

+ 22

# Contemporary C++ in Action

DANIELA ENGERT



**Cppcon**  
The C++ Conference

20  
22 | A white mountain range icon with a yellow peak.  
September 12th-16th

# Outline

- Prologue
- Motivation
- The Experiment
- Demo
- My Conclusion
- Epilogue



# 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
- Observing participant in the C++ committee (learning novice) for 3 years (EWG, SG15)

- employed by



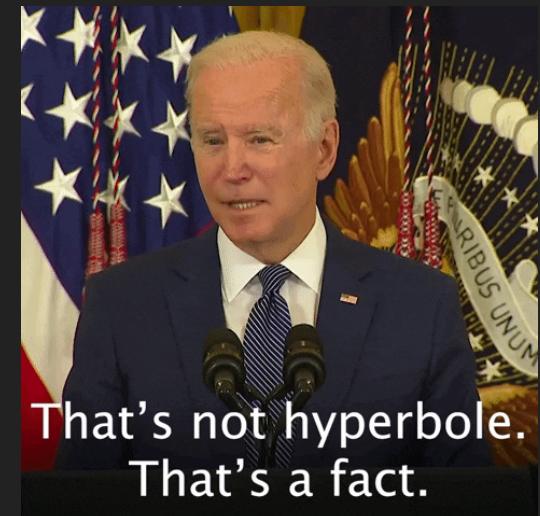
# Motivation



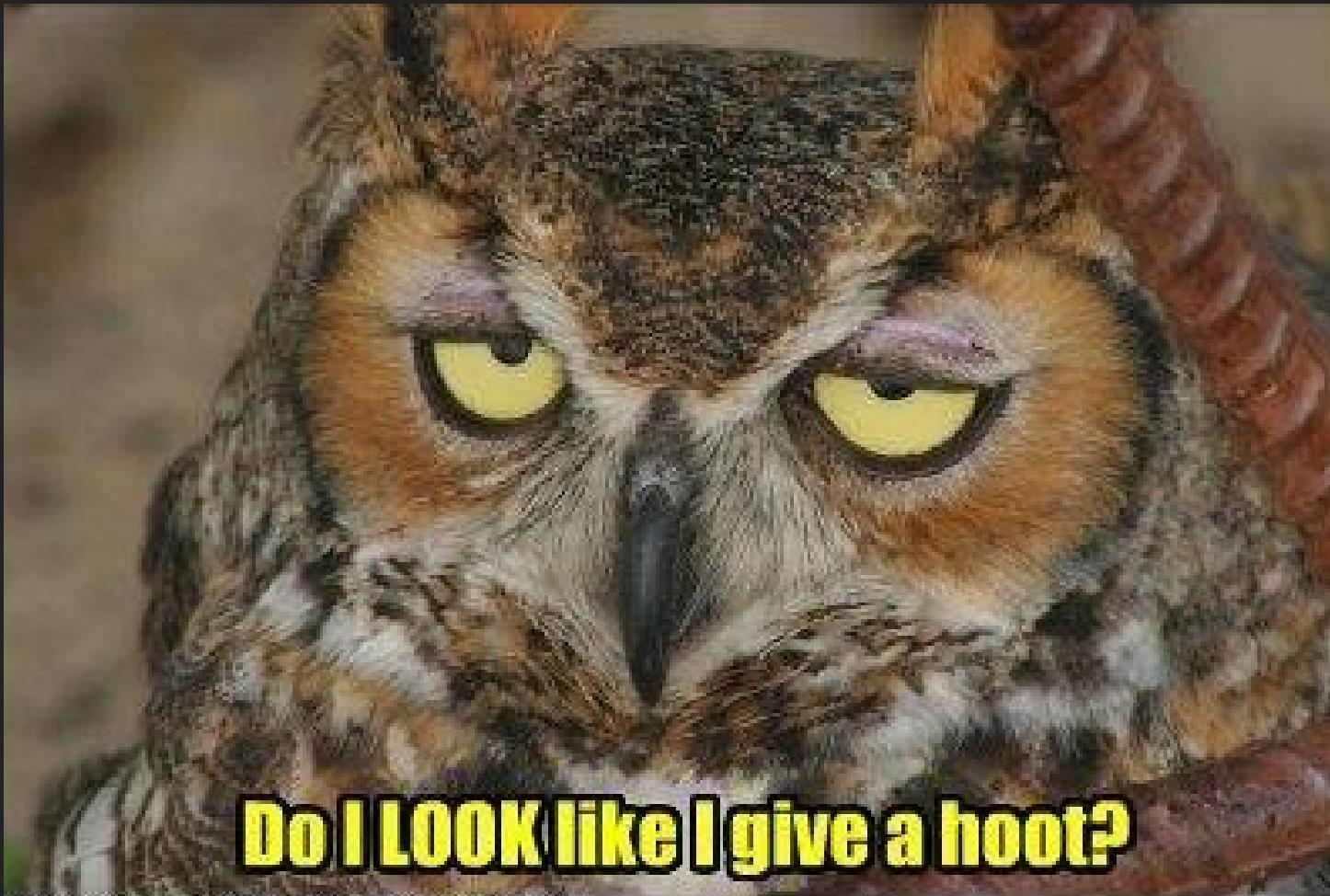
# THIS IS AN EXPERIMENT

Whenever I encounter C++ in any media, there are at least some people very vocal about the "facts":

- C++ is dead!
- The committee cares only about nerds
- The committee cares only about library writers
- The committee is too slow and doesn't deliver
- The committee is too fast and delivers half-baked junk
- The committee is a bureaucratic bunch of troglodytes detached from reality
- C++ 03 is all you need. Everything beyond that is just syntactic sugar



# IS THIS REALLY TRUE?



# THE REALITY CHECK

To disadvantage myself as much as possible I committed myself on writing an application from scratch and implement something non-trivial using as many

- core language features
- standard library features
- 3rd-party libraries

that I've never used before (or failed at doing so). This includes

- C++23, coroutines, concepts, infinite ranges

Using the results of older, pre-2020 WG21 work is ok.

The objective: make them work together as smoothly as possible.

HENCE THE TITLE OF MY TALK

Contemporary C++ in Action

# CONTEMPORARY C++ !

C++23 and  
C++20 and  
C++17 and  
C++14 and  
C++11



# IN ACTION ?

I'm an engineer, and the demo application should contain all the engineery stuff that I deal with every day:

- data collection
- data processing
- data visualization
- near realtime networking
- library usage
- library creation
- interface design



# The Experiment Specification Demo Code



# THE SPECIFICATION

## The server

- waits for clients to connect at any of a list of given endpoints
- when a client connects, observes a given directory for all files in there, repeats endlessly
- filters all GIF files which contain a video
- decodes each video 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 any 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

- performs a clean shutdown from all inputs that the user can interact with
- handles timeouts and errors properly and performs a clean shutdown

# DISCLAIMER

The following code

- contains significant amounts of C++23
- totally relies on C++20
- is following the  style

Allergy warning!

- There may be traces of compile-time programming
- Names are  cased and composed without underlines

# CONVENTIONS

Where necessary, assume these *using directives* and *namespace alias definitions* to be present in the nearest surrounding scope.

```
using namespace std::chrono_literals;
using namespace std::string_view_literals;

namespace fs = std::filesystem;
namespace rgs = std::ranges;
namespace vws = rgs::views;
```

- 'things' start with a capital letter
- 'things' of type kind (in particular *type aliases*) are prefixed with a lower-case 't'
- NSDM 'things' are postfixed with an underscore '\_'
- predicate queries start with a lower-case letter
- actions start with a lower-case letter

# THE LIBRARIES

Asio (<https://think-async.com>)

Semi-standardized, implements the 'Networking TS' (ISO/IEC TS 19216:2018)

provides asynchronous networking functions & an asynchronous execution framework

libav (<https://ffmpeg.org>)

Very popular open-source library to process virtually all media formats

SDL (<https://www.libsdl.org>)

A open-source library to handle multimedia I/O and minimal windowing, excels in simplicity

Two functions from the *in-house* codebase

- conversion from std::path to UTF-8 encoded narrow strings (not implementation-defined)
- command line options parser

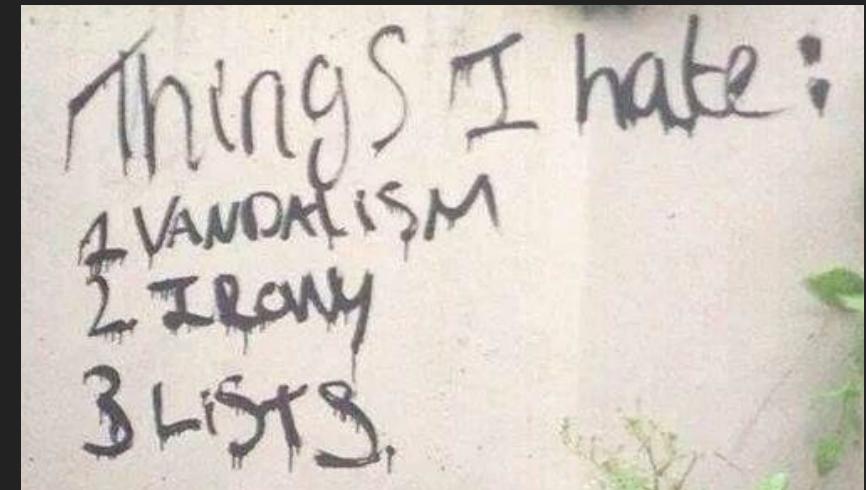
# C-LIBRARIES

libav and SDL are C libraries with heap-allocated objects and may therefore lead to

- manual memory management
- indeterminate responsibilities
- potential memory leaks
- potential memory retention
- accidental copying
- accidental sharing

So, instead of make them

```
// libav
AVCodecContext * pCodec;
avcodec_alloc_context3(&pCodec, ...);
avcodec_free_context(&pCodec);
```



```
// SDL
SDL_Window * pWindow = SDL_CreateWindow(...);
SDL_DestroyWindow(pWindow);
```

# Value Types!

# C\_RESOURCE WRAPPER

Wrap these C objects to get proper C++ objects with

- constructors, destructors → well-defined lifetime
- uniqueness → no sharing, copying is prohibited
- constness → integrity
- independence → easy reasoning
- composability → create higher level abstractions

```
namespace libav {  
    using Codec = stdex::c_resource<AVCodecContext, avcodec_alloc_context3, avcodec_free_context>;  
}  
  
namespace sdl {  
    using Window = stdex::c_resource<SDL_Window, SDL_CreateWindow, SDL_DestroyWindow>;  
}  
  
void f() {  
    libav::Codec MyCodec(...);  
    sdl::Window MyWindow(...);  
}
```

# VIDEO

```
// video frame generator
// wrap the libav (a.k.a. FFmpeg https://ffmpeg.org/) C API types and their
// assorted functions

namespace libav {
using Codec =
    stdex::c_resource<AVCodecContext, avcodec_alloc_context3, avcodec_free_context>;
using File =
    stdex::c_resource<AVFormatContext, avformat_open_input, avformat_close_input>;
using tFrame = stdex::c_resource<AVFrame, av_frame_alloc, av_frame_free>;
using tPacket = stdex::c_resource<AVPacket, av_packet_alloc, av_packet_free>;

// frames and packets are reference-counted and always constructed non-empty
struct Frame : tFrame {
    [[nodiscard]] Frame() : tFrame(constructed){};
    [[nodiscard]] auto dropReference() { return Frame::guard<av_frame_unref>(*this); }
};

struct Packet : tPacket {
    [[nodiscard]] Packet() : tPacket(constructed){};
    [[nodiscard]] auto dropReference() { return Packet::guard<av_packet_unref>(*this); }
};
} // namespace libav
```

# NETWORK - COMPOSED OPERATIONS

example: read with timeout

```
1 Thing {  
2   socket s  
3   timer t  
4  
5   timed_read(...)  
6   callback_socket(...)  
7   callback_timer(...)  
8   return_result(...)  
9   give_up()  
10 }
```

Requirements:

- Things must live long enough, at least until all asynchronous operations have run to completion and the result is announced to the caller of `timed_read`.
- The ownership of Things must necessarily be shared between all asynchronous operations and the caller.
- Some mechanism must allow for cancellation even when no execution agent is acting on behalf of either operation.



# NETWORK - COMPOSED OPERATIONS

Callbacks with individual parameters

```
async_read(socket, data,      Callable(      error_code, size_t ));  
  
timer::async_wait(      Callable(      error_code ));  
  
// composition by (distributed) state-machine
```

```
async_read(S, data, [self](      error_code, size_t result) {});  
  
T.async_wait([self](      error_code result) {});
```

# NETWORK - COMPOSED OPERATIONS

Callbacks with **packed** parameters

```
async_read(socket, data,      Callable(tuple<error_code, size_t>));
timer::async_wait(      Callable(tuple<error_code>));
// composition by (distributed) state-machine
```

```
async_read(S, data, [self](tuple<error_code, size_t> result) {});
T.async_wait([self](tuple<error_code> result) {});
```

# NETWORK - COMPOSED OPERATIONS

Return awaitables with packed parameters

```
async_read(socket, data) -> awaitable<tuple<error_code, size_t>>;  
  
timer::async_wait() -> awaitable<tuple<error_code>>;  
  
// composition t.b.d.
```

```
auto [error, transferred] = co_await async_read(S, data);  
  
auto [error] = co_await T.async_wait();
```

# NETWORK - COMPOSED OPERATIONS

Compose awaitables with automatic mutual cancellation

```
async_read(socket, data) -> awaitable<tuple1>;  
  
timer::async_wait() -> awaitable<tuple2>;  
  
operator||(awaitable<tuple1>, awaitable<tuple2>)  
-> awaitable<variant<tuple1, tuple2>>;
```

```
auto result = co_await (async_read(s, data) || t.async_wait());
```

# NETWORK - COMPOSED OPERATIONS

```
Thing : enable_shared_from_this {
    socket S;
    timer T;
    some data;
    Callable callback;

    ... // more
    void timed_read(Callable);
    void callback_socket(...);
    void callback_timer(...);
    void return_result(...);
    void giveup()
};

// implementation here ...

// on the call site:
// create a shareable Thing
// think about lifetimes
// do something with result
```

```
auto timed_read(socket & S, timer & T, some & data) {
    return async_read(S, data) || T.async_wait();
}

auto f() {
    socket S;
    timer T;
    some data;

    T.expires_after(...);
    auto result = co_await timed_read(S, T, data);
    // do something with result
}
```

# NETWORK - COMPOSED OPERATIONS

Chris Kohlhoff (creator of Asio):  
this is

why C++20 is the *awesomest* language  
for network programming

# NETWORK - COMPOSED OPERATIONS

Flatten the composed result

```
async_read(socket, data) -> awaitable<tuple<error_code, size_t>>;  
  
timer::async_wait() -> awaitable<tuple<error_code>>;  
  
auto operator||(awaitable<tuple1>, awaitable<tuple2>) // plus  
flatten(variant<tuple1, tuple2>) -> expected<size_t, error_code>;
```

```
auto result =  
    expect(co_await (async_read(s, data) || T.async_wait()));
```

# NETWORK - COMPOSED OPERATIONS

me:

this is

why **C++23** is the *even more awesomest*  
language for network programming!

# NETWORK

```
// Convenience types and functions to deal with the networking part of the
// Asio library https://think-async.com/Asio/

// The Asio library is also the reference implementation of the
// Networking TS (ISO/IEC TS 19216:2018, C++ Extensions for Library Fundamentals)
// Draft https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4771.pdf
// plus P0958, P1322, P1943, P2444
// plus executors as described in P0443, P1348, and P1393

namespace aex = asio::experimental;
namespace net {

    // basic networking types

    // customize the regular Asio types and functions such that they can be
    // - conveniently used in await expressions
    // - composed into higher level, augmented operations

    template <typename... Ts>
    using tResult      = std::tuple<std::error_code, Ts...>;
    using use_await   = asio::as_tuple_t<asio::use_awaitable_t<>>;
    using tSocket     = use_await::as_default_on_t<asio::ip::tcp::socket>;
    using tAcceptor   = use_await::as_default_on_t<asio::ip::tcp::acceptor>;
```

# EXECUTORS

I will not talk

about executors

# EXECUTION CONTEXTS

Asio provides this **higher-level abstraction** on top of executors as the **primary interface** for client code.

I will not talk about these.

Except for one feature:

User-definable services that can augment the ones provided by Asio

I add a "StopService" that gives access to a **std::stop\_source**.

This makes the extended `asio::io_service` similar to a `std::jthread`.

And allows **remote-controlled termination** of asynchronous operations.

# EXECUTOR

```
// Convenience types and functions to deal with the executor part of the
// Asio library https://think-async.com/Asio/

// The Asio library is also the reference implementation of the
// Networking TS (ISO/IEC TS 19216:2018, C++ Extensions for Library Fundamentals)
// Draft https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4771.pdf
// plus P0958, P1322, P1943, P2444
// plus executors as described in P0443, P1348, and P1393

namespace executor {
template <typename T>
constexpr inline bool Unfortunate = false;

// an Asio service that holds an 'std::stop_source'.
// each service is unique in every instance of execution contexts!

using ServiceBase = asio::execution_context::service;
struct StopService : ServiceBase {
    using key_type = StopService;

    static asio::io_context::id id;

    using ServiceBase::ServiceBase;
```

# SERVER

```
static constexpr auto SendTimeBudget = 100ms;

// start serving a list of given endpoints.
// each endpoint is served by an independent coroutine.

auto serve(asio::io_context & Context, net::tEndpoints Endpoints, const fs::path Source)
-> net::tExpectSize {
    std::size_t NumberOfAcceptors = 0;
    auto Error = std::make_error_code(std::errc::function_not_supported);

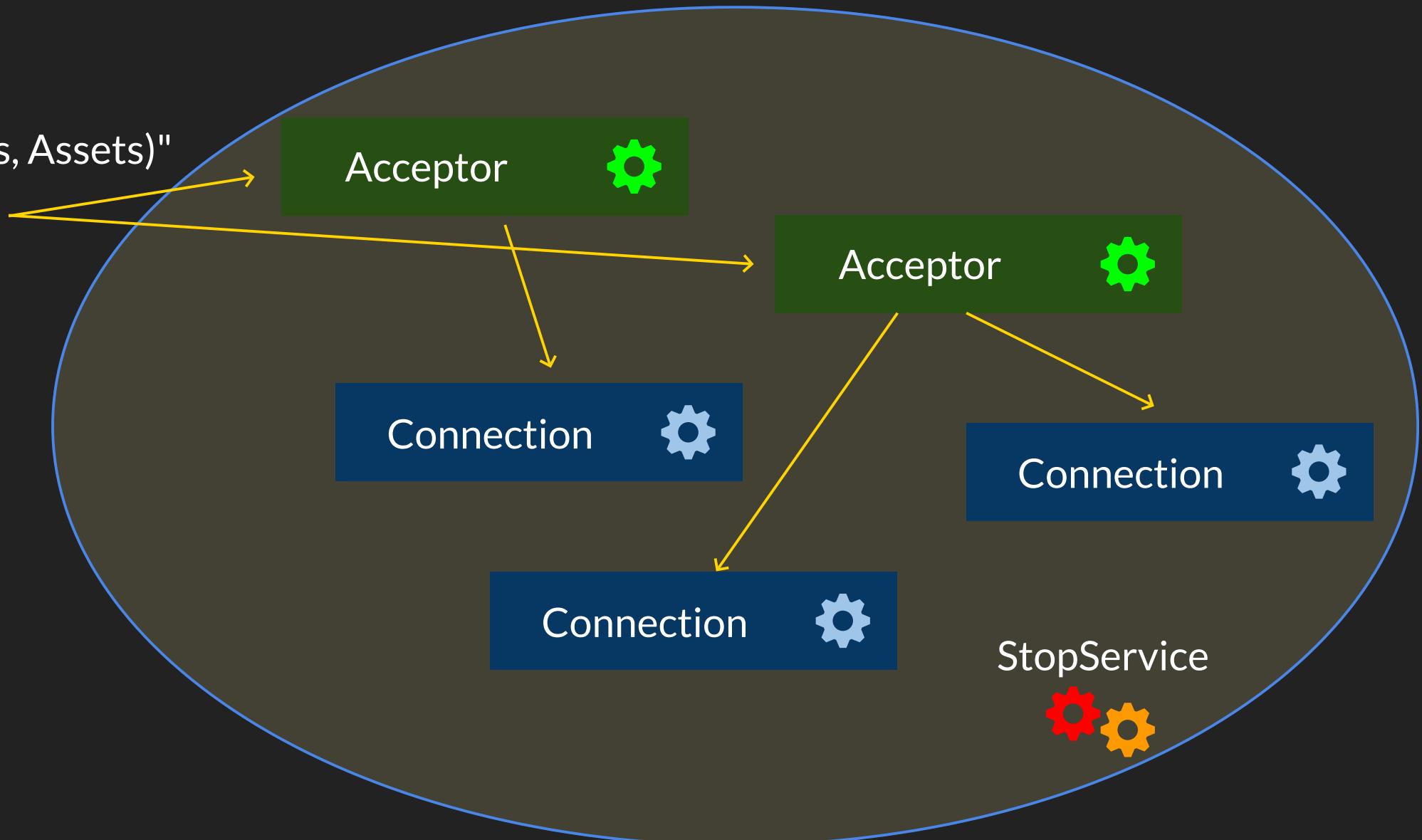
    for (const auto & Endpoint : Endpoints) {
        try {
            executor::commission(Context, acceptConnections,
                                  net::tAcceptor{Context, Endpoint}, Source);
            std::println("accept connections at {}", Endpoint.address().to_string());
            ++NumberOfAcceptors;
        } catch (const std::system_error & Ex) {
            Error = Ex.code();
        }
    }
    if (NumberOfAcceptors == 0)
        return std::unexpected{Error};
    return NumberOfAcceptors;
}
```

# SERVER - INDEPENDENT AGENTS ('ACTORS')

schedule

"serveEndpoints, Assets")

on IO context



# CLIENT

```
static constexpr auto ReceiveTimeBudget = 2s;
static constexpr auto ConnectTimeBudget = 2s;

// connects to the server and starts the top-level video receive-render-present loop.
// initiates an application stop in case of communication problems.

[[nodiscard]] auto showVideos(asio::io_context & Context, gui::FancyWindow Window,
                           net::tEndpoints Endpoints) -> asio::awaitable<void> {
    net::tTimer Timer(Context);
    Timer.expires_after(ConnectTimeBudget);
    if (net::tExpectSocket Socket = co_await net::connectTo(Endpoints, Timer)) {
        co_await rollVideos(std::move(Socket).value(), std::move(Timer),
                            std::move(Window));
    }
    executor::StopAssetOf(Context).request_stop();
}

// present a possibly infinite sequence of video frames until the spectator
// gets bored or problems arise.

[[nodiscard]] auto rollVideos(net::tSocket Socket, net::tTimer Timer,
                           gui::FancyWindow Window) -> asio::awaitable<void> {
    const auto WatchDog = executor::abort(Socket, Timer);
```

# GUI

```
// wrap the SDL (Simple Directmedia Layer https://www.libsdl.org/) C API types
// and their assorted functions

namespace sdl {
    using Window = stdex::c_resource<SDL_Window, SDL_CreateWindow, SDL_DestroyWindow>;
    using Renderer = stdex::c_resource<SDL_Renderer, SDL_CreateRenderer, SDL_DestroyRenderer>;
    using Texture = stdex::c_resource<SDL_Texture, SDL_CreateTexture, SDL_DestroyTexture>;
} // namespace sdl

namespace gui {
    enum width : int {};
    enum height : int {};

    // the most minimal GUI
    // capable of showing a frame with some decor for user interaction
    // renders the video frames
    struct FancyWindow {
        FancyWindow(width Width, height Height);

        void updateFrom(const video::FrameHeader & Header);
        void present(video::tPixels Pixels);

    private:
}
```

# USER INTERACTION

```
namespace handleEvents {
    static constexpr auto EventPollInterval = 50ms;

    // watch out for an interrupt signal (e.g. from the command line).
    // initiate an application stop in that case.

    [[nodiscard]] auto fromTerminal(asio::io_context & Context) -> asio::awaitable<void> {
        asio::signal_set Signals(Context, SIGINT, SIGTERM);
        const auto WatchDog = executor::abort(Signals);

        co_await Signals.async_wait(asio::use_awaitable);
        executor::StopAssetOf(Context).request_stop();
    }

    // the GUI interaction is a separate coroutine.
    // initiate an application stop if the spectator closes the window.

    [[nodiscard]] auto fromGUI(asio::io_context & Context) -> asio::awaitable<void> {
        net::tTimer Timer(Context);
        const auto WatchDog = executor::abort(Timer);

        do {
            Timer.expires_after(EventPollInterval);
```

# MAIN

```
static constexpr auto ServerPort      = uint16_t{ 34567 };  
static constexpr auto ResolveTimeBudget = 1s;  
  
int main() {  
    auto [MediaDirectory, ServerName] = caboodle::getOptions();  
    if (MediaDirectory.empty())  
        return -2;  
    const auto ServerEndpoints =  
        net::resolveHostEndpoints(ServerName, ServerPort, ResolveTimeBudget);  
    if (ServerEndpoints.empty())  
        return -3;  
  
    asio::io_context ExecutionContext; // we have executors at home  
    std::stop_source Stop;           // the mother of all stops  
    const auto schedule = executor::makeScheduler(ExecutionContext, Stop);  
  
    const auto Listening =  
        schedule(server::serve, ServerEndpoints, std::move(MediaDirectory));  
    if (not Listening)  
        return -4;  
  
    schedule(client::showVideos,  
            gui::FancyWindow(gui::width{ 1280 }, gui::height{ 1024 }), ServerEndpoints);
```

# APPLICATION - ALL 'ACTORS'

schedule

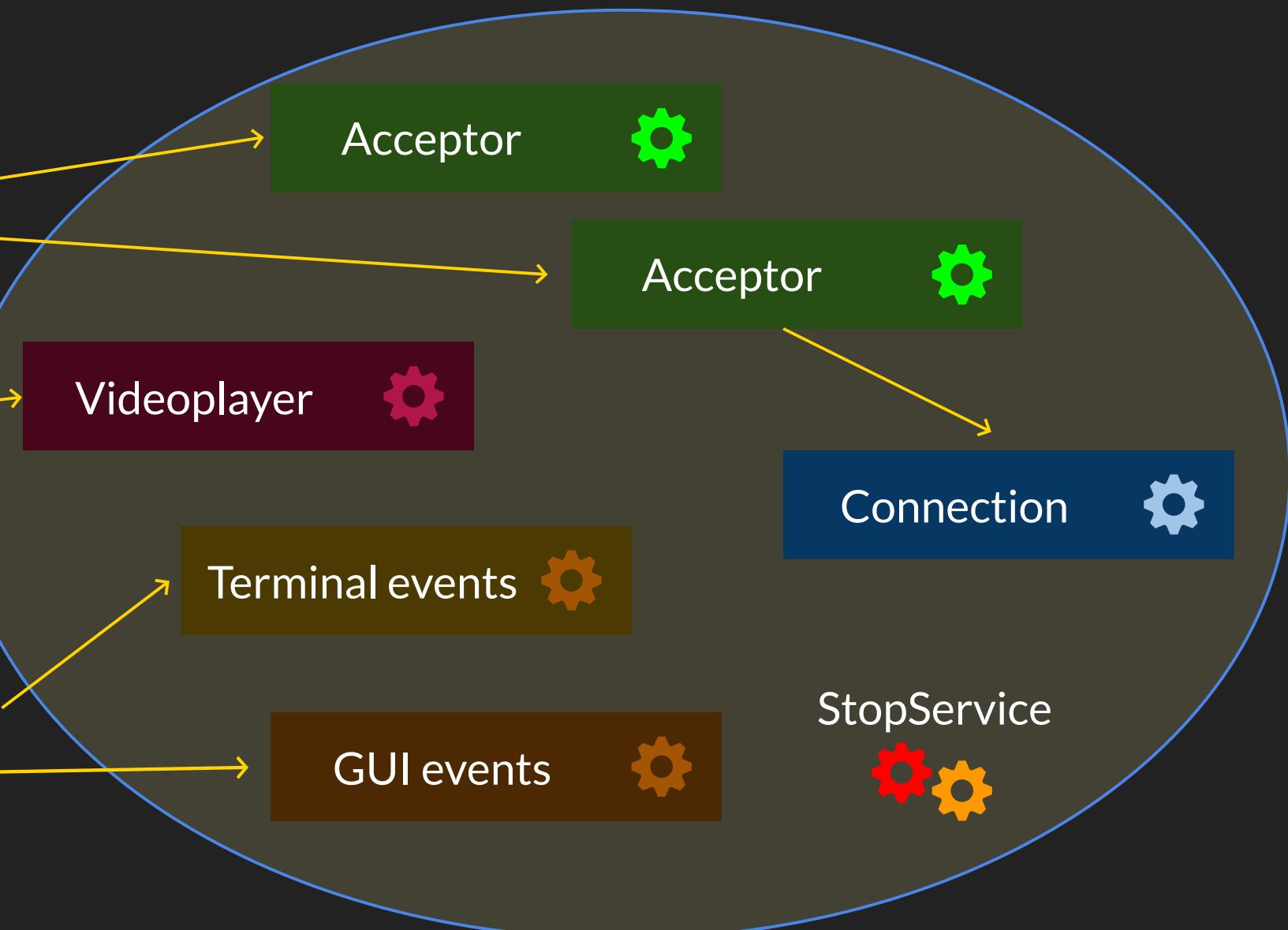
"server::serve(...)"

"client::showVideos(...)"

"handleEvents::fromTerminal()"

"handleEvents::fromGUI()"

on the IO context



# std::generator

std::expected

`std::println`

`std::stop_source`

`std::stop_token`

`std::stop_callback`

# std::span

ranges  
views

concepts  
requires-expressions

# coroutines

# structured bindings

# fold expressions

compile-time decisions  
compile-time functions  
compile-time reflection

# value types

# strong types

# compiler-generated closure types

# compiler-generated state-machines



one more compositional aspect

# source code composition

# MAIN (AGAIN)

```
/* =====
```

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:

- 6 named modules:
  - executor
  - generator
  - gui
  - net
  - the.whole.caboodle
  - video
- 1 cross-project header unit
  - c resource

precompiled and cached:

- 4 named modules from external libraries:
  - asio
  - boost.program\_options
  - libav
  - sdl
- the C++ standard library

# 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		✓		✓	✗		✓	✓	✓
Internal partition				✓	✗		✓	✗	✓
Private module fragment				✓					✓
Header unit	✓	✓		✓	✓	✓	✓	✓	

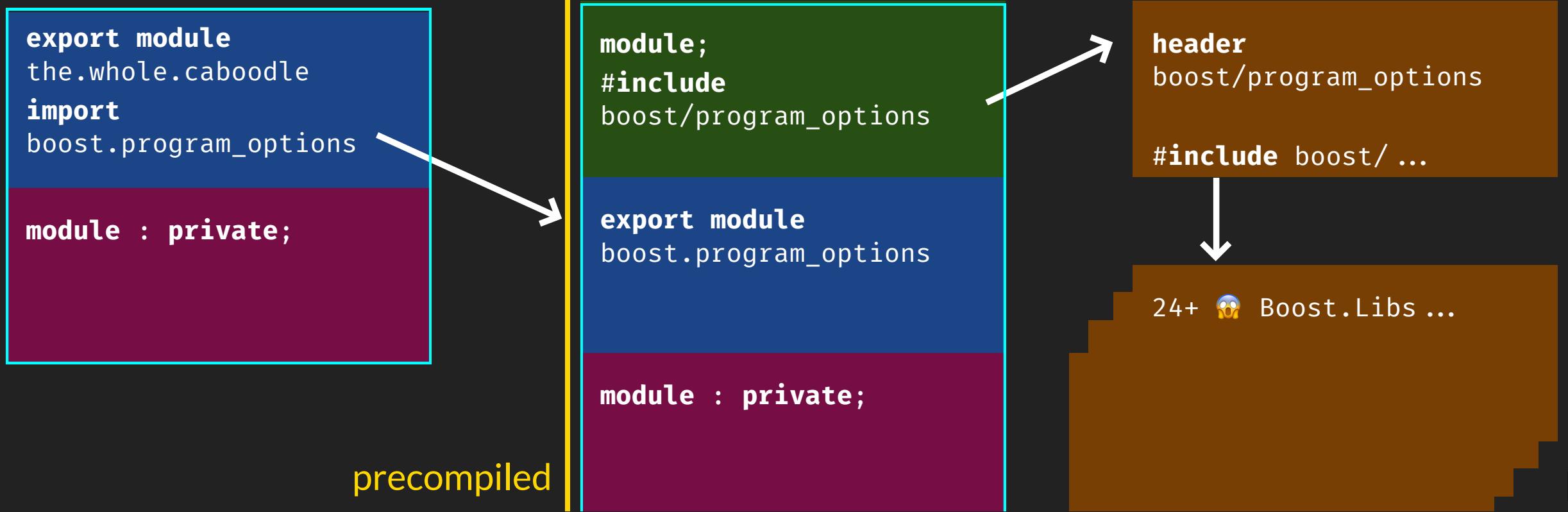
✓ unconditionally

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

# MODULE THE.WHOLE.CABOODLE

Consists of only the primary module interface unit (PMIU)

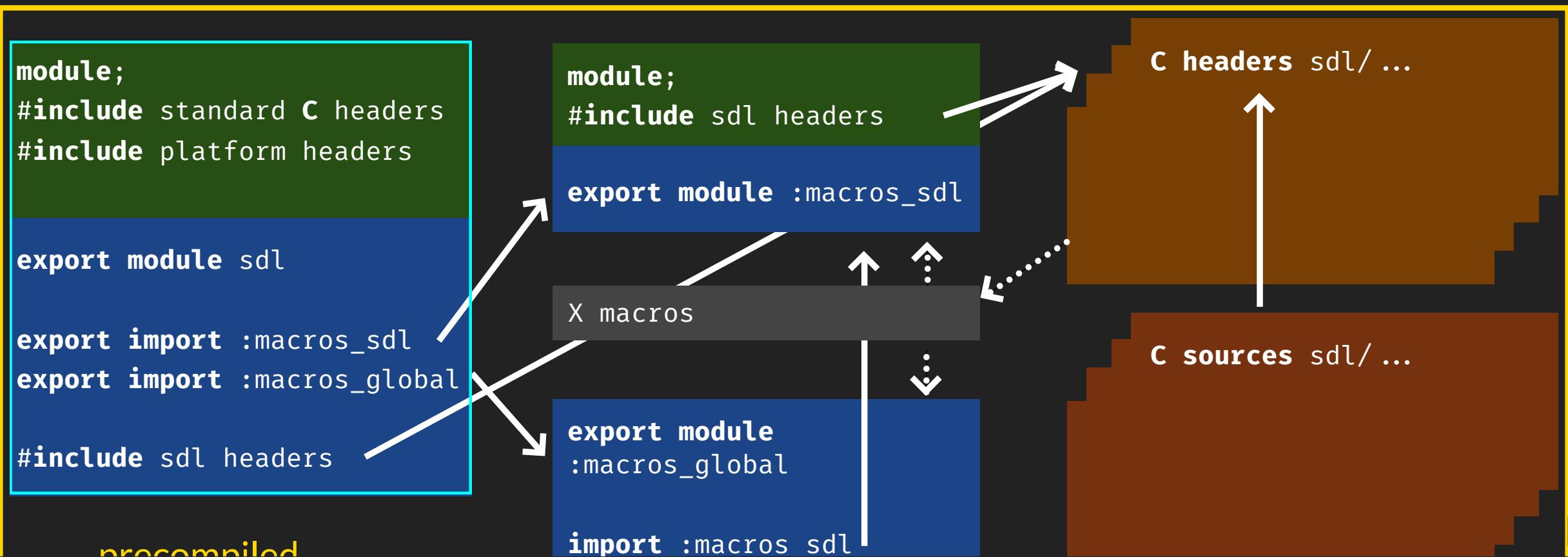
- exports 2 function declarations
- the full implementation is in the private module fragment (PMF)
- imports module boost.program\_options



# MODULE SDL

A multi-part module compiled as a static library that contains

- a global module fragment with all headers that must not be attached to the module
- the module purview with all **exported interfaces**
- the **implementation** is compiled separately with C semantics



# 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**

```
module;  
#include standard C headers  
listed in [tab:c.headers]
```

```
export module std  
#include standard C++ headers  
listed in [tab:headers.cpp]  
and [tab:headers.cpp.c]
```

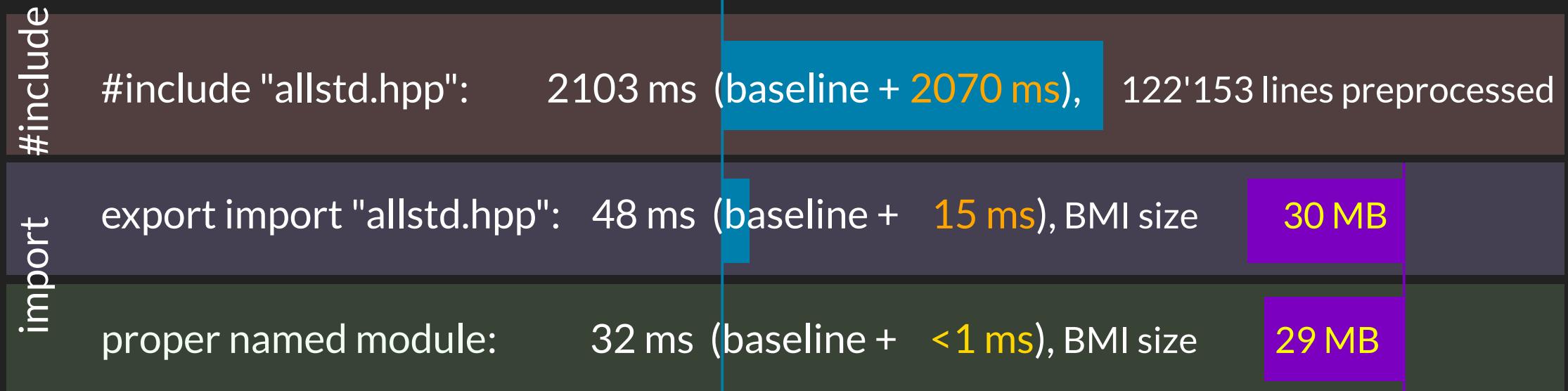
**headers** from the  
standard library (all)  
• C headers  
• C++ headers  
• wrapped C headers

precompiled

# STANDARD LIBRARY USAGE SCENARIOS

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

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

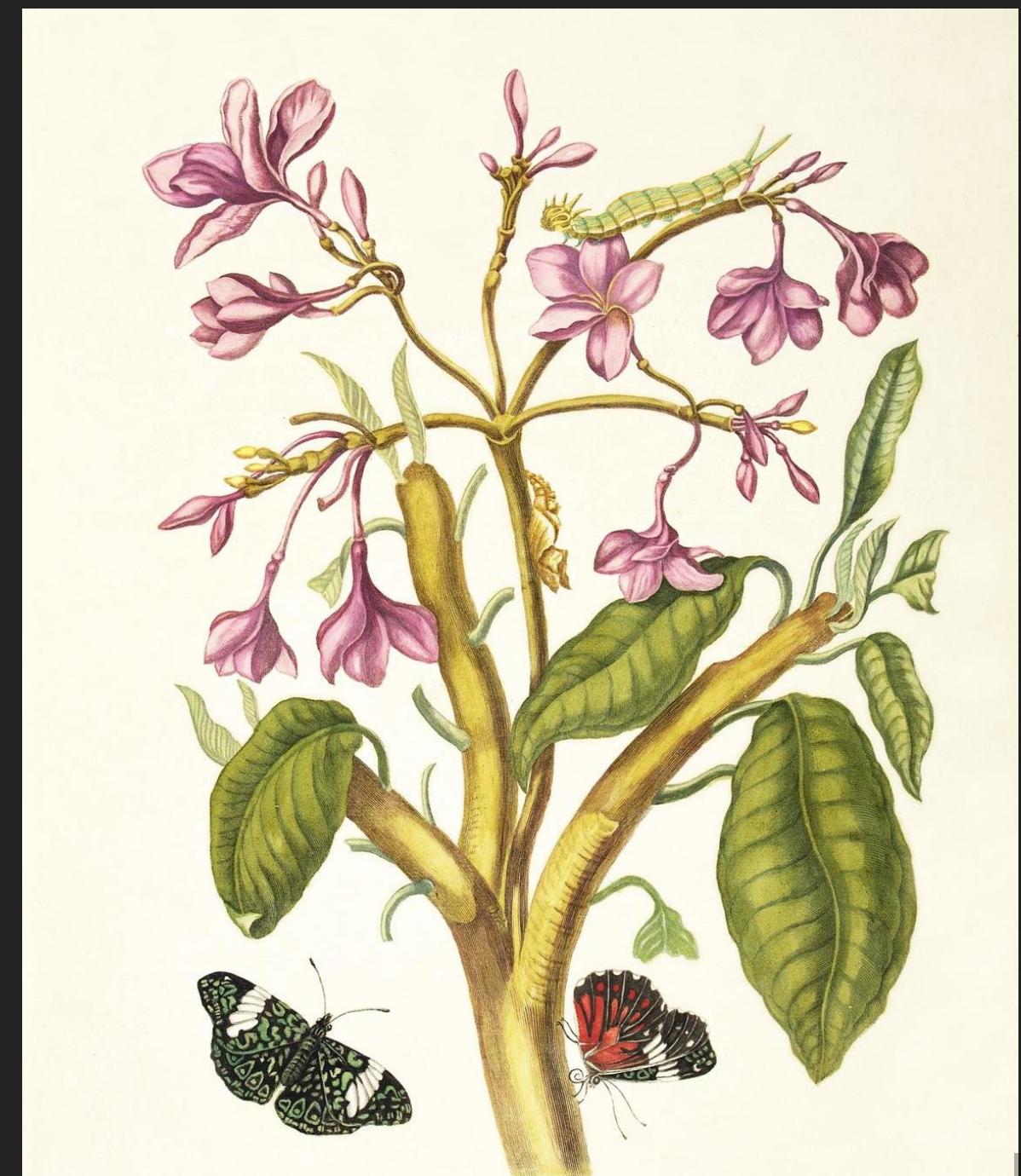


# DEMO CODE TIME

A man with short grey hair and a surprised or shocked expression, with his mouth wide open and eyes wide, is pointing his right index finger upwards towards the top left corner of the frame. He is wearing a dark t-shirt with a white mustache graphic on it. The background is filled with a dense pattern of green binary digits (0s and 1s) on a black background.



# My Conclusion



# CONTEMPORARY C++ IS



simple

# CONTEMPORARY C++ IS

concise



# CONTEMPORARY C++ IS



# CONTEMPORARY C++ IS



composable

# CONTEMPORARY C++ IS



enjoyable

# Epilogue



# RESOURCES

## Links

- Talking Async Ep1: Why C++20 is the Awesomest Language for Network Programming
- Talking Async Ep2: Cancellation in depth
- Demo code [github.com/DanielaE/CppInAction](https://github.com/DanielaE/CppInAction)

## Contact



dani@ngrt.de



@DanielaKEngert



DanielaE



Images: Maria Sibylla Merian (1705), Highlighting magic: Hana Dusíková



Ceterum censeo ABI esse frangendam