENEOUS SEQUENCES

There is this paper by Graham Hutton

A tutorial on the universality and expressiveness of fold

in the Journal of Functional Programming, 1999

SOME EXAMPLES

homogeneous

e.g. vector v of values

count(v) → size_t

count_if(v, pred) → size_t

count_if(v, pred, proj) → size_t

for_each(v, func) → void

for_each(v, func, proj) → void

all(v, pred) → bool

any(v, pred) → bool

none(v, pred) → bool

heterogeneous

e.g. type list <typename... Ts>

size...(Ts) → size_t

(pred<Ts> + ... + 0) → size_t

(pred<proj<Ts>> + ... + 0) → size_t

(func<Ts>, ...) → void

(func<proj<Ts>>, ...) → void

(pred<Ts> and ...) → bool

(pred<Ts> or ...) → bool

((not pred<Ts>) and ...) → bool

1️⃣ FIND_IF

But I need std::ranges::find_if on a type list!

```
1  static constexpr auto noIndex = std::size_t{ 0 } - 1;
2
3  template <typename T>                              // the predicate,  T -> bool
4  struct isMarkedAsPlural {                          // needs to be wrapped in a class for reasons
5      constexpr operator bool() const noexcept { return std::is_same_v<plural, T>; }
6  };
7
8  template <typename... Types>
9  struct format_string_translator {
10     static constexpr auto numPluralArguments = (isMarkedAsPlural<Types>{} + ... + 0);
11     static_assert(numPluralArguments <= 1, "Oops, more than one plural argument was found!");
12
13     static constexpr auto PluralIndex = findFirstIndex<isMarkedAsPlural, Types...>();
14     static_assert(numPluralArguments > 0 ? PluralIndex < sizeof...(Types) : PluralIndex == noIndex);
15 };
```

FINDFIRSTINDEX

```
1  static constexpr auto noIndex = std::size_t{ 0 } - 1;
2
3  template <typename T>
4  struct isMarkedAsPlural {
5      constexpr operator bool() const noexcept { return std::is_same_v<plural, T>; }
6  };
7
8
9  constexpr std::size_t findFirstIndex() {
10     std::size_t Result = noIndex;
11
12     return Result;
13 }
14
15 static_assert(findFirstIndex() == noIndex);
```

FINDFIRSTINDEX

```
1  static constexpr auto noIndex = std::size_t{ 0 } - 1;
2
3  template <typename T>
4  struct isMarkedAsPlural {
5      constexpr operator bool() const noexcept { return std::is_same_v<plural, T>; }
6  };
7
8
9  constexpr std::size_t findFirstIndex() {
10     std::size_t Result = noIndex + 1;
11
12     return Result - 1;
13 }
14
15 static_assert(findFirstIndex() == noIndex);
```

FINDFIRSTINDEX

```cpp
static constexpr auto noIndex = std::size_t{ 0 } - 1;

template <typename T>
struct isMarkedAsPlural {
    constexpr operator bool() const noexcept { return std::is_same_v<plural, T>; }
};

template <template <typename T> typename Predicate, typename... Types>
constexpr std::size_t findFirstIndex() {
    std::size_t Result = noIndex + 1;

    return (false or ... or Predicate<Types>{}),
           Result - 1;
}

static_assert(findFirstIndex<isMarkedAsPlural>()        ==  noIndex);
static_assert(findFirstIndex<isMarkedAsPlural, void>()  ==  noIndex);
static_assert(findFirstIndex<isMarkedAsPlural, plural, void>() == noIndex);
static_assert(findFirstIndex<isMarkedAsPlural, void, plural>() == noIndex);
```

must be a type because of http://eel.is/c++draft/temp.arg.template

FINDFIRSTINDEX

```
1  static constexpr auto noIndex = std::size_t{ 0 } - 1;
2
3  template <typename T>
4  struct isMarkedAsPlural {
5      constexpr operator bool() const noexcept { return std::is_same_v<plural, T>; }
6  };
7
8  template <template <typename T> typename Predicate, typename... Types>
9  constexpr std::size_t findFirstIndex() {
10     std::size_t Result = noIndex + 1;
11
12     return (false or ... or (Predicate<Types>{} ? true : false)),
13            Result - 1;
14 }
15
16 static_assert(findFirstIndex<isMarkedAsPlural>()        ==  noIndex);
17 static_assert(findFirstIndex<isMarkedAsPlural, void>()  ==  noIndex);
18 static_assert(findFirstIndex<isMarkedAsPlural, plural, void>() == noIndex);
19 static_assert(findFirstIndex<isMarkedAsPlural, void, plural>() == noIndex);
```

FINDFIRSTINDEX

```cpp
static constexpr auto noIndex = std::size_t{ 0 } - 1;

template <typename T>
struct isMarkedAsPlural {
    constexpr operator bool() const noexcept { return std::is_same_v<plural, T>; }
};

template <template <typename T> typename Predicate, typename... Types>
constexpr std::size_t findFirstIndex() {
    std::size_t Result = noIndex + 1;
    std::size_t Index  = 0;

    return (false or ... or (++Index, Predicate<Types>{} ? true : false)),
           Result - 1;
}

static_assert(findFirstIndex<isMarkedAsPlural>()        ==  noIndex);
static_assert(findFirstIndex<isMarkedAsPlural, void>()  ==  noIndex);
static_assert(findFirstIndex<isMarkedAsPlural, plural, void>() == noIndex);
static_assert(findFirstIndex<isMarkedAsPlural, void, plural>() == noIndex);
```

FINDFIRSTINDEX

```cpp
static constexpr auto noIndex = std::size_t{ 0 } - 1;

template <typename T>
struct isMarkedAsPlural {
    constexpr operator bool() const noexcept { return std::is_same_v<plural, T>; }
};

template <template <typename T> typename Predicate, typename... Types>
constexpr std::size_t findFirstIndex() {
    std::size_t Result = noIndex + 1;
    std::size_t Index  = 0;

    return (false or ... or (++Index, Predicate<Types>{} ? Result = Index : false)),
           Result - 1;
}

static_assert(findFirstIndex<isMarkedAsPlural>()        ==  noIndex);
static_assert(findFirstIndex<isMarkedAsPlural, void>()  ==  noIndex);
static_assert(findFirstIndex<isMarkedAsPlural, plural, void>() == 0);
static_assert(findFirstIndex<isMarkedAsPlural, void, plural>() == 1);
```

2️⃣ INDEX ARGUMENT

```
1  template <typename... Types>
2  std::string format(const format_string_translator<Types...> XFmt, Types &&... Args) {
3      const auto Quantity = XFmt.Quantity(
4          { reinterpret_cast<uintptr_t>(std::addressof(Args))... }
5      );
6  }
7
8  template <typename... Types>
9  struct format_string_translator {
10     static constexpr auto PluralIndex = findFirstIndex<isMarkedAsPlural, Types...>();
11
12     static plural::type Quantity(const uintptr_t (&pArgs)[]) noexcept {
13         if constexpr (PluralIndex != noIndex)
14             return *std::bit_cast<const plural *>(pArgs[PluralIndex]);
15         else
16             return 1;
17     }
18 };
```

This is valid code, no undefined behaviour here! 😊

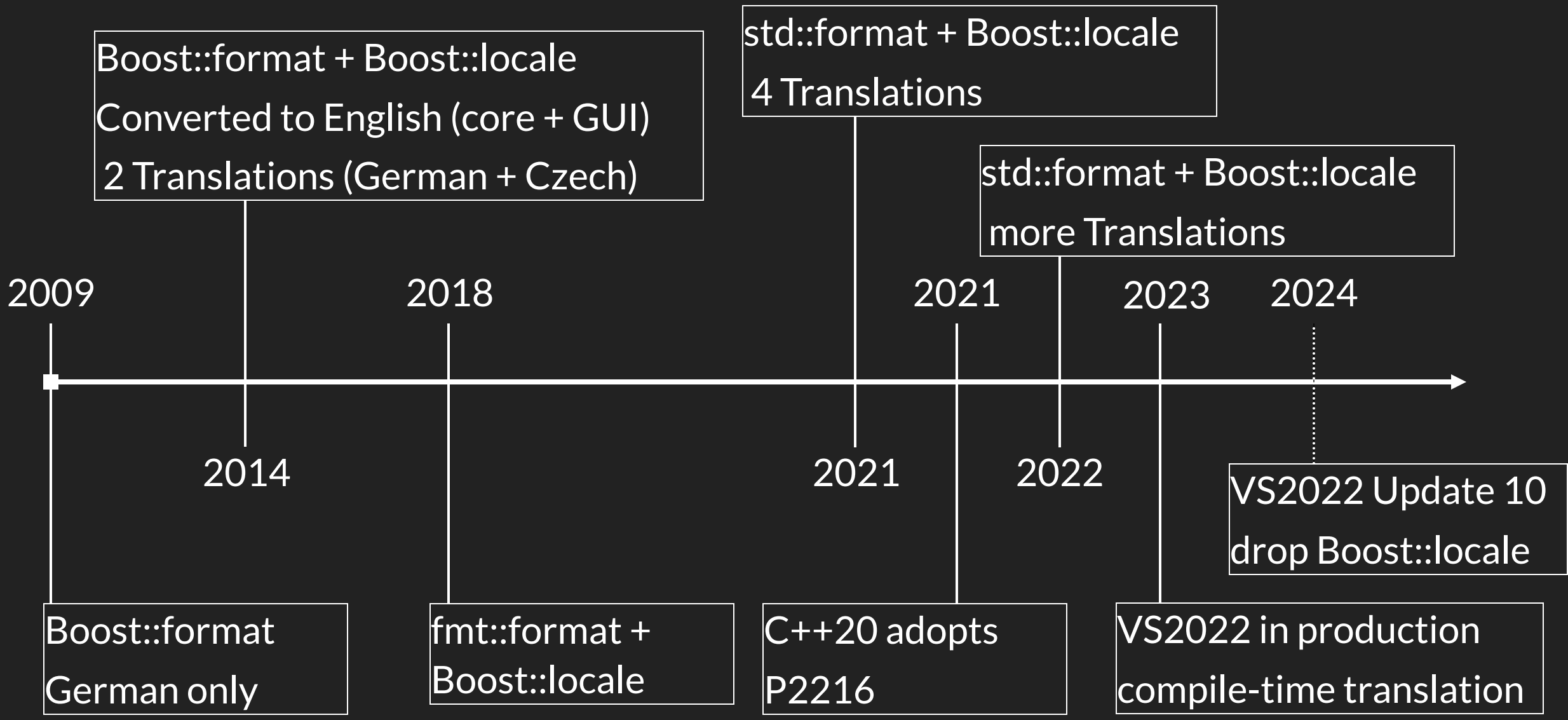
3️⃣ TYPE-BASED SELECTION

a.k.a. pattern-matching (P2688)

```cpp
template <typename T>
struct wrapped {
    using type = T;
    static constexpr const type & translate(const T & t) { return t; }
};

template <typename Char>
struct wrapped<tTranslate<Char>> {
    using type = std::basic_string_view<Char>;
    static constexpr type translate(const tBaseTranslate<Char> & t) { return t.single(); }
};

template <>
struct wrapped<plural> {
    using type = plural::type;
    static constexpr type translate(const plural & t) { return t; }
};
```

an
experiment

TIMELINE


Boost::format + Boost::locale
Converted to English (core + GUI)
2 Translations (German + Czech)
std::format + Boost::locale
4 Translations
std::format + Boost::locale
more Translations
2009
2018
2021
2023
2024
2014
2021
2022
VS2022 Update 10
drop Boost::locale
Boost::format
German only
fmt::format +
Boost::locale
C++20 adopts
P2216
VS2022 in production
compile-time translation

INTEGRATION TEST

```
#ifndef __cpp_lib_modules
#define __cpp_lib_modules 202207L  // use the modularized standard library 😊
#endif

#include "Translate.hpp"   // bring in the library: expose more than is exposed from the module
import utility;            // bring in a helper: a module, because why not

using namespace Translate; // slideware!

static constexpr auto Breton = utility::embed(
#if __has_include("br.bin")
#   include "br.bin"       // bring in the Breton translations (≈ 500) from the compiled .mo
#endif                     // this is a LOOONG compile-time constant (≈ 40 kBytes)
    "");                   // make it valid syntax even when it doesn't exist yet
```

INTEGRATION TEST

```
1  static constexpr auto Breton = "...";
2
3  using IntegrationDomain = tDomain<TranslationDomain("Integration")>;
4
5  static_assert(
6      [Tr = tTranslator(IntegrationDomain{})  // the frontend is in the λ capture
7          .load(LanguageId("br"), Breton)]    // load and process the Breton translations
8      (uint64_t Cardinal) {                   // pass in the determining cardinal value
9
10         return wlookup(Tr.DefaultLanguage(),
11                        translate(L"IntegrationContext", "Singular", "Plural", use_plural)
12                            .Digest_,        // the 64-bit representation of all three strings!
13                        Cardinal);
14
15     } (0)                                   // the cardinal 0 in the closure call
16     ==                                      // translates to
17     L"Language Form 4"                      // language form 4, UTF-16 because why not?
18 );
19
20 int main() {}
```

DEMO CODE TIME

LOAD

```
1 struct tTranslator {
2     constexpr tTranslator(const tTranslationDomain) { ... }
3     constexpr tTranslator & load(const tLanguageId, const std::span<const char>) { ... }
4     ...
5 };
```

- checks the .MO header for validity
- checks the two string tables for validity
- locates the meta info strings
- checks them for validity and UTF-8 conformance
- locates the plural rules in the meta info
- locates in each translation (source)
  - the singular
  - the optional context
  - the optional plural

  and calculates a 64 bit digest
- determines the uniqueness of all digests
- locates in each translation all language forms present there (which may differ)
- checks the validity and UTF-8 conformance of each language form
- builds a string table with all forms
- builds a lookup-table from all digests, optimized for cache-accesses
- each translation entry points to the first correspondent language form in the string table

LOAD

```
1 struct tTranslator {
2     ...
3     constexpr tTranslator & load(const tLanguageId, const std::span<const char>) { ... }
4     ...
5 };
```

- create a second string table
  - same size
  - UTF-16 instead of UTF-8
  - different posititions for each language form
- checks the plural rule for syntax and well-formedness
- parses into terminals and tokens
- checks for grammar rules
- builds an abstract syntax tree (AST)
- optimizes the AST for pattern matching
- creates a sequence of fixed-size opcodes (kind of VLIW)
- optimizes the generated program

In the end, we have (per language ID)

- one map: digest → 1st string index
- two maps: string index → language form string
- a list with the pattern matcher program

std::vector + std::string + std::span + views

TRANSLATE

```
constexpr auto translate(string-like auto Context,
    std::string_view Singular, std::string_view Plural) -> tTranslate<char>;

constexpr auto translate(string-like auto Context,
    std::wstring_view Singular, std::wstring_view Plural) -> tTranslate<wchar_t> {
    return {
        {
            .Digest_ = hash(0, Context, ContextSeparator, Singular),
        },
        ...
    };
}

template <typename Char>
struct tBaseTranslate {
    uint64_t Digest_;        // the 64-bit representation of all strings
};
```

'string-like' is a concept

LOOKUP

```
1 auto  lookup(const Maps &, uint64_t Digest, uint64_t Cardinal) -> std::string_view;
2 auto wlookup(const Maps &, uint64_t Digest, uint64_t Cardinal) -> std::wstring_view;
```

- looks for the given Digest in the 1st map
  - gets string index or 'invalid'
- executes the precompiled program in a virtual machine, with the Cardinal as input
  - gets the matched language form as plural index
- returns the final string_view, taking the string index + plural index as final index into the 2nd map

BRETON

- 62 terminals
- 57 operations
  - arithmetic
  - comparison
  - logical
- VLIW operation (32 bits), encoding
  - up to 1 arithmetic operation + 1 exponentiation
  - up to 3 constants
  - up to 1 comparison operation (all 8 possibilities)
  - up to 1 logical operation
  - up to 2 stack pushes
  - up to 2 stack pops
  - up to 1 return
- up to 21 bytecode operations until match

TOTAL

- ≈ 500 translations (complete machine business logic)
- a couple more language forms
- lookup table size
  - 1024 entries
  - 8 bytes per entry
  - 8 entries per cacheline

More than 2 million and less than 3 million execution steps in the constant evaluator to load the compiled MO file.

move on

RESOURCES

- Living, up-to-date C++ standard (currently at C++26)
- The Journal of Functional Programming, Cambridge University Press
- GNU gettext utilities
- Unicode CLDR

- Library code github.com/DanielaE/t.b.d.

Contact

dani@ngrt.de

@DanielaKEngert@hachyderm.io

DanielaE

Images: Maria Sibylla Merian (1705)

source: WikiMedia Commons, public domain , Unsplash & GIPHY

Ceterum censeo ABI esse frangendam