SO, YOU WANT TO USE C++MODULES ...

CROSS-PLATFORM?

Daniela Engert - Meeting C++ 2023

Outline

- Modules Recap
- Compilers
- Modularization
- Build Systems
- Module Dependencies
- Demo
- Conclusion

ABOUT ME

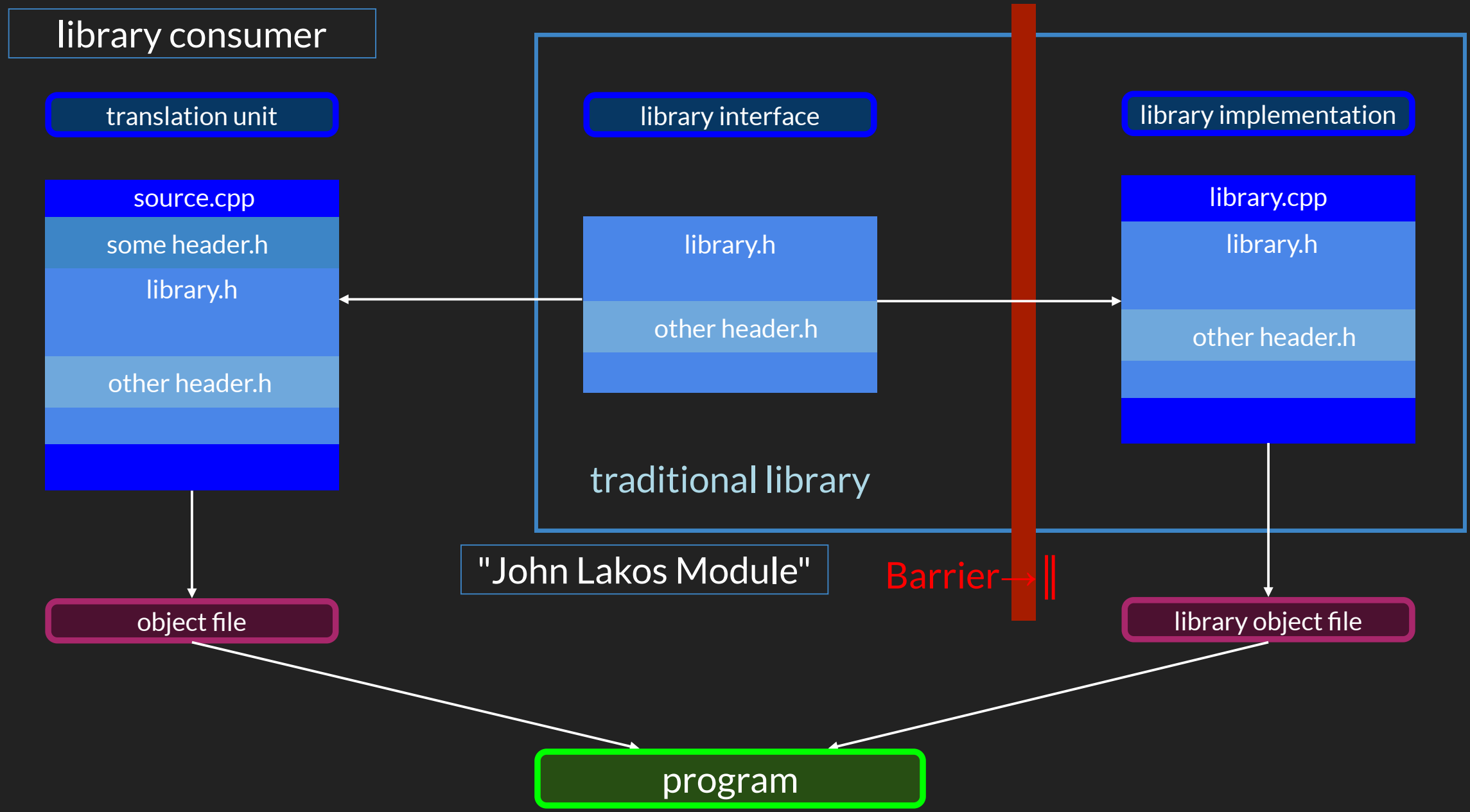
- Electrical engineer
- Build computers and create software for 40 years
- Develop hardware and software in the field of applied digital signal processing for 30 years
- Member of the C++ committee (learning novice) for 3 years (EWG, SG15)

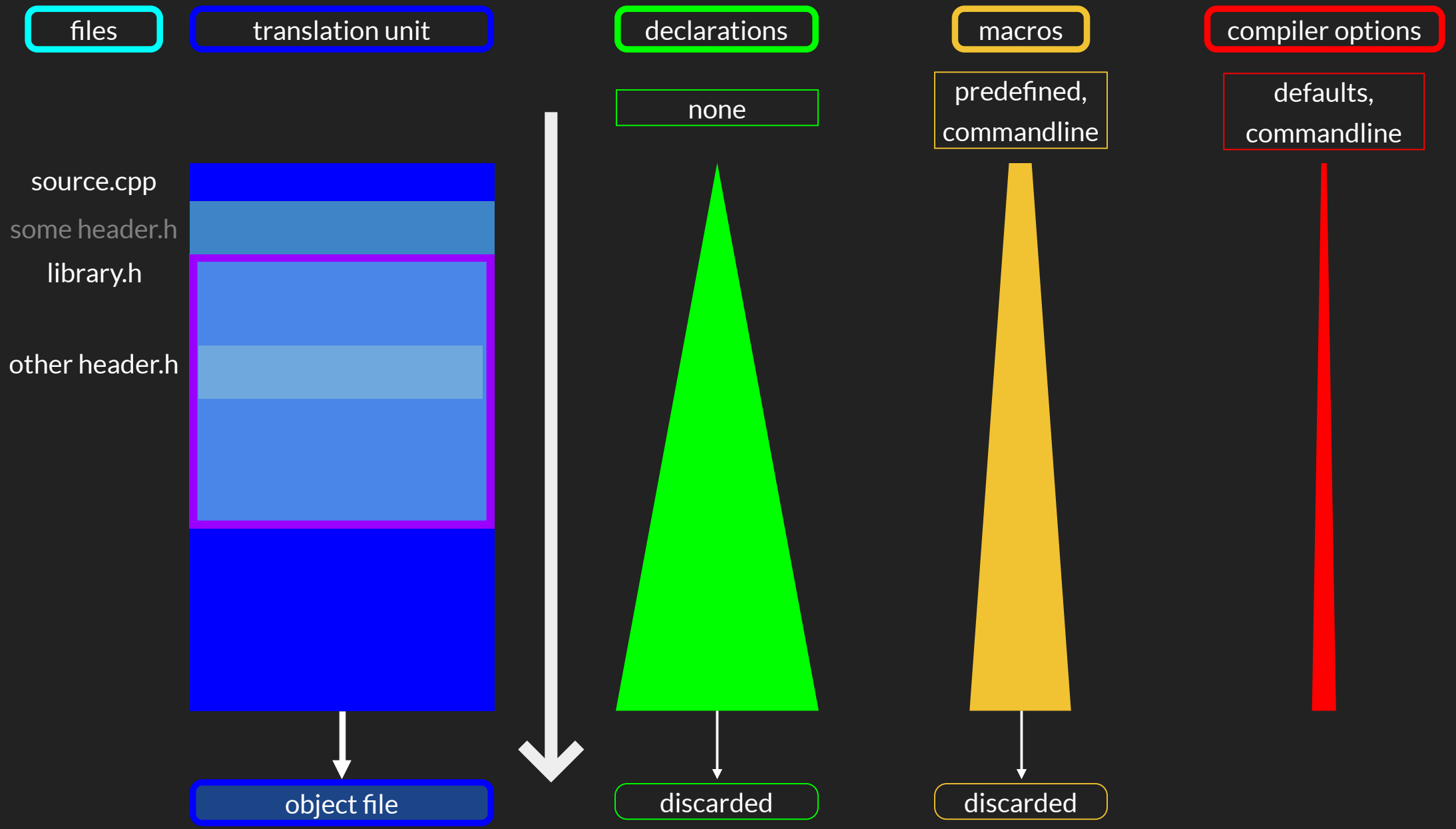
- employed by GMH Prüftechnik GmbH · ND-Testing – Systems – Services

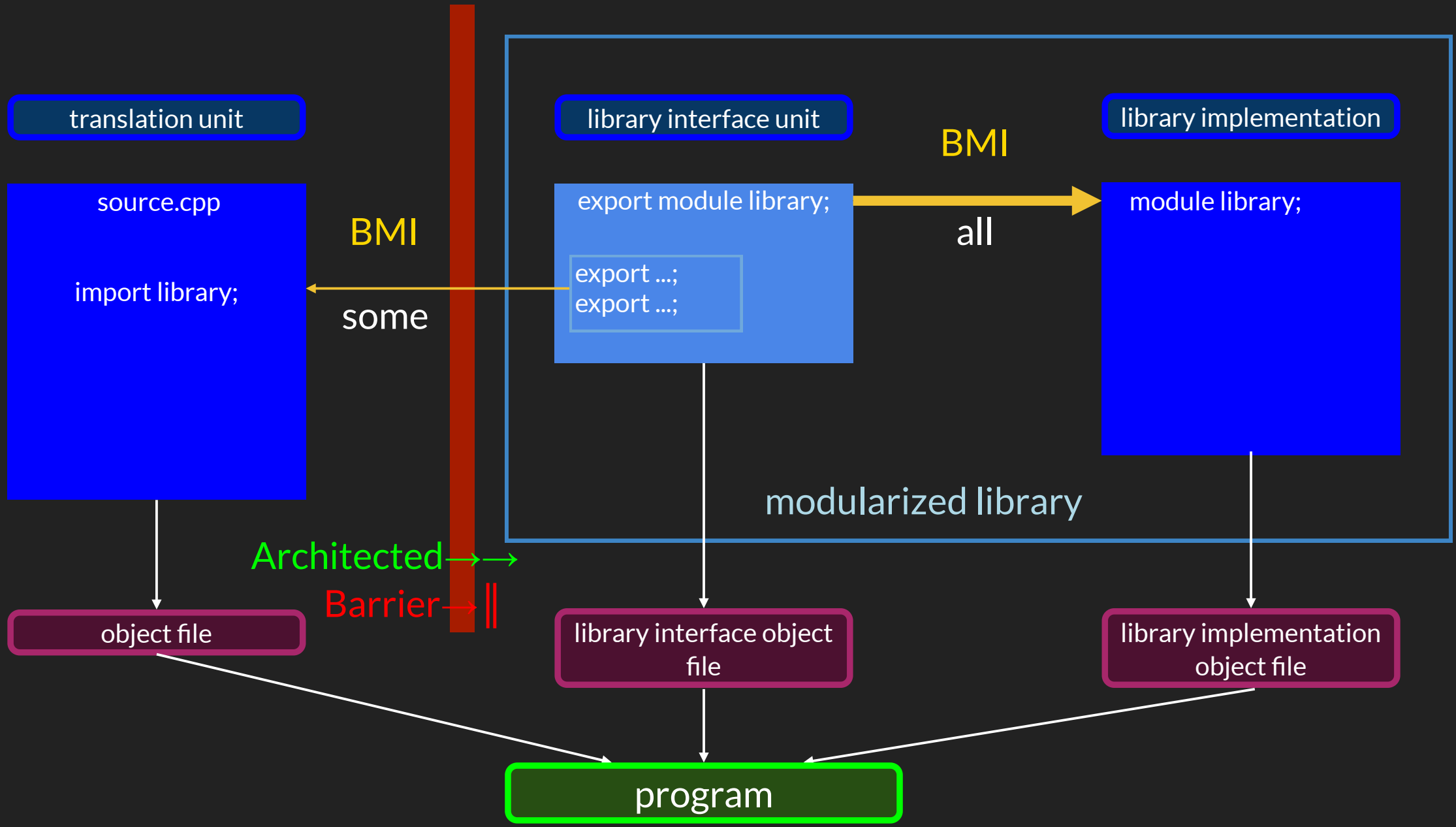
A recap of Modules

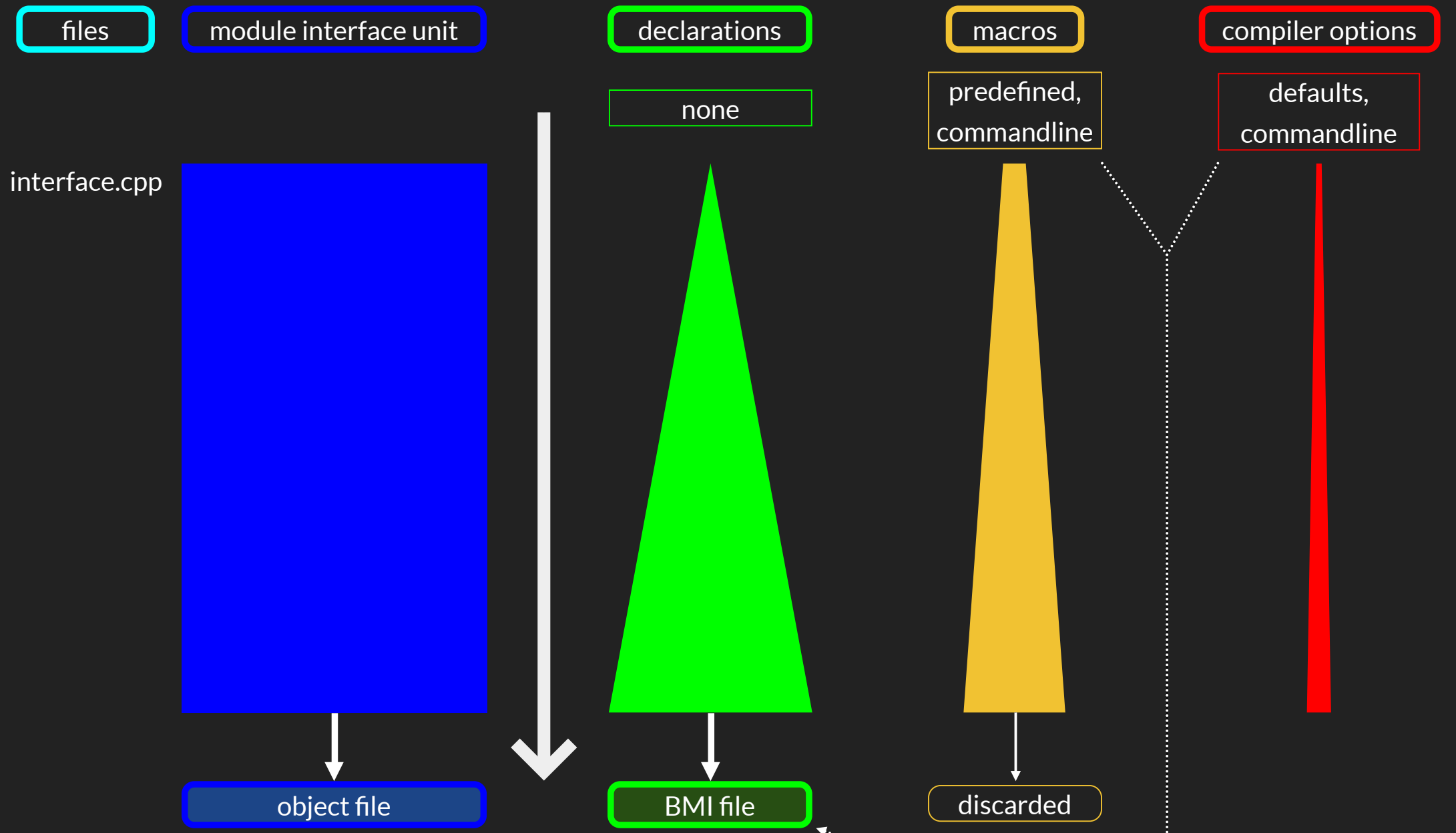
or

The shortest introduction to modules ever


library consumer
translation unit
source.cpp
some header.h
library.h
other header.h
library interface
library.h
other header.h
library implementation
library.cpp
library.h
other header.h
traditional library
"John Lakos Module"
Barrier
object file
library object file
program


files
translation unit
declarations
macros
compiler options
none
predefined, commandline
defaults, commandline
source.cpp
some header.h
library.h
other header.h
object file
discarded
discarded


translation unit
library interface unit
library implementation
BMI
source.cpp
import library;
BMI
some
export module library;
export ...;
export ...;
BMI
all
module library;
modularized library
Architected
Barrier
object file
library interface object file
library implementation object file
program


files
module interface unit
declarations
macros
compiler options
none
predefined, commandline
defaults, commandline
interface.cpp
object file
BMI file
discarded

(primary) module interface unit

global module fragment

- no declarations
- only preprocessor directives

```
module;

#include <vector>
```

standard library

platform library, etc.

exported

"exportedness" applies to names

```
export module my.first_module;

export import other.module;
#include "mod.h"
constexpr int beast = 666;

export std::vector<int> frob(S);
```

```
// mod.h
#pragma once;

struct S {
    int value = 1;
}
```

The name of this module can be referred to only in

- module declaration
- import declaration

module implementation unit

module declaration without a name

```
module;

#include <vector>
```

module purview

global module fragment

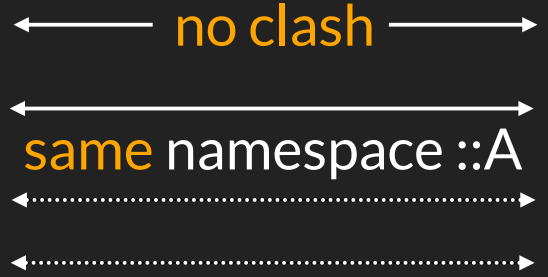
default name 'domain'

```
module my.first_module;

std::vector<int> frob(S s) {
    return { s.value + beast };
}
```

- not a namespace
- separate name 'domain'

NAME ISOLATION

```
export module mod1;

int foo(); // module linkage

export namespace A { // external l.

int bar() { // external linkage
  return foo();
}

} // namespace A
```




```
export module mod2;

int foo(); // module linkage

namespace A { // external linkage

export int baz() { // external link.
  return foo();
}

} // namespace A
```

name '::foo' is attached to module 'mod1', i.e. '::foo@mod1',

exported name '::A::bar' is also attached to the module

```
import mod1;
import mod2;

using namespace A;

int main() {
  return bar() + baz();
}
```

name '::foo' is attached to module 'mod2', i.e. '::foo@mod2',

exported name '::A::baz' is also attached to the module

namespace name '::A' is attached to the global module, as it is oblivious of module boundaries

MODULE TU TYPES & FEATURES

	Defines interface	contributes to interface	implicitly imports interface	part of module purview	part of global module	exports MACROs	creates BMI	contributes to PMIF	fully isolated
Primary Module Interface	✅	✅		✅	🟡	❌	✅	✅	✅
Mod. Implementation unit	❌	❌	✅	✅	🟡	❌	❌	❌	✅
Interface partition	❌	✅	❌	✅	🟡	❌	✅	✅	✅ !
Internal partition	❌	❌	❌	✅	🟡	❌	✅	🔶	✅ ⚠️
Private module fragment		❌		✅		❌		❌	✅
Header unit	✅	✅		❌	✅	✅	✅	❌	❌ !!‍🕳️

✅ unconditionally 🟡 if a GMF exists in the TU 🔶 if TU's BMI is (transitively) imported into the PMIF

Compilers


REGE AM:- HIC PORTATVR:CORPVS:EADWARDI:REGIS:AD:ECCLESIAM:SCI
PETRI APLI
HIC EADWARDVS:REX
INTECTO:ALLOQVIT:FIDE LES:-
HIC DEDERVNT:HAROLDO:
CORO NA: REGIS
HIC RE
REX:AN
SIDET:HAROLD
GLORVM:
STIGANT
ARChi EPS
ET HIC: DEFVNCTVS
EST

COMPILER SUPPORT AND SPECIFICS

- supported features
- completeness
- file extensions for modular translation units
  - preferred
  - others allowed
- compiler flags for compiling
  - named modules
  - header units

as advertised and / or documented ⚠️

LANGUAGE / LIBRARY FEATURES

	gcc / libstdc++	clang / libc++	msvc / ms stl
Syntax specification	C++20 -fmodules-ts 🙄	≥ 15.0: C++20 (≤ 8.0: Modules TS)	≥ 19.23: C++20 (≤ 19.22: Modules TS)
Named modules	✅ partial	✅ partial	✅
Module partitions	✅ partial	✅ partial	✅
Header units	✅	✅	✅
Private mod. fragment	⛔	✅	✅
Name attachment	✅ strong model	✅ strong model	✅ strong model
#include → import	✅	✅	✅
__cpp_modules	🟨 201810L 🤨	⛔	✅ 201907L
Modularized std library	⛔	🟨 partial, experimental	✅
Documentation	🟩 terse	✅	✅

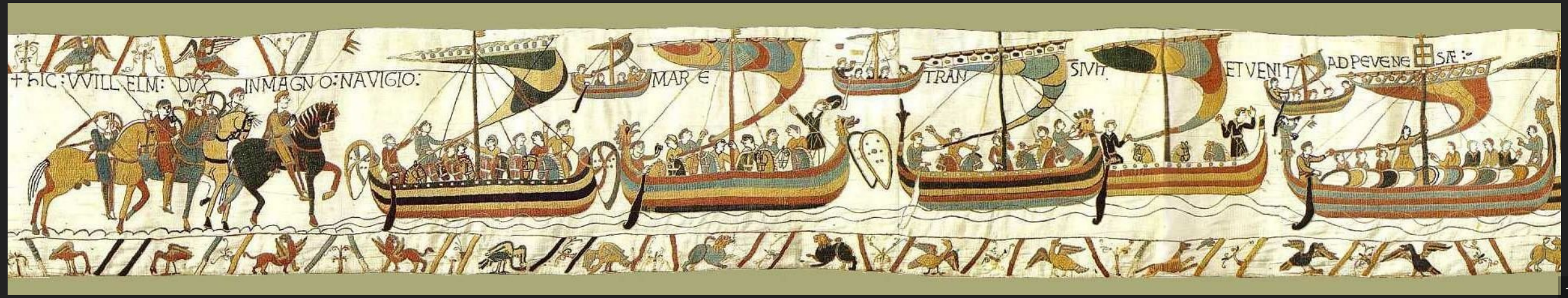
BMI CONSISTENCY - COMPILER FLAGS

- msvc:
  - pretty lenient
  - fastmath, execution encoding, RTTI, exception handling
  - warnings, can be suppressed -> ODR violations possible
- clang:
  - very strict, an unspecified set of flags matters, language flags matter, release/debug doesn't matter -> ODR violations possible
  - not overridable, warnings can be suppressed 🤦
- gcc:
  - no documentation

FILE NAMING - EXTENSIONS

- clang:
  - all extensions possible, if given the `-x c++-module` option
  - preferred: .cppm (ccm, cxxm, c++m), infers `-x c++-module`
  - header units: `-x c++-header`, `-x c++-user-header`, or `-x c++-system-header`
- gcc:
  - all extensions possible, if given the `-x c++` option
  - preferred: .cpp (cc, cp, cxx, c++), infers `-x c++`
  - header units: `-x c++-header`, `-x c++-user-header`, or `-x c++-system-header`
- msvc:
  - all extensions possible, given proper `/interface` or `/internalPartition`, and `/TP`
  - preferred: .ixx for all sources that contribute to the interface of a module, infers `/interface`
  - header units: `/exportHeader` with optional header type qualification

Modularizing traditional libraries



ARGPARSE

argparse.hpp, C++17, single-file, header-only, portable

```
#include <algorithm>
#include <any>
... more standard library headers

namespace argparse {

namespace details {
... const variables, classes, and functions, some templated
}

... enums, operator&()

class ArgumentParser;

class Argument {
... many variables, classes, and functions, many templated
};

class ArgumentParser {
... many variables, classes, and functions, many templated
};

} // namespace argparse
```

ARGPARSE

argparse.ixx, naïve modularization using wholesale export

```
export module argparse; // module interface declaration [module.unit]/2

#include <algorithm>
#include <any> // ... more standard library headers

namespace argparse {

namespace details { // not exported -> invisible, but reachable!
...
}

export { // export declaration sequence [module.interface]/1
... enums, operator&()

class ArgumentParser;

class Argument {
};

class ArgumentParser {
};

} // export
} // namespace argparse
```

COMPILE IT

Clang 17.0.4

```
// -std=c++2b      : C++23
// -fmodule-output : compile module interface (or partition) -> generate BMI module-name.pcm
// -c              : create object file, too
// -x c++-module   : input is a C++ module

clang++ -std=c++2b -fmodule-output -c -x c++-module argparse.ixx


->  doesn't compile


In file included from argparse.ixx:32:
In file included from /include/c++/13.2.0/algorithm:60:
In file included from /include/c++/13.2.0/bits/stl_algobase.h:65:
In file included from /include/c++/13.2.0/bits/stl_iterator_base_types.h:71:
/include/c++/13.2.0/bits/iterator_concepts.h:72:12: error: declaration of 'iterator_traits' in module
      argparse follows declaration in the global module
    struct iterator_traits;
           ^
/include/c++/13.2.0/bits/cpp_type_traits.h:470:29: note: previous declaration is here
  template<typename> struct iterator_traits;

more errors to follow
```

COMPILE IT

gcc 13.2.0

```
// -std=c++2b             : C++23
// -fmodules-ts           : enable C++20 modules
// (deduced from content) : module interface -> generate BMI
// -x c++                 : input is C++

g++ -std=c++2b -fmodules-ts -c -x c++ argparse.ixx


  -> doesn't compile


In file included from /include/c++/13.2.0/bits/stl_algobase.h:59,
                 from /include/c++/13.2.0/algorithm:60,
                 from argparse.ixx:32:
/include/c++/13.2.0/bits/c++config.h: In function 'void std::__terminate()':
/include/c++/13.2.0/bits/c++config.h:321:10: error: block-scope extern declaration 'void std::terminate()' not permit
  321 |     void terminate() _GLIBCXX_USE_NOEXCEPT __attribute__ ((__noreturn__));
      |          ^~~~~~~~~
In file included from /include/c++/13.2.0/bits/stl_iterator_base_types.h:71,
                 from /include/c++/13.2.0/bits/stl_algobase.h:65:
/include/c++/13.2.0/bits/iterator_concepts.h: At global scope:
/include/c++/13.2.0/bits/iterator_concepts.h:72:12: error: cannot declare 'struct std::iterator_traits< <template-par
   72 |     struct iterator_traits;
      |            ^~~~~~~~~~~~~~
In file included from /include/c++/13.2.0/bits/stl_algobase.h:61:
/include/c++/13.2.0/bits/cpp_type_traits.h:470:29: note: declared here
```

COMPILE IT

MSVC 14.38

```
// /std:c++latest : C++23
// /interface     : compile a module interface (or partition) -> generate BMI
// /TP            : input is C++

cl /c /EHsc /std:c++latest /interface /TP argparse.ixx

  ->  compiles and creates BMI 'argparse.ifc' + object 'file argparse.obj'


argparse.ixx(32): warning C5244: '#include <algorithm>' in the purview of module 'argparse' appears erroneous.
  Consider moving that directive before the module declaration,
  or replace the textual inclusion with 'import <algorithm>;'.

more warnings to follow
```

FIX IT

introduce the GMF and place the standard library headers there

```
// at most comments or white space lines here !
module; // introduce the so-called global module fragment (GMF) [module.global.frag]

// put all your #includes here that come from the traditional,
// non-modular code that lives in the global module

// *all* of them, pretty please! 😺

#include <algorithm>
#include <any> //
... all the other library headers

export module argparse; // module interface declaration [module.unit]/2

// no more *traditional* #includes

namespace argparse {
...
```

COMPILE IT

Clang 17.0.4

```
// -fmodule-output=<name>=<BMI path> : generate BMI at given location

clang++ -std=c++2b -fmodule-output=$(BMIpath)/argparse.pcm -c -x c++-module argparse.ixx

  ->  compiles and creates BMI 'argparse.pcm' + object file 'argparse.o'  😊🚀
```

gcc 13.2.0

```
g++ -c -std=c++2b -fmodules-ts -x c++ argparse.ixx

  -> compiler crashes  😟🤯

argparse.ixx:57:8: internal compiler error: Segmentation fault
   57 | export module argparse;
      |        ^~~~~~
```

MSVC 14.38

```
cl /c /EHsc /std:c++latest /interface /TP argparse.ixx

  ->  compiles and creates BMI 'argparse.ifc' + object file 'argparse.obj'  😊✨
```

TEST IT

```
import argparse;

int main(int argc, char *argv[]) {
  argparse::ArgumentParser program("test");
  program.add_argument("--foo").implicit_value(true).default_value(false);

  auto unknown_args = program.parse_known_args(argc, argv);

  if (program.is_used("--foo"))
    return program.get<bool>("--foo");
}
```

- Clang
- MSVC

GCC is no longer in the game

TEST IT

```cpp
#include <string> // required by msvc  😢

import argparse;

int main(int argc, char *argv[]) {
  argparse::ArgumentParser program("test");
  program.add_argument("--foo").implicit_value(true).default_value(false);

  auto unknown_args = program.parse_known_args(argc, argv);

  if (program.is_used("--foo"))
    return program.get<bool>("--foo");
}
```

Both compilers succeed and create an executable 🎉

But also 😮😢

THERE'S MORE ...

The test code was carefully chosen to sidestep issues still lurking in the non-exported namespace 'details':

- unqualified name lookup that
  - (unwittlingly) invokes ADL and at least reduces compilation throughput
  - can lead to lookup failures in 2nd-phase name lookup of templates that are instantiated outside the module
- declarations with internal linkage that may cause a so-called exposure of TU-local entities [basic.link]/14

The latter is particularily damning [basic.link]/17:

> *If a (possibly instantiated) declaration of, or a deduction guide for, a non-TU-local entity in a module interface unit (outside the private-module-fragment, if any) or module partition is an exposure, the program is ill-formed. Such a declaration in any other context is deprecated.*

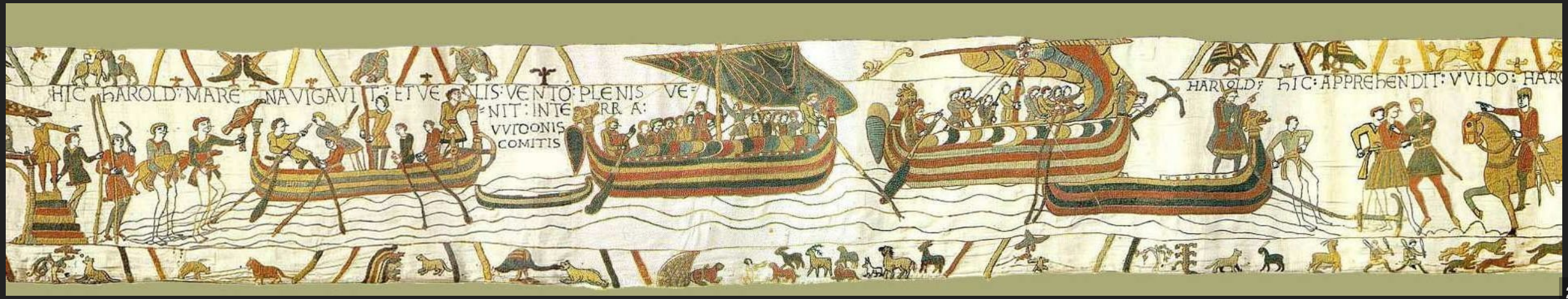
ASIO

≈6 MB source text in 673 files, ported to many compilers and platforms

-> single-TU module with the module "trinity", i.e. GMF, purview, PMF

```
module;
#include "asio-gmf.h" // all the platform- and compiler-specific includes

export module asio;

#ifdef ASIO_ATTACH_TO_GLOBAL_MODULE
extern "C++" { // [module.unit]/7.2.2, detach all exported entities from
#endif         // module 'asio' and attach them to the global module

export {     // export all of the Asio headers wholesale,
             // without exception, they all must have *external* linkage!
#include "asio.hpp"

#include "asio/ts/buffer.hpp"
... more Networking TS headers

#include "asio/experimental/awaitable_operators.hpp"
... more "experimental" headers

#if defined(ASIO_USE_SSL)
#  include <asio/ssl.hpp>
#endif
} // export
```

Build systems



BUILD SYSTEMS

There are a few build systems with some support for C++ modules

In some order:

- build2 (by Boris Kolpackov, build2.org)
  - claims to support all TU types
  - currently supports only GCC,
    formerly also Clang and MSVC
  - module dependency scanner ?
- CMake (by Kitware, CMake.org)
  - supports all named module TU types
  - no support for header units
  - supports Clang 16+, MSVC 19.32+, and GCC 14+ ⚠️
  - does module dependency scanning (required)
  - manual dependency specifications are tedious and hard

BUILD SYSTEMS

- MSBuild (by Microsoft, since toolset 16.28, Visual Studio)
  - MSVC only (for modules)
  - supports all TU types
  - supports and prefers automatic module dependency scanning
  - manual module depency specifications are possible
  - extensive documentation
- xmake (community driven, xmake.io)
  "A cross-platform build utility based on Lua"
  - claims to support all C++ module TU types
  - supports only GCC?
  - no documented module dependency scanning facility
  - documentation is lacking

CMAKE

CMake 3.25 (3.26+ recommended) introduced *experimental* support (link) using modules with

- Clang (16.0 or newer)
- MSVC (19.34 or newer)

In 3.28, the support is no longer experimental, and it can eventually also handle

- GCC (14.0 or newer)

These compilers can produce dependency scanning results according to the specification described in P1689 (discussed in WG21 SG15 'Tooling')

Tip: use the Ninja generator (Ninja 1.10 or newer), the Visual Studio generator is fine, too.

CMAKE FILESETS

The former, experimental API opt-ins are obsolete, and full support is invoked with

```
cmake_minimum_required(VERSION 3.28)
```

and compiling with C++20 or newer.

Module TU sources need to be indicated a such by grouping the target sources into different *file sets* like so:

```
add_executable(demo)

target_sources(demo
  PRIVATE ${non-modular-TUs} ${module-implementation-TUs} # consume BMIs
  PRIVATE
    FILE_SET moduleunits TYPE CXX_MODULES     # those create BMIs
      FILES ${module-interface-TUs} ${module-internal-partition-TUs}
    FILE_SET headerunits TYPE CXX_MODULE_HEADER_UNITS # still TODO 😢
      FILES ${header-unit-TUs}                # those create BMIs, too!
)
```

CMAKE MODULE DEPENDENCIES

With some true dedication, you technically can describe the dependency between BMI creation and BMI consumption in CMake syntax. I do that in the module test suite of the {fmt} library. But that's brittle, limited to the most simple cases, 100% manual in all aspects, and therefore not scalable.

CMake's true support for C++ modules revolves around (dynamic) dependency discovery:

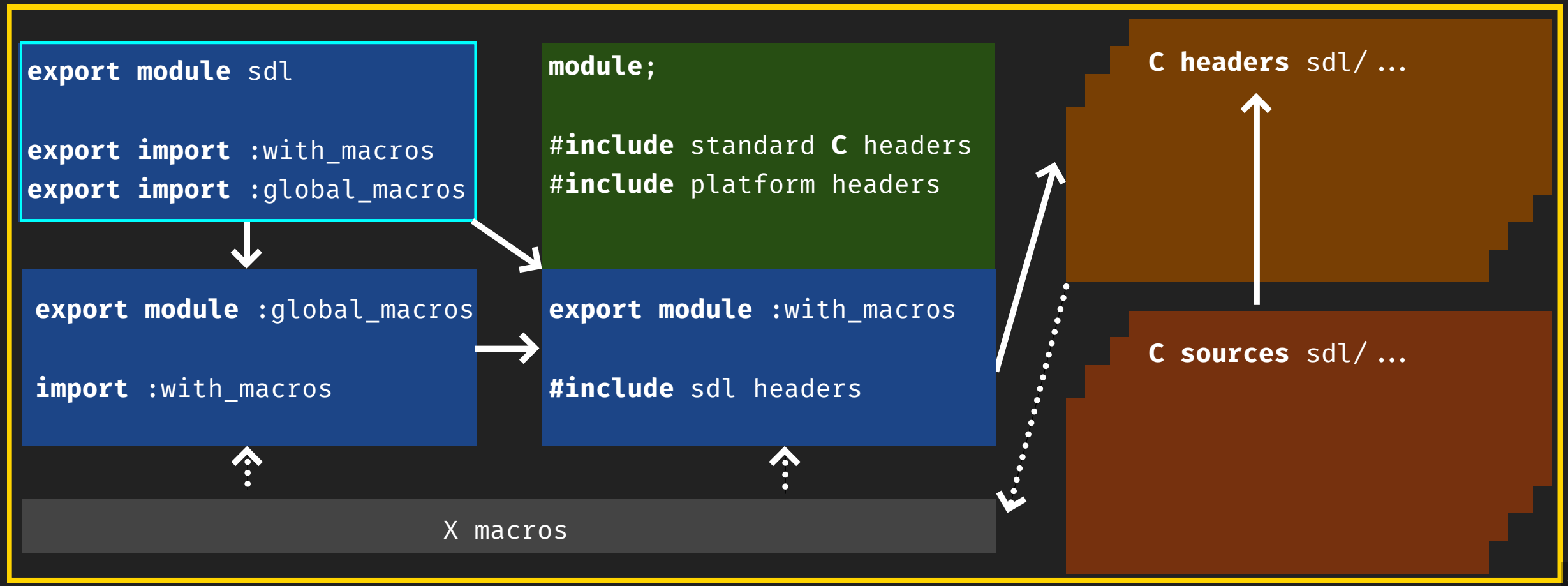
- implementation-supplied module depency scanners
- a standard-defined dependency report for each module
- module maps created and supplied to compiler invocations by CMake
- build-node processors with dynamic dependency re-evaluation (Ninja, MSBuild)

```
cmake_minimum_required(VERSION 3.28)
set(CMAKE_CXX_EXTENSIONS OFF)                    # required with Clang
```

Module dependencies

MODULE DEPENDENCY SCANNING

Let's look at module 'sdl', a static C library with a C++ module interface, composed from a primary module interface and two module interface partitions.



MODULE DEPENDENCY REPORT

The report from scanning interface partition 'sdl:global_macros' generated by Clang:


```
{
  "revision": 0,
  "rules": [
    {
      "primary-output": "SDL/CMakeFiles/sdl.dir/module/sdl2-global-macros.ixx.obj",
      "provides": [
        {
          "logical-name": "sdl:global_macros",
          "is-interface": true,
          "source-path": "SDL/module/sdl2-global-macros.ixx"
        }
      ],
      "requires": [
        {
          "logical-name": "sdl:with_macros"
        }
      ]
    }
  ],
  "version": 1
}
```

MODULE MAPS

CMake generates module maps from the reports.

Clang:

```
-x c++-module
-fmodule-output=SDL/CMakeFiles/sdl.dir/sdl-global_macros.pcm
-fmodule-file=sdl:with_macros=SDL/CMakeFiles/sdl.dir/sdl-with_macros.pcm
```

MSVC:

```
-interface
-ifcOutput SDL/CMakeFiles/sdl.dir/sdl-global_macros.ifc
-reference sdl:with_macros=SDL/CMakeFiles/sdl.dir/sdl-with_macros.ifc
```

These are fed to the compilers to compile the module unit.

The demo code

MAIN

```
/* ================================================================
The server

 - waits for clients to connect at anyone of a list of given endpoints
 - when a client connects, observes a given directory for all files in there,
   repeating this endlessly
 - filters all GIF files which contain a video
 - decodes each video file into individual video frames
 - sends each frame at the correct time to the client
 - sends filler frames if there happen to be no GIF files to process

The client

 - tries to connect to anyone of a list of given server endpoints
 - receives video frames from the network connection
 - presents the video frames in a reasonable manner in a GUI window

The application

 - watches all inputs that the user can interact with for the desire to end
   the application
 - handles timeouts and errors properly and performs a clean shutdown if needed
```

MODULE STRUCTURE

Besides the main translation unit 'main.cpp', there are

compiled in project:

- 8 named modules:
  - executor
  - gui
  - net
  - the.whole.caboodle
  - video
  - client
  - server
  - events
- 1 header (because reasons 🙄)
  - c_resource.hpp

out-of-project dependencies:

- 4 named modules from external libraries:
  - asio
  - argparse
  - libav
  - sdl
- the modularized C++23 standard library
  - compiled from the platform C++ library
  - + polyfill.hpp
    - std::generator (P2502 reference impl.)
    - std::print (partial)
    - std::start_lifetime_as (partial)

DEMO CODE TIME

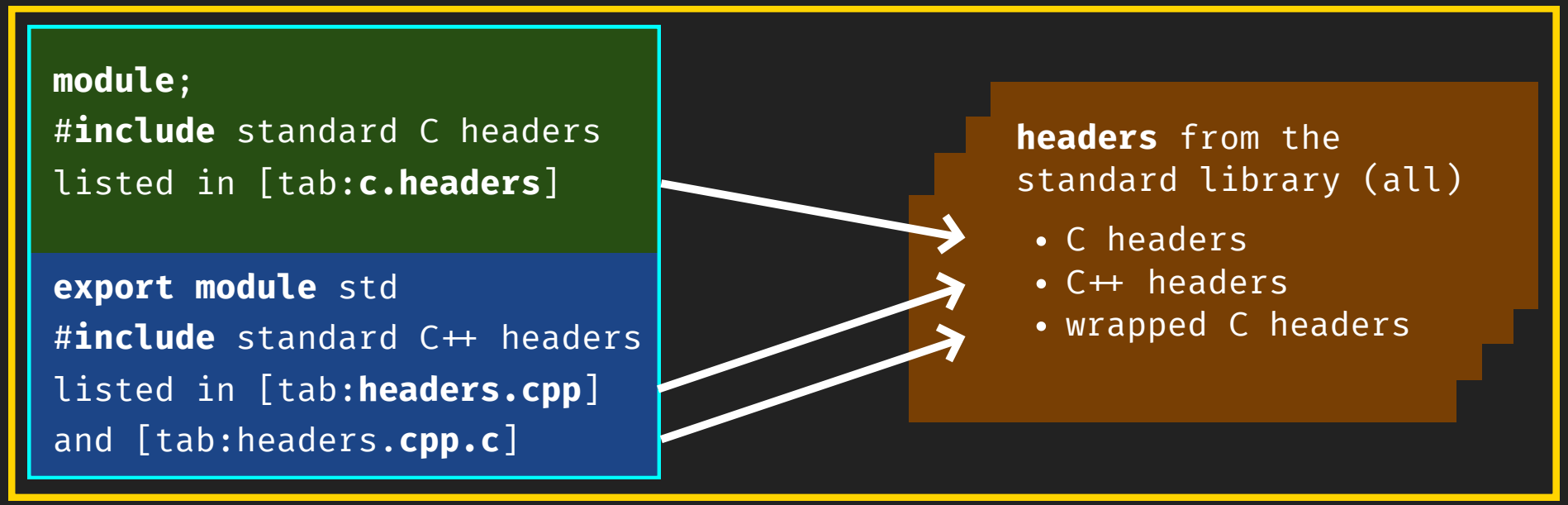
Conclusion


ET:VENERVNT
AD PRELIVM: CON TRA: HAROL DVM:REGE HIC : V VILLELM:DVX INTERROGAT:VITAL: SIVI DISSET
HAROLDI

MODULE STD

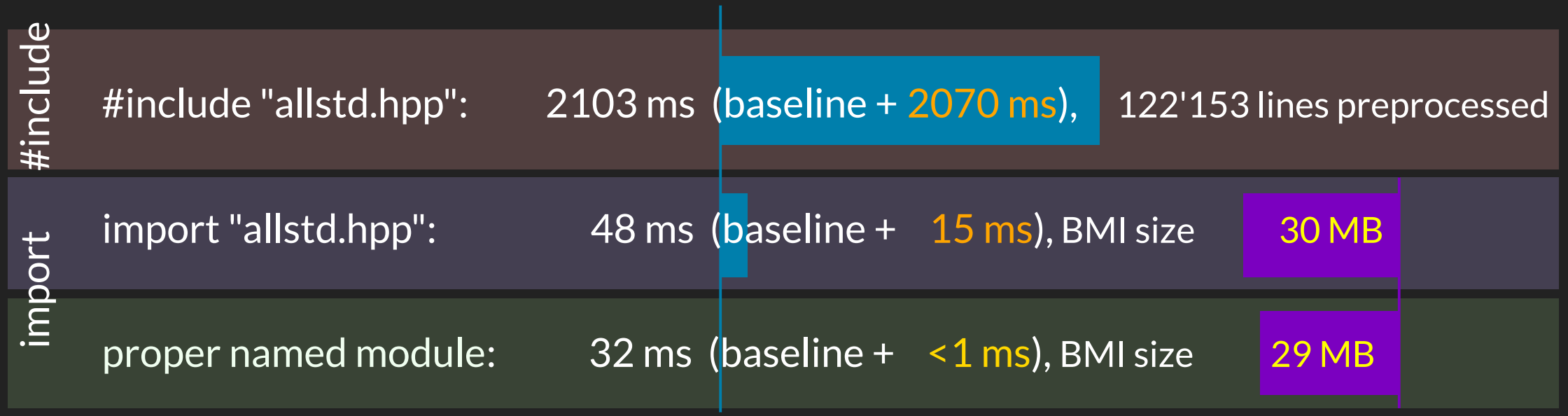
A single-file module compiled as a static lib that contains

- a global module fragment with all headers that must not be attached to the module
- the module purview with all exported interfaces



STANDARD LIBRARY USAGE SCENARIOS

```
#include "allstd.hpp"   // make the complete API of
import "allstd.hpp";    // the C++ standard library
import std;             // visible

int main {}             // but don't use it
```



{FMT} USAGE SCENARIOS

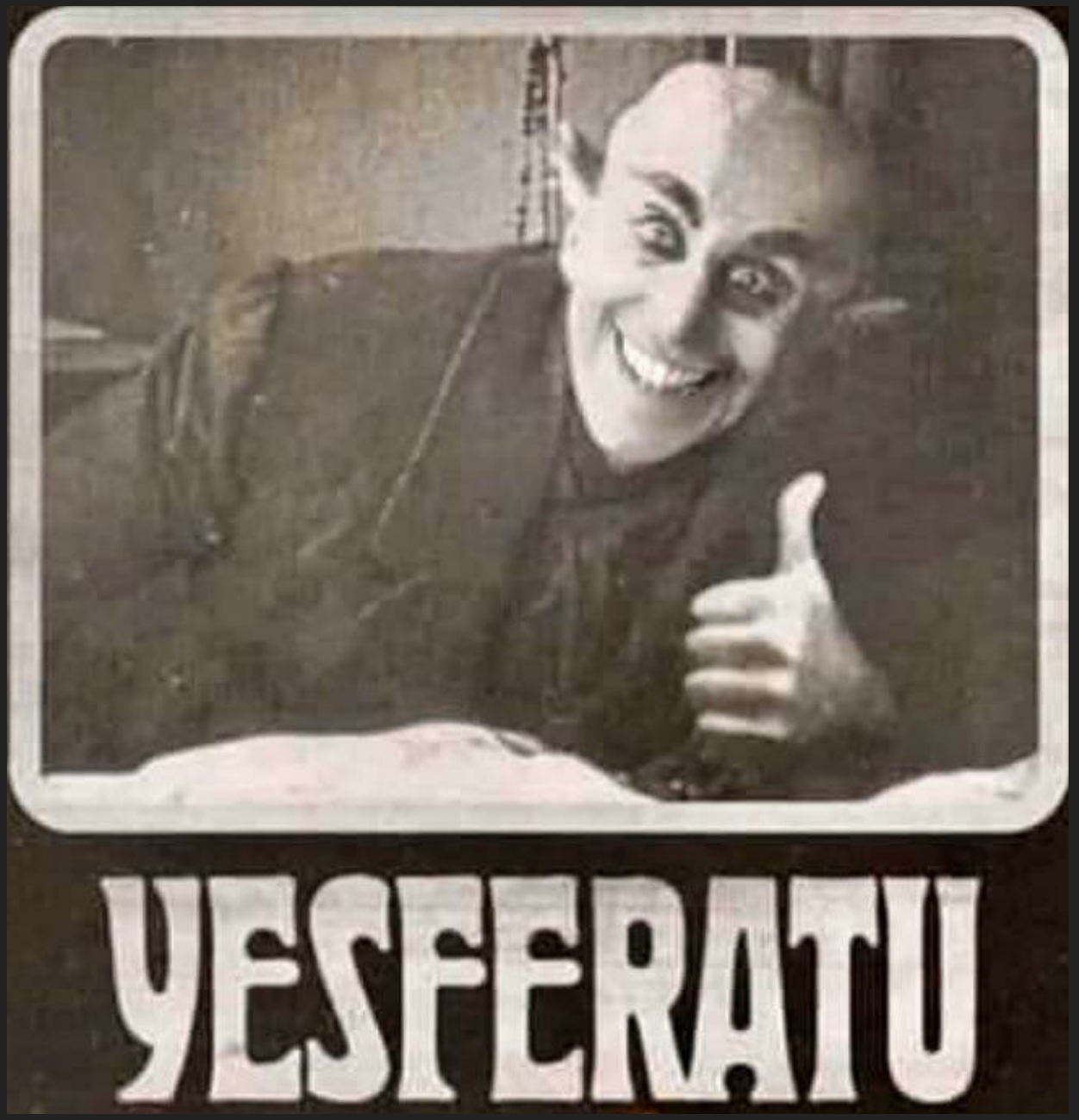
THE FINAL COMPARISON RESULT

```
#include <fmt/*.h>   // provide the *full* API
import <fmt/*.h>     // provide the *full* API
import fmt;          // provide the *full* API
```

#include	#include (header-only):	1599 ms	(baseline + 1568 ms), 90'431 lines preprocessed
	#include (static library):	1422 ms	(baseline + 1391 ms), 88'576 lines preprocessed
	Mod. STL (header-only):	658 ms	(baseline + 627 ms), 10'249 code lines
	Mod. STL (static library):	430 ms	(baseline + 399 ms), 8'038 code lines
import	import (header-only):	160 ms	(baseline + 129 ms), BMI size 117 MB
	import (static library):	155 ms	(baseline + 124 ms), BMI size 91 MB
	named module:	31 ms	(baseline + <1 ms), BMI size 8 MB

This is the way!

128'431 lines preprocessed


YESFERATU

RESOURCES

- C++26 draft — eel.is/c++draft, the latest available draft standard text
- argparse module — github.com/DanielaE/argparse/tree/module
- Asio module — github.com/DanielaE/asio/tree/module
- Demo code — github.com/DanielaE/CppInAction

Contact

dani@ngrt.de

@DanielaKEngert@hachyderm.io

DanielaE

@DanielaKEngert (mostly ignored)

Images: Bayeux Tapestry, 11th century, world heritage

Highlighting magic: Hana Dusíková

source: WikiMedia Commons, public domain , Unsplash & GIPHY

QUESTIONS?

Ceterum censeo ABI esse frangendam