A (Short) Tour of C++ Modules

DANIELA ENGERT

CppCon
The C++ Conference
2021
October 24-29

A (SHORT) TOUR OF C++ MODULES

Modules demystified and applied

Daniela Engert - CppCon 2021

ABOUT ME

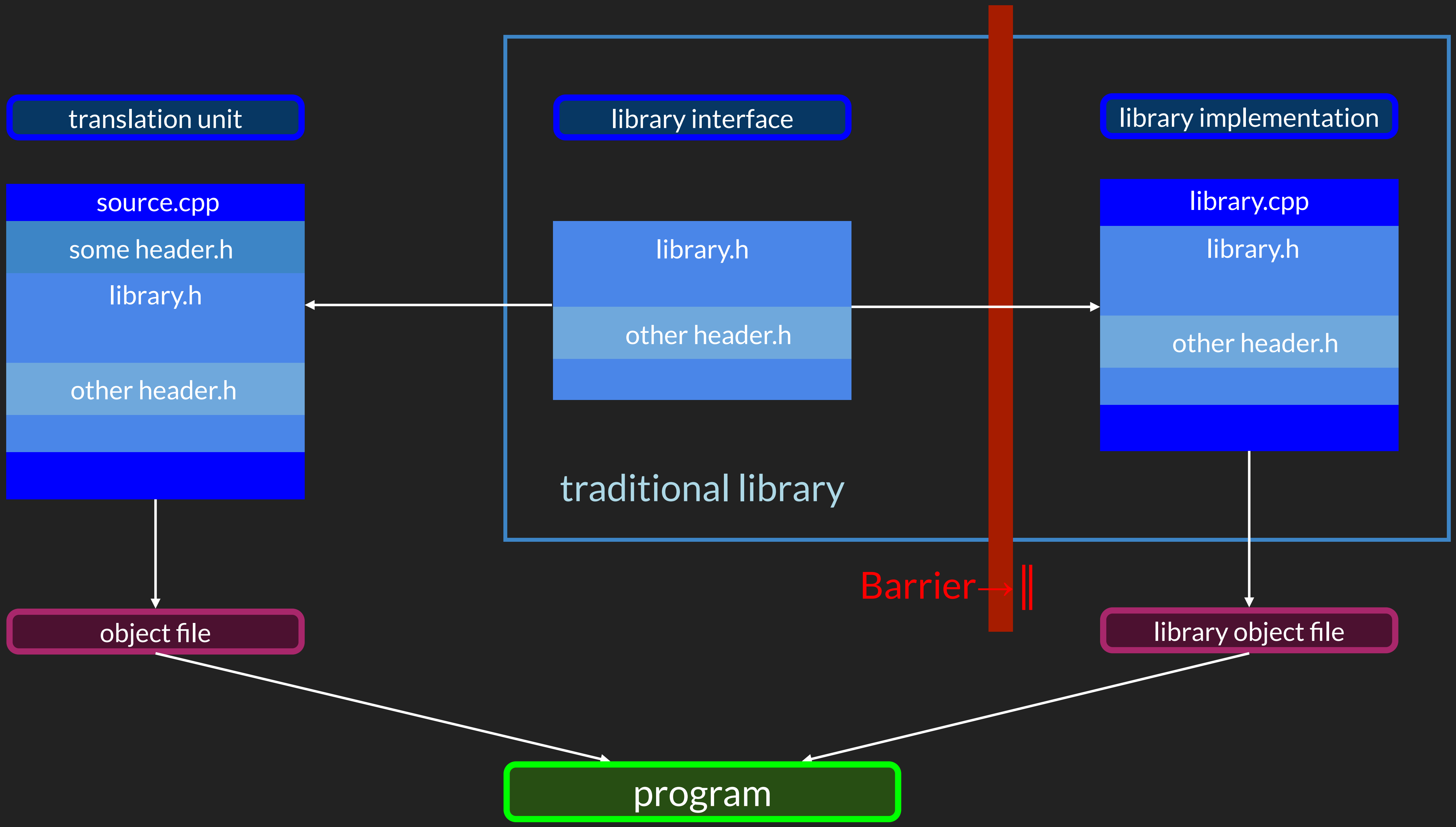
- Electrical engineer
- Build computers and create software for 40 years
- Develop hardware and software in the field of applied digital signal processing for 30 years
- Member of the C++ committee (learning novice)
- employed by GMH Prüftechnik GmbH · ND-Testing - Systems - Services

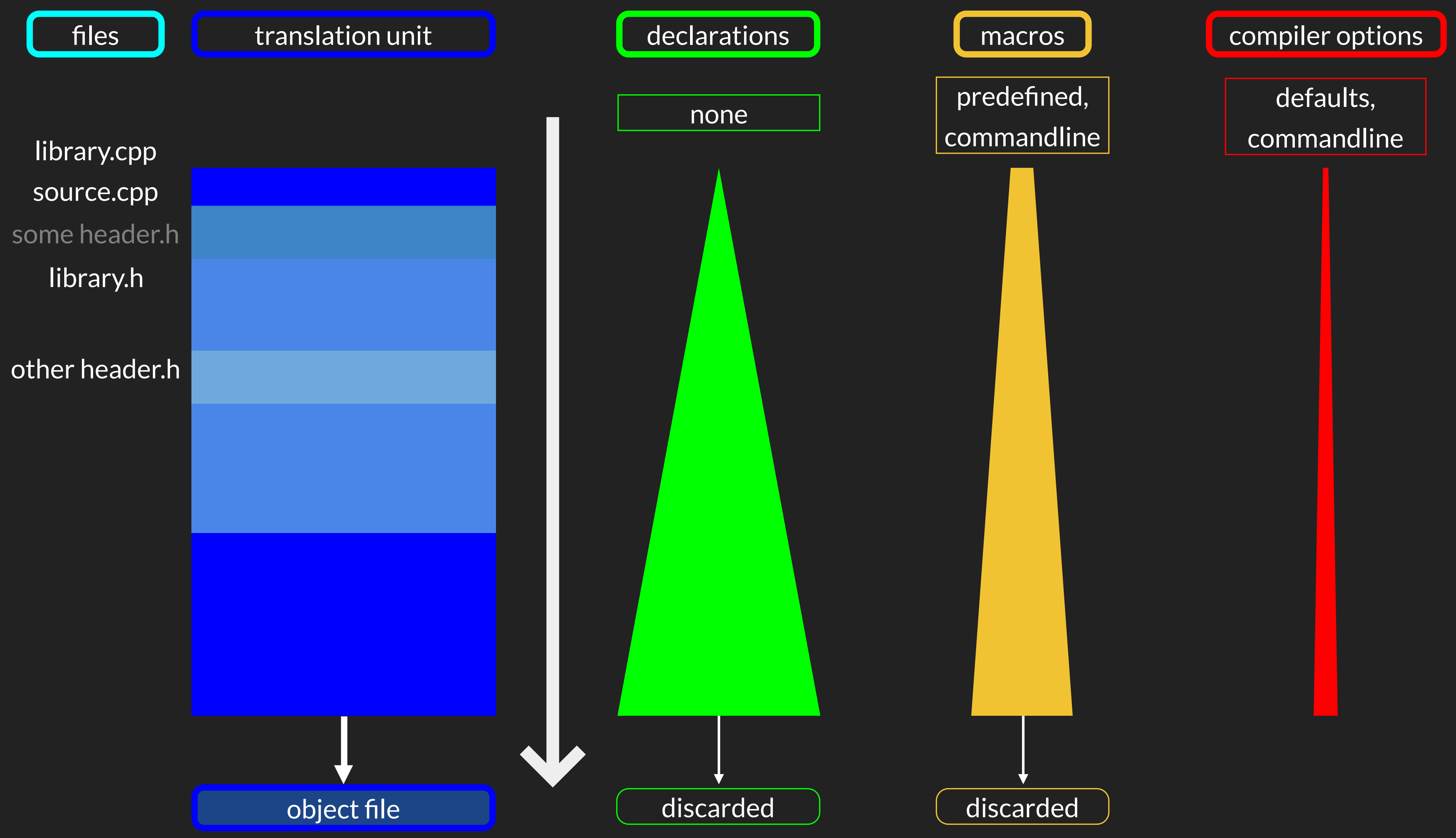
OVERVIEW

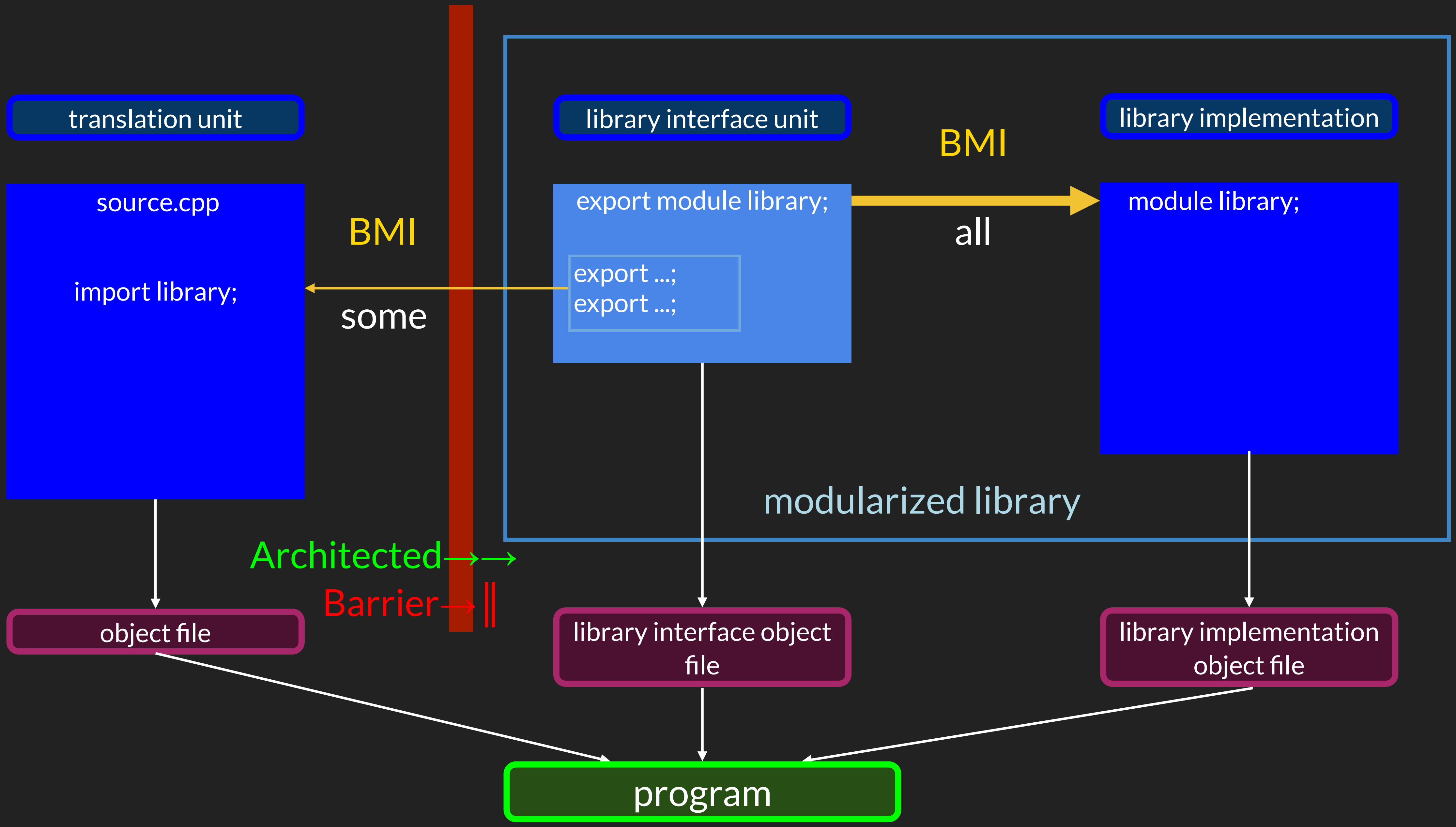
- Modules Foundations
  - C++20 Modules, a short recap
  - Module unit types and Module composition
  - Visibility of Identifiers vs. Reachability of Declarations
  - Relationships, linkage and linker symbols
- Modules in practice
  - Moving towards modules (by example)
  - Imports are different!
  - Is it worth it? (a case study)
  - The state of the ecosystem

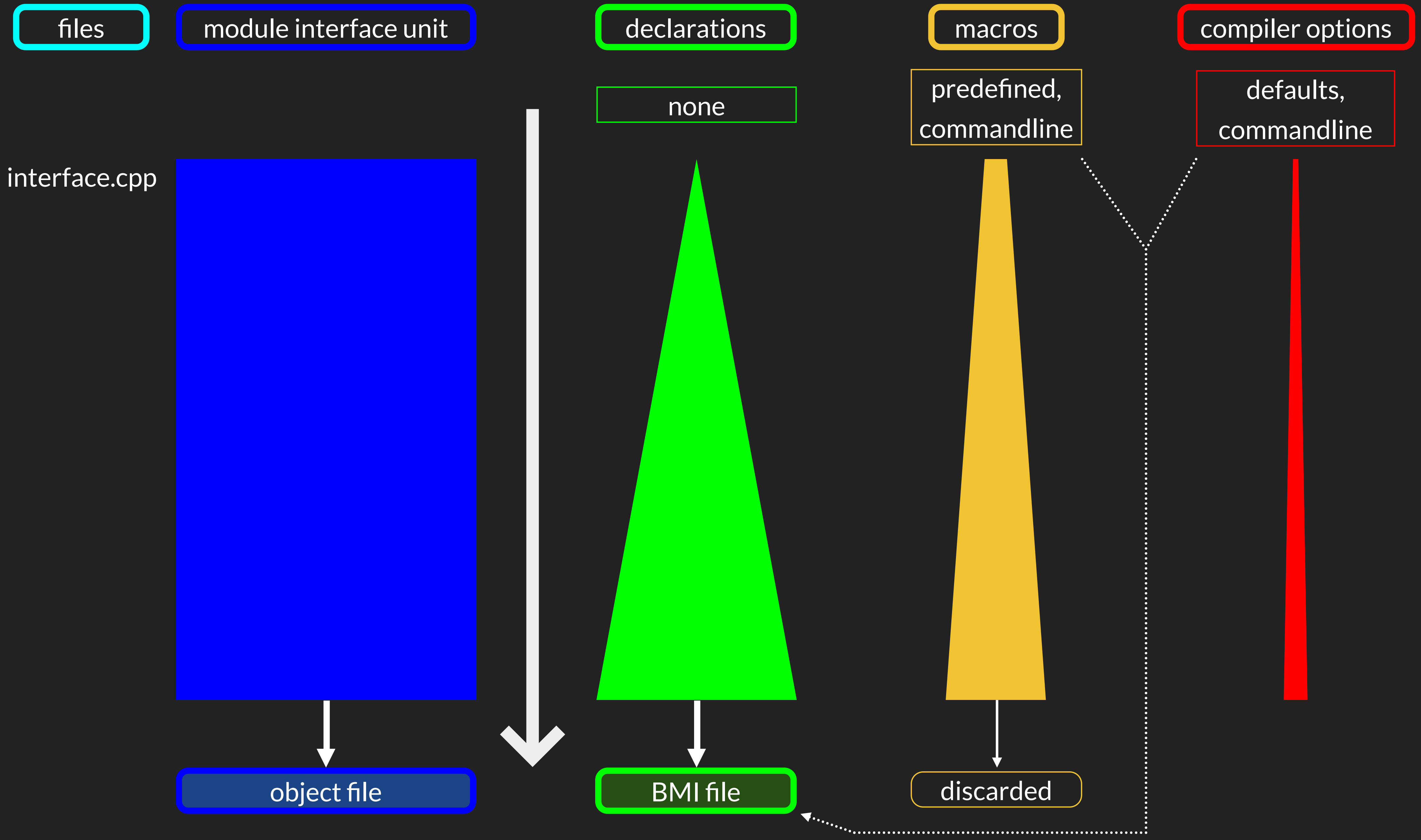
C++20 MODULES

a short recap


translation unit
source.cpp
some header.h
library.h
other header.h
object file
library interface
library.h
other header.h
traditional library
library implementation
library.cpp
library.h
other header.h
library object file
Barrier⟶‖
program


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


translation unit
source.cpp
import library;
BMI
some
library interface unit
export module library;
export ...;
export ...;
BMI
all
library implementation
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

```cpp
module;

#include <vector>

export module my.first_module;

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

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

module purview

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

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

exported

"exportedness" applies to names

The name of this module can be referred to only in

- module declaration
- import declaration

module implementation unit

module declaration without a name

```cpp
module;

#include <vector>

module my.first_module;

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

global module

default name 'domain'

- not a namespace
- separate name 'domain'

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 BMI	fully isolated
Interface unit	✅	✅		✅	❎		✅	✅	✅
Implementation unit			✅	✅	❎				✅
Interface partition		✅		✅	❎		✅	✅	✅
Implementation partition				✅	❎		✅	❎	✅
Private module fragment				✅					✅
Header unit	✅	✅			✅	✅	✅	✅	

✅ unconditionally ❎ if a GMF exists in the TU / if TU's BMI is imported into the primary module interface

MODULE COMPOSITION

This zoo of module TU types allow for various module structures:

- simple module: primary module interface unit + 1 ... n module implementation units

```
module; // GMF

#include <vector>

export module SimpleModule;

// non-exported declarations
struct detail {
  int answer = 42;
};

export
namespace SimpleModule {

void f(const detail &);
std::vector<detail> g();

}
```

```
module SimpleModule;

namespace SimpleModule {

void f(const detail & D) {
// do something with D
}

}
```

```
module;

#include <vector>

module SimpleModule;

namespace SimpleModule {

std::vector<detail> g() {
  return {{ 42 }, { 43 }};
}

}
```

formerly the interface header

formerly implementation sources for f() and g()

MODULE COMPOSITION

- large module: primary module interface unit + 1 ... n module partitions

```
export module LargeModule;

export import : iface.f;
export import : iface.g;
```

```
export
module LargeModule : iface.f;
import : detail;

namespace LargeModule {

export
void f(const detail & D);

}
```

```
module LargeModule : impl.f;
import : iface.f;

namespace LargeModule {

void f(const detail & D) {
// do something with D
}

}
```

```
module LargeModule : detail;

// non-exported declarations
struct detail {
  int answer = 42;
};
```

```
module;
#include <vector>

export
module LargeModule : iface.g;
import : detail;

namespace LargeModule {

export
std::vector<detail> g();

}
```

```
module;
#include <vector>

module LargeModule : impl.g;
import : iface.g;

namespace LargeModule {

std::vector<detail> g() {
  return {{ 42 }, { 43 }};
}

}
```

MODULE COMPOSITION

- large module: hierarchy of primary module interface unit + 1 ... n related modules

```cpp
export module HierModule;

export import HierModule.f;
export import HierModule.g;
```

```cpp
export
module HierModule.f;
import HierModule.detail;

namespace HierModule {

export
void f(const detail & D);

}
```

```cpp
module HierModule.f;
import HierModule.detail;

namespace HierModule {

void f(const detail & D) {
// do something with D
}

}
```

```cpp
export module HierModule.detail;

struct detail {
  int answer = 42;
};
```

```cpp
module;
#include <vector>

export
module HierModule.g;
import HierModule.detail;

namespace HierModule {

export
std::vector<detail> g();

}
```

```cpp
module;
#include <vector>

module HierModule.g;
import HierModule.detail;

namespace HierModule {

std::vector<detail> g() {
  return {{ 42 }, { 43 }};
}

}
```

MODULE COMPOSITION

- small module: only primary module interface unit

```
module;

#include <vector>

export module SmallModule;

struct detail {
  int answer = 42;
};

export
namespace SmallModule {

void f(const detail & D) {
  // do something with D;
}

std::vector<detail> g() {
  return {{ 42 }, { 43 }};
}

}
```

MODULE COMPOSITION

- single file module: only primary module interface unit with private partition

```
module;
#include <vector>

export module SingleFileModule;

struct detail {    // not exported but reachable
  int answer = 42;
};

export namespace SingleFileModule {
void f(const detail & D);
std::vector<detail> g();
}

module : private; // neither exported nor reachable!

namespace SingleFileModule {
void f(const detail & D) {
  // do something with D;
}

std::vector<detail> g() {
  return {{ 42 }, { 43 }};
}
}
```

VISIBILITY

hide and seek

STARTING SIMPLE

global scope

Lookup of entities
at global scope

relative to their
point of declaration

```
// translation unit 1

int i;     // point of declaration (POD), introduces entity 'i'

int j = i; // POD, introduces entity 'j'
           // point of look-up (POL), names visible entity 'i'

int k = l; // POD, introduces entity 'k'
           // POL, names invisible entity 'l'
           // entity 'l' is not yet declared
           // (relative invisibility)

int l;     // point of declaration (POD), introduces entity 'l'

int m = n; // POD, introduces entity 'm'
           // POL, names invisible entity 'n'
           // entity 'n' is declared in a different TU
           // (total invisibility)
```

```
// translation unit 2

int n;     // POD, introduces entity 'n'
```

LESS OBVIOUS

class scope

Lookup of entities at class scope

from outside the class

```
template <typename T>
int foo(T t) {
  return t.value_;  // POL ?, names not yet visible entity 'value'
}                   // at class scope of dependent, visible entity 't'

struct S {
  int value_ = 1;   // POD, introduces entity 'value'
};                  // at class scope of 'S'

int x = S{}.value_; // POL, names visible entity 'value' at
                    // class scope of visible entity 'S

x = foo(S{});       // POL !, names visible entity 'value' at
                    // class scope of visible entity 'S'
```

Lookup of entities at class scope

from within the class

```
struct S {
  int foo() {
    return value;   // POL ?, names not yet declared entity 'value'
  }                 // at class scope of visible entity 'S'

  int value = 1;    // POD, introduces entity 'value'
                    // at class scope of 'S'

};                  // POL !, names visible entity 'value'
                    // at class scope of visible entity 'S'
```

LOOKUP ALGORITHMS

visibility, it depends on *how* you look

Entities may become hidden (i.e. invisible to lookup) by

- names introduced in scopes nearer to the point of lookup
- hidden friends

but become visible again by selecting the appropriate lookup algorithm

```
namespace N {
int n = 1;              // POD, introduces entity 'N::n'

class S {
  int j_ = 0;
  friend int f(S s) {   // POD, introduces entity 'N::f', declared as friend from within
    return s.j_;        // class scope 'N::S', → so-called "hidden friend"
  }
};

namespace M {
void n();               // POD, introduces entity 'N::M::n'

int x = n;              // POL, names entity 'n', unqualified lookup (UL)
                        // FAIL: entity 'N::M::n' hides entity 'N::n' from UL
auto y = &f;            // POL, names entity 'f', UL
                        // FAIL: entity 'N::f' is invisible to UL
auto z = &S::f;         // POL, names entity 'f' in scope 'S', class member lookup (CML)
                        // FAIL: entity 'N::f' is not member of class S
auto z = &N::f;         // POL, names entity 'f' in scope 'N', qualified lookup (QL)
                        // FAIL: entity 'N::f' was not declared in scope 'N' but scope 'S'

int a = N::n;           // POL, names entity 'n' in scope 'N', QL
int b = f(S{});         // POL, names entity 'n' using argument of type S, ADL
} // namespace M
} // namespace N
```

EVEN WEIRDER

near total invisibility

```
auto foo(int x) {
  struct S {      // POD, introduces entity 'S' in function block scope
    int s_;
  };
  return S{x};    // POL, names entity 'S', part of the function's signature
}

auto what = foo(1);
assert(what.s_ == 1);

static_assert(std::is_same_v<decltype(what), ? ::S>); // POL, names invisible entity 'S'
```

Even though name 'S' is visible at function block scope and it is the function's return type, it is totally invisible from outside the function.

This is the best you can achive in terms of name hiding in a TU. Alas, it denies foo forward-declarability from another TU despite having external linkage.

SELECTIVE VISIBILITY

```
// module interface TU
export module M;

struct S {           // POD, introduces entity 'S', not exported
  int s_ = 1;
};

export S foo(int); // POD, introduces entity 'foo' and exports name 'foo'
                   // POL, names visible entity 'S'
```

```
// module implementation TU
module M;

S foo(int x) {       // POL, names visible entities 'S' and 'foo'
  return S{x};
}
```

```
// client TU
import M;            // introduces entity 'foo' by BMI, exported from module M

auto y = foo(1);     // POL, names entity 'foo'
                     // the name of result type of 'foo' is totally invisible!
```

- without modules, total invisibility of entities declared within a TU is impossible
- moving declarations from headers into modules makes them totally invisible from the outside
- exporting names from a module and importing them controls the extent to which names become visible in the translation unit importing the module's interface.

REACHABILITY

of declarations

AN EXAMPLE

```
export module mod;  // may become "necessarily reachable" if the interface of 'mod' is imported
import stuff;       // not exported, implementation detail, not part of module interface
                    // creates "interface dependency" to 'stuff' which is "necessarily reachable"

struct S {          // not exported, not meant to be used elsewhere outside
  S(const char * msg) : sth_{ msg } {}
  auto what() const { return sth_.message(); }
  something sth_;   // there must be 'something' exported from module 'stuff'
};

export              // exports name 'foo'
S foo(const char * msg) {
  return { msg };
}
```

```
#include <type_traits>
import mod;                                 // creates "interface dependency" to 'mod' and 'stuff'
                                            // makes 'mod' "necessarily reachable"

int main(int, char * argv[]) {
  const auto result = foo(argv[0]);  // so far, so good

  const auto huh = result.what();    // why is entity 'what' nameable? 🤔
  using mystery  = decltype(huh);    // it was never exported from 'mod'
                                     // -> it's a reachable semantic property of 'S'!

  static_assert(std::is_class_v<mystery>);                       // compiles
  static_assert(sizeof(mystery::value_type) == sizeof(char)); // compiles
  return huh.empty();                                            // compiles
}
```

SEMANTIC PROPERTIES

```cpp
// header file "some.h"

extern "C++" {                       // default
  using func = int(int)              // mandatory
                noexcept(false);     // default
  struct D                           // mandatory
  {                                  // optional or mandatory
    operator int() const { return v_; }
    int v_ = 1;
  };

  extern "C" {                       // optional
    namespace N {                    // mandatory
      func bar;                      // mandatory
    }

    [[nodiscard]]                    // optional
    extern                           // default
    inline                           // optional or mandatory
    int N::bar(int                   // mandatory
                x = D{})             // optional or mandatory
    {                                // optional or mandatory
      return x + D{2};
    }
  }
}
```

LINKAGE

about relationships between TUs

NAME ISOLATION

```
export module mod1;

int foo(); // module linkage

export namespace A { // external

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

} // namespace A
```

no clash

same 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

LINKER SYMBOLS

```cpp
export module mod3;

int foo();           // module linkage, attached to module 'mod3'

namespace A {        // external linkage, attached to global module

export int bar() { // external linkage, attached to module 'mod3'
  return foo();
}

} // namespace A
```

msvc 16.11 name mangling:

?foo@mod3@@YAHXZ::<!mod3> (🤔)

?bar@A@@YAHXZ::<!mod3>

clang13 & gcc(modules) name mangling:

_ZW4mod3E3foov

_ZN1A3barEv

The module name will be encoded into the linker symbol if an entity has module linkage, and may be encoded if it is exported and therefore has external linkage

OWNERSHIP

```
export module mod3;

namespace A {

export int bar() { // external linkage, attached to module 'mod3'
  return foo();
}

} // namespace A
```

Strong ownership model, the linker symbols of exported names contain the module name they are attached to

e.g. msvc

`?bar@A@@YAHXZ::<!mod3>`

Benefit:

Identical names can be exported from multiple modules and used in separate TUs without causing linker errors.

Weak ownership model, the linker symbols of exported names are oblivious of module attachment

e.g. clang & gcc

`_ZN1A3barEv`

Benefit:

Exported names can be moved freely between modules or from headers into modules.

BUT PLATFORMS...

```cpp
export module awesome.v1;

// other stuff, not exported,
// must go here because reasons

export namespace libawesome {
  // implemented elsewhere
  int deep_thought(...);
} // namespace libawesome
```

perfectly tested™

compatible

```cpp
export module awesome.v2;

// other stuff, not exported,
// must go here because reasons

export namespace libawesome {
  // for compatibility
  int deep_thought(...);
  // implemented elsewhere
  int much_deeper_thought(...);
} // namespace libawesome
```

used in implementations of OEM1 and OEM2, not exposed

```cpp
// OEM 1, traditional header
// implementation calls 'deep_thought'

namespace OEM1 {
  int doit();
} // namespace OEM1
```

perfectly tested™,

distributed as static libraries & header

```cpp
// OEM 2, traditional header
// impl. calls 'much_deeper_thought()'

namespace OEM2 {
  int makeit();
} // namespace OEM2
```

Compiles on platform A 😁
Links on platform A 😊
Profit! 🤑

```cpp
// Poor customer's application
#include "oem1/interface.h"
#include "oem2/interface.h"

int main(){
  return OEM1::doit() +
         OEM2::makeit();
}
```

Compiles on platform B 😁
Linker error on platform B 😱
but why? 🤔(weak ownership)

DETACHING NAMES

```cpp
export module mod4;

int foo();                 // module linkage, attached to module 'mod4'
extern "C" int bar();      // external linkage, attached to global module

extern "C++" int baz() { // external linkage, attached to global module
  return foo() + bar();
}
```

msvc 16.11 name mangling:

```
?foo@A@mod4@@YAHXZ::<!mod4>(🤔)
_bar
?baz@@YAHXZ
```

gcc(modules) name mangling:

```
_ZW4mod4E3foov
bar
_ZW4mod4E3bazv   (wat? 🤔)
```

Giving explicit language linkage specifications reattaches the names to the global module and mangles them accordingly into linker symbols

TRANSITIONING TO MODULES

The road forward

TRANSITIONING TO MODULES

Options available and advice on how to proceed into the modules world:

- If the interface of a library is (mostly) separate from the implementation
  - consider a named module by turning the interface headers into a module interface unit with optionally some interface partitions (see slide 9.1)
  - consider refactoring the interface to make this happen
  - think about macros in the interface and how to get rid of them
- If macros are indispensable
  - consider a named module like above plus a header file which imports the module and augments it with the macros
- Otherwise, consider using the existing headers as header units (discouraged)

- If a library must still be usable as a non-modular, traditional library consider a dual-mode library which can - by default - be #included as before, or alternatively be provided through a module interface unit.

DUAL-MODE LIBRARY

CASE STUDY: THE {FMT} LIBRARY

For the most part, the code is located in 12 headers defining the API

- core.h
- format.h
- compile.h
- printf.h

  ...

Plus 2 source files that can be precompiled into a static or shared library

- format.cc (+ format-inl.h)
- os.cc

THE {FMT} LIBRARY

Question: how can this traditional library become a full-fledged module of the same name, i.e. become a ***dual-mode*** library?

Requirements:

- a lot of macros are used in the implementation to support as many platforms, compilers and language standards as possible. This must still work.
- there are even two macros as API features in the interface.
- the unrestricted usability as a traditional library as before must be maintained.
- all implementation details must be hidden when the "Named Module" option is selected in the configuration.

THE {FMT} LIBRARY

Answer: this set of requirements is unattainable!

Neither a header module nor a named module has all necessary properties:

- header modules *can't hide* the implementation details
- named modules *can't export* macros

C++20 to the rescue: *screw macros* and support only the modern alternatives (i.e. UDLs) provided by the latest version of {fmt} !

THE {FMT} LIBRARY

Question: which implementation strategy? (refer to slides 9.x)

There is a lot of coupling between most headers because of

- the *vast* amount of *macros* used internally
- the *liberal* use of implementation details from *other* headers

And this applies to the compilable sources just as well.

This is not bad per se if the library is seen as a whole and therefore it is not untypical in traditional libraries. If a clean, layered module structure is the primary goal, untangling that 'mess' becomes necessary.

THE {FMT} LIBRARY

Answer: restructuring a dual-mode library is probably not worth the effort as long as the details are clearly separatable from the API!

Strategy:

- Wrap the existing headers and source files into a *single-file module*
- Mark the *exported* API with some *opt-in* syntax
- Make everything in the *source* files *strictly invisible* and unreachable

In other words:

- Apply preprocessor gymnastics to separate the API from details
- Move the contents of the source files into the private module fragment

THE {FMT} MODULE INTERFACE UNIT

```
module;                    // start of the 'global module fragment' (GMF)

// put *all* non-library headers (STL, platform headers, etc.) here
// to prevent further inclusion into the module purview!
#include <algorithm>
#include <sys/stat.h>
...
                           // end of external code attached to the 'global module'
export module fmt;         // start of the 'module purview'

#define macros to differentiate between interface and details

// put *all* library headers here to become
//   * the exported interface (API)
//   * the non-exported, but reachable implementation details
#include "format.h"
#include "chrono.h"
...
                           // end of declarations affecting other TUs
module : private;          // start of the 'private module fragment' (PMF)

// put *all* library sources here to become part of the compiled object file
// all required macros are available, yay!
#include "format.cc"
#include "os.cc"
```

THE {FMT} LIBRARY

This single module interface TU compiles into two artefacts:

- the compiled interface (a.k.a. BMI)
- the compiled implementation (a single object file)

This is basically a unity build of the whole library that provides the full API.

The object file may then optionally be wrapped into a static or shared library.

THE {FMT} LIBRARY

Benefits of a dual-mode library:

- usable as both a traditional library and a named module from identical sources
- has the same properties as a named library, i.e.
  - total isolation from other sources changing the compile environment
  - (hopefully) cleaner interface free of implementation details being visible
- enables gradual transitioning into the modules world depending on the maturity of compilers
- doesn't require changes to existing tests
- doesn't require re-architecting the inner dependencies

LESSIONS LEARNED
THE POTENTIAL PITFALLS

On the journey to making {fmt} a full-fledged named module, I've encountered a couple of stumbling blocks that needed to be adressed.

The properties of module interfaces require special care when implementing them. Stuff that never had to be taken into consideration with headers becomes really important now!

Most of them are due to the separation of visibility of names when

- compiling the interface TU (unrestricted visibility)
- compiling TUs that import the module (restricted visibility)

This applies to both named modules and header units!

THE POTENTIAL PITFALLS

Instantiations of templates in user code perform name lookup of dependent entities from outside of the module. Non-exported names are invisible now and may cause compile failures.

```cpp
export module M;

namespace detail {

template <typename T>
void baz(T) {}

template <typename T>
void bar(T t) {
  baz(t);                        // ok while compiling the module interface
}                                // fails to find 'baz' when 'bar' is implicitly instantiated

} // namespace detail

export template <typename T>
void foo(T t) {
  detail::bar(t);                // ok, fully qualified name lookup is done at module compilation time
}
```

THE POTENTIAL PITFALLS

Two potential solutions:

```
export module M;

namespace detail {
template <typename T> void baz(T) {}

template <typename T>
void bar(T t) {
  detail::baz(t);       // do fully-qualified name lookup if you *really* mean
}                       // to call 'detail::baz' only (i.e. disable ADL)
} // namespace detail
...
```

```
export module M;

namespace detail {

void baz(int) {}

template <typename T>
void bar(T t) {
  using detail::baz;    // "symbolic link" 'detail::baz' (looked up at module compilation time)
  baz(t);               // into the function body (thereby available at template instantiation time)
}                       // if you want to make the call to 'baz' a customization point (i.e. enable ADL)

} // namespace detail
...
```

THE POTENTIAL PITFALLS

beware of entities with internal linkage at namespace-scope when importing headers as header-units.
This is quite common when defining constants.

```
// header file 'some.h'

static const int the_answer = 42;  // internal linkage  ->  not exported from header unit

namespace {
  constexpr int the_beast = 666;   // internal linkage  ->  not exported from header unit
}
```

```
import "some.h"                    // import rather than #include!

int main() {
  return the_answer;               // name 'the_answer' is unknown because
}                                  // it wasn't eligible to be exported from 'some.h'
```

THE POTENTIAL PITFALLS

Solution:

- make them entities with external linkage
- or wrap them into other entities

```cpp
// header file 'some.h'

inline const int the_answer = 42; // define variable with external linkage

enum int_constants : int {         // enum definition has external linkage
  the_beast = 666;
};

struct constants {                 // struct definition has external linkage
  static constexpr int      no_answer = 0;
  static constexpr unsigned pi = 4;
};
```

```cpp
import "some.h"                    // import rather than #include!

int main() {
  return the_answer;               // name 'the_answer' is known now
}
```

THE POTENTIAL PITFALLS

Within the purview of a module, the 'inline' specifier gets its original meaning back!
Member bodies with function definitions at class scope are no longer implicitly 'inline'. You may give 'inline' hints if you mean it.

```
export module M;

struct S {
  int foo() { return bar(); }          // no longer implicitly 'inline',
                                       // the function call might be
                                       // invalid in module context!

  inline int bar() { return 42; }      // safe to inline
};
```

THE POTENTIAL PITFALLS

Beware of entities that are local to the TU.
Do not expose them e.g. by naming them in non-TU-local inline functions!

Learn more at [basic.link]#14 in the standard

```
export module M;

static void foo();                         // TU-local

static inline void bar() { foo(); }        // ok, TU-local

inline void baz() { bar(); }               // error, 'baz()' has module linkage
                                           // must not "expose" TU-local 'bar()'!
```

FROM HEADER TO MODULE

A reality check

USAGE SCENARIOS

1. Use {fmt} in traditional way by #including the required {fmt} headers
2. As before, but with a modularized standard library and #include translation for all standard library headers included by {fmt}
3. Use existing {fmt} headers as header units and import them
4. Use {fmt} as named module

Let's examine them in detail.

USAGE SCENARIOS

TEST SCENARIO

```
// #include <...>, import <...>, import fmt; go here
// The fictitious code requires at least the API from fmt/format.h and fmt/xchar.h

int main {
  /* empty main to zoom in on making the API available to TU
  fictitious call:

  fmt::print(L"The answer is {}", 42);
  */
}
```

Baseline, no #include or import:

compile time 31 ms

all configurations taken on an AMD Ryzen 9 5900X, compiled with msvc 16.11.3, release mode

USAGE SCENARIOS

#INCLUDE THE HEADERS

```
#include <fmt/format.h>
#include <fmt/xchar.h>

int main {
  /* empty main to zoom in on making the API available to TU
  fictitious call:

  fmt::print(L"The answer is {}", 42);
  */
}
```

Two configurations

header-only: compile time 944 ms (baseline + 913 ms),
6896 non-blank {fmt} code lines, 59'430 lines after preprocessing

static library: compile time 562 ms (baseline + 531 ms),
4685 non-blank {fmt} code lines, 42'735 lines after preprocessing

USAGE SCENARIOS

MODULARIZED STANDARD LIBRARY + #INCLUDE TRANSLATION

[CPP.INCLUDE]#7

```cpp
#include <fmt/format.h>
#include <fmt/xchar.h>

int main {
  /* empty main to zoom in on making the API available to TU
  fictitious call:

  fmt::print(L"The answer is {}", 42);
  */
}
```

Two configurations

header-only: compile time 511 ms (baseline + 480 ms)

static library: compile time 304 ms (baseline + 273 ms)

Total std lib BMI size 41 MB (461 MB if std lib user-compiled from original headers)

USAGE SCENARIOS

IMPORT THE HEADERS

```
import <fmt/format.h>;
import <fmt/xchar.h>;

int main {
  /* empty main to zoom in on making the API available to TU
  fictitious call:

  fmt::print(L"The answer is {}", 42);
  */
}
```

Two configurations

header-only: compile time 64 ms (baseline + 33 ms), BMI size 22 MB

static library: compile time 64 ms (baseline + 33 ms), BMI size 16 MB

USAGE SCENARIOS

MAKE THE COMPARISON FAIR!

```
#include <fmt/args.h>                          // provide the *full* API
#include 8 more {fmt} headers here                just as the named module does!
#include <fmt/xchar.h>

int main {
}
```

#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 (header-only):	160 ms	(baseline + 129 ms),	BMI size 117 MB
import (static library):	155 ms	(baseline + 124 ms),	BMI size 91 MB

USAGE SCENARIOS

IMPORT NAMED MODULE

```
import fmt;

int main {
  /* empty main to zoom in on making the API available to TU
  fictitious call:

  fmt::print(L"The answer is {}", 42);
  */
}
```

Sorry, only one configuration with everything that {fmt} can provide!

Module interface unit: 10'672 non-blank code lines from {fmt}, 128'431 lines after preprocessing

compile time about 31 ms
There is no measurable difference to baseline!

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

int main {
}
```

#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

IMPLEMENTATION STATUS

bumpy roads ahead ...

LANGUAGE / LIBRARY FEATURES

	gcc (trunk)	clang	msvc
Syntax specification	C++20	<= 8.0: Modules TS >= 9.0: TS and C++20	<= 19.22: Modules TS >= 19.23: C++20
Named modules	✅	✅	✅
Module partitions	✅	⛔	✅
Header modules	🟩 (undocumented)	🟩 (undocumented)	✅
Private mod. fragment	⛔	✅	✅
Name attachment	✅ weak model	✅ weak model	✅ strong model
#include → import	⛔	⛔	✅
__cpp_modules	⚠ 201810L	⛔	✅ 201907L
Modularized std library	⛔	⛔	🟩 (experimental)

BUILD SYSTEMS

Build systems with support for C++ modules are rare

- build2 (by Boris Kolpackov, build2.org)
  - supports clang, gcc, and msvc
- Evoke (by Peter Bindels, GitHub)
  - clang only
- MSBuild (by Microsoft, since msvc16.8, Visual Studio)
  - msvc only
- make
  - bring your own build rules, f.e. like Bloomberg's P2473

- more ?

RESOURCES

Papers

- Modules in C++, 2004, Daveed Vandevoorde
- Modules, 2012, Doug Gregor
- A Module System for C++, 2014, Gabriel Dos Reis, Mark Hall, Gor Nishanov
- C++ Modules TS, 2018, Gabriel Dos Reis
- Another take on Modules, 2018, Richard Smith
- Merging Modules, 2019, Richard Smith
- C++23 Draft

Contact

- dani@ngrt.de
- @DanielaKEngert on Twitter

Images: Bayeux Tapestry, 11th century, world heritage

source: WikiMedia Commons, public domain

QUESTIONS?

Ceterum censeo ABI esse frangendam