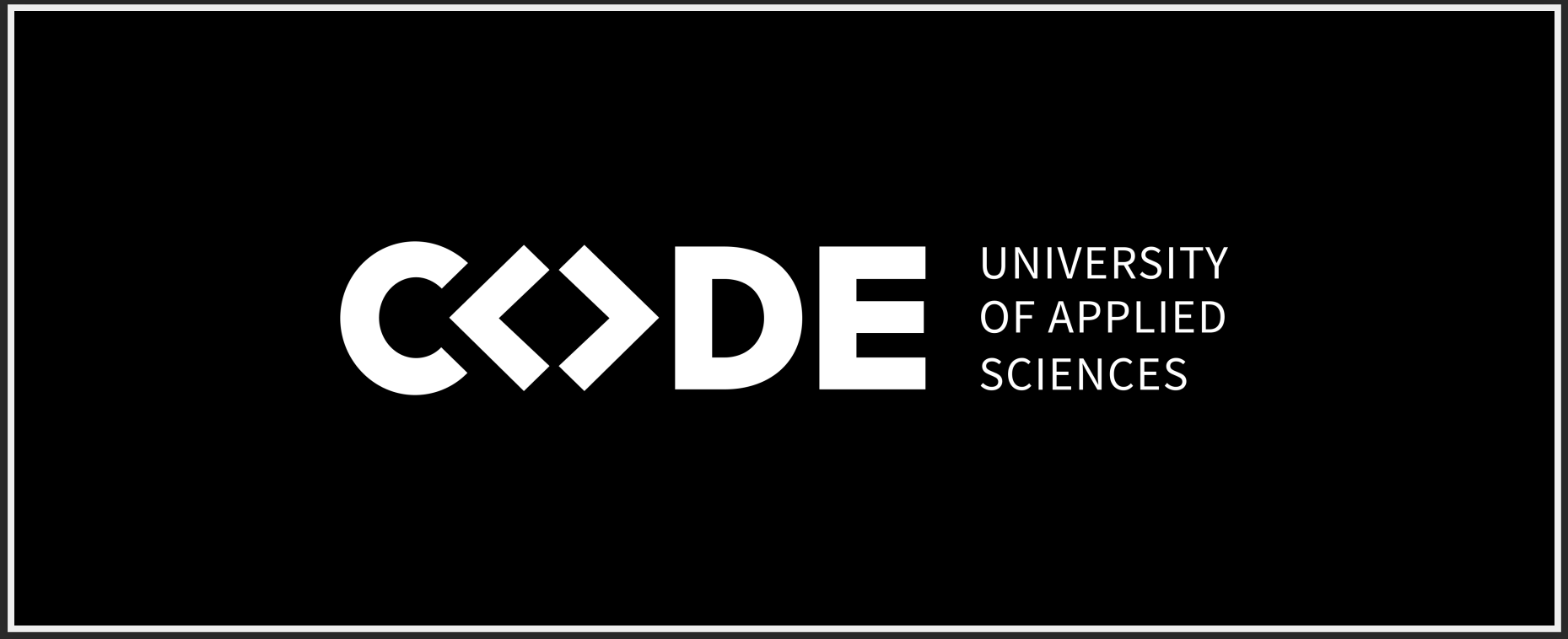
THE C++20 FIREHOSE TALK

Fabio Fracassi **fabio@fracassi.de**

https://code.berlin/ fabio.fracassi@code.berlin



Header:	<format>
Paper:	[p0645]
Feature Test:	__cpp_lib_format 2

Output Formatting

Typesafe text formatting with pythonesque format string syntax:

```
auto s = std::format("The answer is {:-^{}}!", 42, 20);
       // s == `The answer is ---------42---------!`
```

```
A | U+0041 { LATIN CAPITAL LETTER A }
Á | U+00C1 { LATIN CAPITAL LETTER A WITH ACUTE }
Á | U+0041 U+0301 { LATIN CAPITAL LETTER A } { COMBINING ACUTE ACCENT }
Ĳ | U+0132 { LATIN CAPITAL LIGATURE IJ }
Δ | U+0394 { GREEK CAPITAL LETTER DELTA }
Щ | U+0429 { CYRILLIC CAPITAL LETTER SHCHA }
א | U+05D0 { HEBREW LETTER ALEF }
ش | U+0634 { ARABIC LETTER SHEEN }
〉 | U+3009 { RIGHT-POINTING ANGLE BRACKET }
界 | U+754C { CJK Unified Ideograph-754C }
🦄 | U+1F921 { UNICORN FACE }
👨‍👩‍👧‍👦 | U+1F468 U+200D U+1F469 U+200D U+1F467 U+200D U+1F466 { Family: Man, Woman, Girl, Boy }
```

- Performant (faster than sprintf)

- Performant (faster than sprintf)
  - locale independent by default

- Performant (faster than sprintf)
  - locale independent by default
  - constexpr format strings allow for even more performance once we have constexpr strings

- Performant (faster than sprintf)
  - locale independent by default
  - constexpr format strings allow for even more performance once we have constexpr strings
- Little to no binary bloat (better than iostreams, close to printf)

- Performant (faster than sprintf)
  - locale independent by default
  - constexpr format strings allow for even more performance once we have constexpr strings
- Little to no binary bloat (better than iostreams, close to printf)
- extensible format string syntax

char8_t

`char8_t` is a new fundamental type to capture UTF-8 data.

char8_t…

```
auto process(std::u8string_view d);

char8_t const* hello_world = u8"hello 🌍";
process(hello_world);
```

`char8_t`…

- … never aliases with other types

```
auto process(std::u8string_view d);

char8_t const* hello_world = u8"hello 🌍";
process(hello_world);
```

`char8_t` …

- … never aliases with other types
- … is always unsigned

```
auto process(std::u8string_view d);

char8_t const* hello_world = u8"hello 🌍";
process(hello_world);
```

`char8_t` …

- … never aliases with other types
- … is always unsigned
- … can be overloaded upon

```
auto process(std::u8string_view d);

char8_t const* hello_world = u8"hello 🌍";
process(hello_world);
```

But you broke my code!

This code is valid in C++17 but no longer works in C++20!

```
char const* utf8 = u8"hello 🌍";
```

Yes, we want to be able to have first class UTF8 support

```
void process(std::u8string_view utf8);  // (1)
void process(std::string_view d);       // (2)

process("hello world"); // calls (2)
process(u8"hello 🌍");  // calls (1)
```

Header:	<chrono>
Paper:	[p1466]
Feature Test:	__cpp_lib_chrono 2

Time after Time

```
auto tp = local_days{sat[13]/jun/2020} + 11h + 25min;
auto zt = zoned_time{"America/Los_Angeles", tp};
std::cout << zt << '\n';
          // current time in LA 2020-06-13 02:25:00 PST
std::cout << std::format("%F %H:%M %z", zt) << '\n';
          // current time in LA 2020-06-13 02:25 -0900
```

All the functionality you want for working with time and dates

All the functionality you want for working with time and dates

- Seamlessly extents `std::chrono`

All the functionality you want for working with time and dates

- Seamlessly extents `std::chrono`
  - time/duration arithmetic works

All the functionality you want for working with time and dates

- Seamlessly extents `std::chrono`
  - time/duration arithmetic works
- Timezone handling

All the functionality you want for working with time and dates

- Seamlessly extents `std::chrono`
  - time/duration arithmetic works
- Timezone handling
  - loading/reloading databases

All the functionality you want for working with time and dates

- Seamlessly extents `std::chrono`
  - time/duration arithmetic works
- Timezone handling
  - loading/reloading databases
  - translating time points from to different calendars

All the functionality you want for working with time and dates

- Seamlessly extents `std::chrono`
  - time/duration arithmetic works
- Timezone handling
  - loading/reloading databases
  - translating time points from to different calendars
- Formatting output and input

All the functionality you want for working with time and dates

- Seamlessly extents `std::chrono`
  - time/duration arithmetic works
- Timezone handling
  - loading/reloading databases
  - translating time points from to different calendars
- Formatting output and input
  - works with streams and `std::format`

Header:	<source_location>
Paper:	[p1208]
Feature Test:	__cpp_lib_source_location 2

Source Location

```
template <typename... Ts> struct log {
    log(Ts&&... ts, std::source_location const& loc
                        = std::source_location::current())
    {
        std::clog << std::format("[{}:{:>5}] {}"
                        , loc.file_name(), loc.line()
                        , std::format(std::forward<Ts>(ts)...));
    }
};
template <typename... Ts> log(Ts&&...) -> log<Ts...>;
```

```
std::string msg = "Log data";
log("Important {}!", msg);
```

```
[app.cpp:  301] Important Log data!
```

Look, ma! No Macros!

Header:	<concepts>
Feature Test:	__cpp_concepts 201811 __cpp_lib_concepts 20

Concepts

Constraining allowed types for Template parameters.

```
template<typename T>
concept my_equality_comparable = requires(T a, T b) {
  { a == b } -> bool;
};

template<std::range C, my_equality_comparable V>
auto find(C const& c, V const& v) {  /* ... */ }
```

generates better error messages:

```
Widget w = ...;
auto r = find(w, "MainView");
```

Will give you something like:

```
error: cannot call find with Widget
  note: concept std::forward_range<...>
        was not satisfied for Widget
```

More complex constraints can handled

```
template<std::range C, typename V>
auto find(C const& c, V const& v)
    requires std::equality_comparable_with<C::value_type, V>
{ /* ... */ }
```

no function definition checking

No check if template definition uses **only** expressions as defined by the requirements

```
template<std::range C, typename V>
auto find(C const& c, V const& v)
    requires std::equality_comparable_with<C::value_type, V>
{
  // ...
  log("found value: {}\n", v);
}
```

no function definition checking

No check if template definition uses **only** expressions as defined by the requirements

(by design)

```
template<std::range C, typename V>
auto find(C const& c, V const& v)
    requires std::equality_comparable_with<C::value_type, V>
{
  // ...
  log("found value: {}\n", v);
}
```

no function definition checking

No check if template definition uses **only** expressions as defined by the requirements

(by design)

```
template<std::range C, typename V>
auto find(C const& c, V const& v)
    requires std::equality_comparable_with<C::value_type, V>
{
  // ...
  log("found value: {}\n", v);
}
```

works, but `equality_comparable_with` does not require `log(/*...*/, v)` to be valid

generic functions

`auto` is now allowed in function parameters

```
auto fun(auto t) {
  return t;
}
```

```
template<typename T> auto fun(T t) {
  return t;
}
```

type placeholders

concept `auto` allowed anywhere that `auto` was allowed before

```
auto sum(std::range auto const& c) {
  my::addable auto s = std::range_value_t<decltype(c)>{};
  for(auto const& e : c) { s += e; }
  return s;
}
```

Header:	<ranges> and others
Paper:	[p0896]
Feature Test:	__cpp_lib_ranges 20

Ranges

Compose algorithms to fully exploit their power:

```
std::vector<int> vi{1,2,3,4,5,6,7,8,9,10};
auto rng = vi | view::filter([](int i){ return i % 2 == 1; })
              | view::transform([](int i) {
                                  return std::to_string(i);
                                });
// rng == {"2","4","6","8","10"};
```

- Basic views

- Basic views
  - `std::filter_view`, `std::transform_view`

- Basic views
  - `std::filter_view`, `std::transform_view`
  - `std::iota_view`

- Basic views
  - `std::filter_view`, `std::transform_view`
  - `std::iota_view`
  - `std::join_view`, `std::split_view`

ranges can be two iterators, or an iterator/sentinel pair

ranges can be two iterators, or an iterator/sentinel pair

this allows for efficient unbounded ranges

```
int sum = accumulate(view::iota(1)
                   | view::transform([](int i){return i*i;})
                   | view::take(5), 0);
// sum = 1 + 4 + 9 + 16 + 25 = 55
```

- Basic Concepts

- Basic Concepts
  - `std::copyable`, `std::regular`

- Basic Concepts
  - `std::copyable`, `std::regular`
  - `std::range`

- Basic Concepts
  - `std::copyable`, `std::regular`
  - `std::range`
- Range Access and Traits

- Basic Concepts
  - `std::copyable`, `std::regular`
  - `std::range`
- Range Access and Traits
- Range-ified STL Algorithms
  - `std::ranges::sort`, `std::ranges::find`

```
auto vi = std::vector<int>{10,2,3,5,6,4,7,9,0,8};
auto it = std::ranges::find(vi, 9);
std::ranges::sort(vi);

std::pair<int, const char*> pairs[] = {{2, "foo"}, {1, "bar"}, {0
std::ranges::sort ( pairs, std::less<>{}
                  , [](auto const& p) { return p.first; } );
```

A few useful features have to wait until C++23

`zip_view`, `enumerate`, `ranges::to`

```
auto v = zip_view(iota(0), v)
         | view::filter([](auto const& e) {
                        return std::get<0>(e) % 2 == 1;
                    })
         | ranges::to<std::vector>;
```

```
for(auto [i, e] : enumerate(v)) {
  std::cout << std::format("Item #{}: `{}`", i, e);
}
```

Small improvements in algorithms

```
auto n1 = std::list.remove(/* ... */);
auto n2 = std::forward_list.unique(/* ... */);
auto n3 = std::list.remove_if(/* ... */);
```

number of removed elements

```
std::erase_if(container, [](auto const& e) { return e % 2 == 1; }
std::erase(c, 0);
// same as c.erase(remove(c.begin(), c.end(), 0), c.end());
```

New execution policy: `std::execution::unseq` for vectorization.

Header:	
Paper:	[p0122]
Feature Test:	__cpp_lib_span 20

std::span

Vocabulary type for contiguous views. Think `std::string_view` but for arrays/vector

```
void process3(std::span<int const, 3> ints3);
void process8(std::span<int const, 8> ints8);
void processd(std::span<int> ints);

int arr[] = {1, 2, 3};

process3(arr)
process8(arr)
process(arr)
```

Header:	
Paper:	[p0122]
Feature Test:	__cpp_lib_span 20

std::span

Vocabulary type for contiguous views. Think `std::string_view` but for arrays/vector

```
void process3(std::span<int const, 3> ints3);
void process8(std::span<int const, 8> ints8);
void processd(std::span<int> ints);

int arr[] = {1, 2, 3};

process3(arr)
process8(arr)
process(arr)
```

✅ statically checked

Header:	
Paper:	[p0122]
Feature Test:	__cpp_lib_span 20

std::span

Vocabulary type for contiguous views. Think `std::string_view` but for arrays/vector

```
void process3(std::span<int const, 3> ints3);
void process8(std::span<int const, 8> ints8);
void processd(std::span<int> ints);

int arr[] = {1, 2, 3};

process3(arr)
process8(arr)
process(arr)
```

❌ Compile Error - not enough elements

Header:	
Paper:	[p0122]
Feature Test:	__cpp_lib_span 20

std::span

Vocabulary type for contiguous views. Think `std::string_view` but for arrays/vector

```
void process3(std::span<int const, 3> ints3);
void process8(std::span<int const, 8> ints8);
void processd(std::span<int> ints);

int arr[] = {1, 2, 3};

process3(arr)
process8(arr)
process(arr)
```

✅ OK - dynamic `.size() == 3`

- subspans

- subspans
  - `first(n)`/`first<n>()`

- subspans
  - `first(n)`/`first<n>()`
  - `last(n)`/`last<n>()`

- subspans
  - `first(n)`/`first<n>()`
  - `last(n)`/`last<n>()`
  - `subspan(offset, n)`/`subspan<n>(offset)`

Useful as a bridge to low level code

```
span<std::byte const> c_raw = std::as_bytes(s);
span<std::byte> raw = std::as_writeable_bytes(s);

write(socket, raw.data(), raw.size());
```

Header:	<memory>
Feature Test:	__cpp_lib_smart_ptr_for_overwrite 20

Uninitialized Smart Pointers

array versions `std::make_unique`, `std::alloc_shared`, etc always initialize the array

```
auto a = std::make_unique_for_overwrite<int[]>(1'000);
    // do not read from a until it is filled
```

bit fiddleing

bit_cast

```
float f = 3.14;
auto u = std::bit_cast<std::uint32_t>(f);

constexpr std::uint32_t u2 = 0xabacadabul;
constexpr auto f2 = std::bit_cast<float>(u2);
```

detect endianess

```
if (std::endian::native == std::endian::big) {
    // handle big endian
}
```

bit operations like `rotl`/`rotr`, `popcount`, ...

Bindings & Lambdas

Lambdas are great! Let us use them everywhere!

Seriously: There are restrictions on where lambdas could not be used, where it would be useful to do so. Several of those restrictions have been lifted

Lambda Template syntax

```
auto f = []<typename T>(std::vector<T>){}
```

Pack Expansion in Lambda captures

```
template<class F, class... Args>
auto delay_invoke(F f, Args... args) {
    return [f=std::move(f), ...args=std::move(args)]()
              -> decltype(auto)
           {
              return std::invoke(f, args...);
           };
}
```

Header:	<functional>
Paper:	[p0356]
Feature Test:	__cpp_lib_bind_front 20

bind_front

```
struct MyStrategy {
  double process(std:string, double);
};
std::unique_ptr<MyStrategy> createStrategy();

auto s2 = std::bind_front(&MyStrategy::process, createStrategy())
double r1 = s2("free", 10.0);
auto s1 = std::bind_front(s2, "bound");
double r2 = s1(2.0);
```

Modules

```
export module my.module;

import some.module

int internal_helper(int i) {
  return some_module::calculate(i);
}

export namespace my_module {
  int functionality(int i) {
    return internal_helper(i);
  }
}
```

```
import my.module

int main() {
  return my_module::functionality(42);
}
```

Modules

```
export module my.module;

import some.module

int internal_helper(int i) {
  return some_module::calculate(i);
}

export namespace my_module {
  int functionality(int i) {
    return internal_helper(i);
  }
}
```

```
import my.module

int main() {
  return my_module::functionality(42);
}
```

Modules

```
export module my.module;

import some.module

int internal_helper(int i) {
  return some_module::calculate(i);
}

export namespace my_module {
  int functionality(int i) {
    return internal_helper(i);
  }
}
```

```
import my.module

int main() {
  return my_module::functionality(42);
}
```

Modules

```
export module my.module;

import some.module

int internal_helper(int i) {
  return some_module::calculate(i);
}

export namespace my_module {
  int functionality(int i) {
    return internal_helper(i);
  }
}
```

```
import my.module

int main() {
  return my_module::functionality(42);
}
```

Modules

```
export module my.module;

import some.module

int internal_helper(int i) {
  return some_module::calculate(i);
}

export namespace my_module {
  int functionality(int i) {
    return internal_helper(i);
  }
}
```

```
import my.module

int main() {
  return my_module::functionality(42);
}
```

Modules

```
export module my.module;

import some.module

int internal_helper(int i) {
  return some_module::calculate(i);
}

export namespace my_module {
  int functionality(int i) {
    return internal_helper(i);
  }
}
```

```
import my.module

int main() {
  return my_module::functionality(42);
}
```

Huge changes to …

Huge changes to …

- … the physical build model

Huge changes to …

- … the physical build model
- … the linkers

Huge changes to …

- … the physical build model
- … the linkers
- … the build system(s)

Huge changes to …

- … the physical build model
- … the linkers
- … the build system(s)
- … our thinking

There is no Silver Bullet

Modules will not …

There is no Silver Bullet

Modules will not …

- … magically make your code more modular

There is no Silver Bullet

Modules will not …

- … magically make your code more modular
- … magically improve your build times

There is no Silver Bullet

Modules will not …

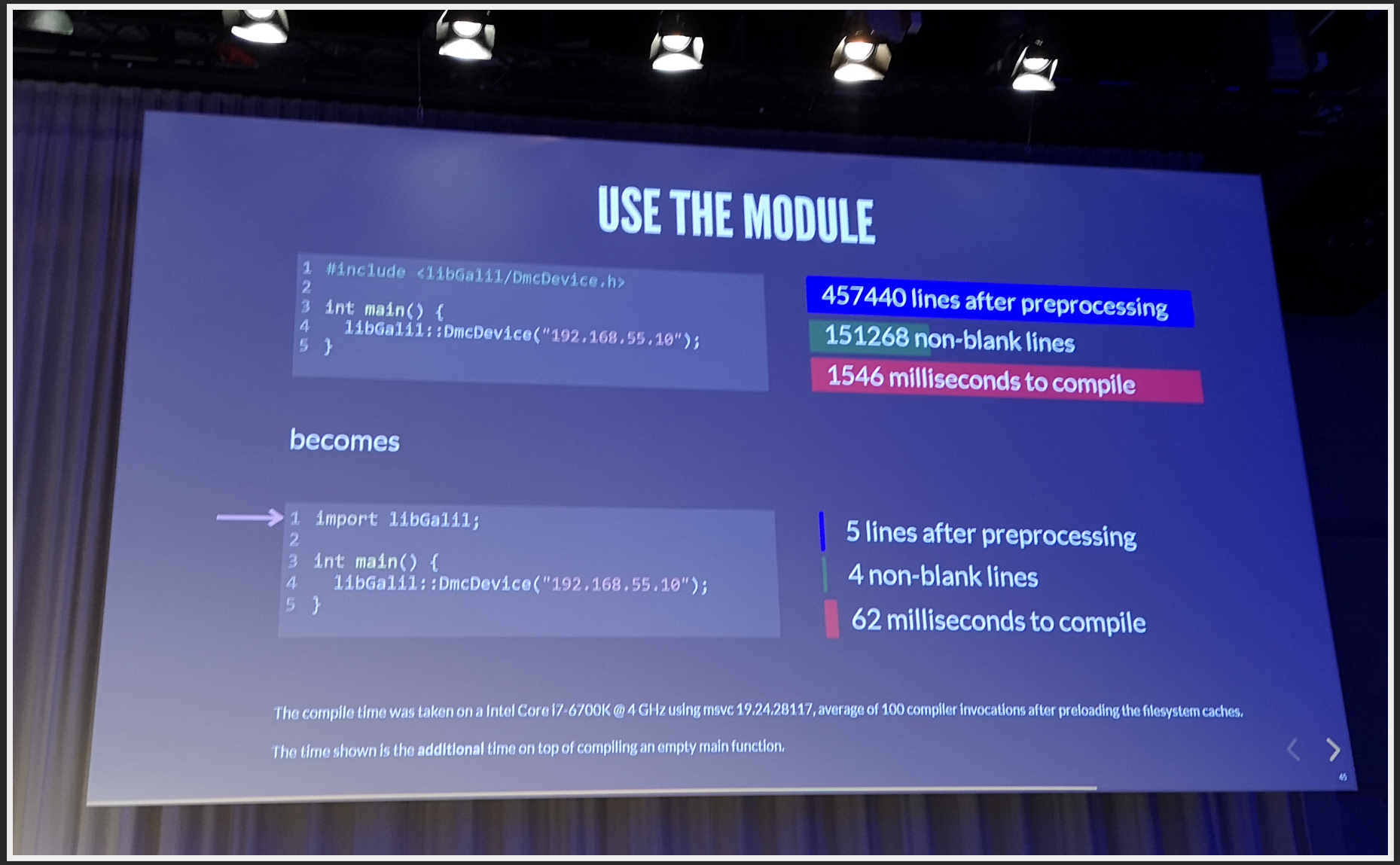
- … magically make your code more modular
- … magically improve your build times
- … magically materialize a package manager system

There is no Silver Bullet

Modules will not …

- … magically make your code more modular
- … magically improve your build times
- … magically materialize a package manager system
- … make interface/implementation files obsolete

but ...


USE THE MODULE
1 #include <libGali1/DmcDevice.h>
2
3 int main() {
4 libGali1::DmcDevice("192.168.55.10");
5 }
457440 lines after preprocessing
151268 non-blank lines
1546 milliseconds to compile
becomes
1 import libGali1;
2
3 int main() {
4 libGali1::DmcDevice("192.168.55.10");
5 }
5 lines after preprocessing
4 non-blank lines
62 milliseconds to compile
The compile time was taken on a Intel Core i7-6700K @ 4 GHz using msvc 19.24.28117, average of 100 compiler invocations after preloading the filesystem caches.
The time shown is the additional time on top of compiling an empty main function.

coroutines

```
nonstd::cppcoro::generator<const std::uint64_t> fibonacci() {
    std::uint64_t a = 0, b = 1;
    while (true) {
        co_yield b;
        auto tmp = a; a = b; b += tmp;
    }
}

void usage() {
    for (auto i : fibonacci()) {
        if (i > 1'000'000) { break; }
        std::cout << i << std::endl;
    }
}
```

coroutines

```
nonstd::cppcoro::generator<const std::uint64_t> fibonacci() {
    std::uint64_t a = 0, b = 1;
    while (true) {
        co_yield b;
        auto tmp = a; a = b; b += tmp;
    }
}

void usage() {
    for (auto i : fibonacci()) {
        if (i > 1'000'000) { break; }
        std::cout << i << std::endl;
    }
}
```

Start the coroutine

coroutines

```
nonstd::cppcoro::generator<const std::uint64_t> fibonacci() {
    std::uint64_t a = 0, b = 1;
    while (true) {
        co_yield b;
        auto tmp = a; a = b; b += tmp;
    }
}

void usage() {
    for (auto i : fibonacci()) {
        if (i > 1'000'000) { break; }
        std::cout << i << std::endl;
    }
}
```

`generator` produces/is an infinite range

coroutines

```
nonstd::cppcoro::generator<const std::uint64_t> fibonacci() {
    std::uint64_t a = 0, b = 1;
    while (true) {
        co_yield b;
        auto tmp = a; a = b; b += tmp;
    }
}

void usage() {
    for (auto i : fibonacci()) {
        if (i > 1'000'000) { break; }
        std::cout << i << std::endl;
    }
}
```

produce first value (1)

coroutines

```
nonstd::cppcoro::generator<const std::uint64_t> fibonacci() {
    std::uint64_t a = 0, b = 1;
    while (true) {
        co_yield b;
        auto tmp = a; a = b; b += tmp;
    }
}

void usage() {
    for (auto i : fibonacci()) {
        if (i > 1'000'000) { break; }
        std::cout << i << std::endl;
    }
}
```

output that value

coroutines

```
nonstd::cppcoro::generator<const std::uint64_t> fibonacci() {
    std::uint64_t a = 0, b = 1;
    while (true) {
        co_yield b;
        auto tmp = a; a = b; b += tmp;
    }
}

void usage() {
    for (auto i : fibonacci()) {
        if (i > 1'000'000) { break; }
        std::cout << i << std::endl;
    }
}
```

calculate next value

coroutines

```
nonstd::cppcoro::generator<const std::uint64_t> fibonacci() {
    std::uint64_t a = 0, b = 1;
    while (true) {
        co_yield b;
        auto tmp = a; a = b; b += tmp;
    }
}

void usage() {
    for (auto i : fibonacci()) {
        if (i > 1'000'000) { break; }
        std::cout << i << std::endl;
    }
}
```

...

coroutines

```
nonstd::cppcoro::generator<const std::uint64_t> fibonacci() {
    std::uint64_t a = 0, b = 1;
    while (true) {
        co_yield b;
        auto tmp = a; a = b; b += tmp;
    }
}

void usage() {
    for (auto i : fibonacci()) {
        if (i > 1'000'000) { break; }
        std::cout << i << std::endl;
    }
}
```

done!

coroutines

```
nonstd::cppcoro::generator<const std::uint64_t> fibonacci() {
    std::uint64_t a = 0, b = 1;
    while (true) {
        co_yield b;
        auto tmp = a; a = b; b += tmp;
    }
}

void usage() {
    for (auto i : fibonacci()) {
        if (i > 1'000'000) { break; }
        std::cout << i << std::endl;
    }
}
```

Lewis Baker's cppcoro Library

```
nonstd::cppcoro::task<> tcp_echo_server() {
    std::array<char, 1024> buf;
    for (;;) {
        size_t n = co_await socket.async_read_some(buffer(buf.dat
        co_await socket.async_write(buffer(buf.data(), n));
    }
}
```

- Coroutines are a language feature
  - not much library (yet)
- much of C++23 Library work will likely be in this area
  - Executors p0443

Threads & Syncronisation

Header:	<thread>, <stop_token>
Paper:	[p0660]
Feature Test:	__cpp_lib_jthread 20

jthread

```
int main() {
    std::jthread work{
      []{ /* */ }};


    /* */









}
```

```
//  int main() {
//      std::thread work{
//        []{ /* */ }};
//
//      try {
//        /* */
//      } catch (...) {
//          work.join()
//          throw;
//      }
//
//      work.join()
//  }
```

```
void main() {
    std::jthread t([] (std::stop_token stoken) {
            while (!stoken.stop_requested()) {
                //...
            }
        });
    //...
}
```

Updates of locks, mutexes and condition variables to handle `stop_token`s

New Synchronisation

Latches, Barriers, Semaphores, …

Upgrading the atomic arsenal

… `atomic_wait`, …

`std::atomic<float>`, `std::atomic<double>` and `std::atomic<long double>` now work

atomic_shared_ptr

```
std::atomic_shared_ptr p = std::make_shared<int>(42);
auto ta = std::jthread([&p]{
              p = std::make_shared<int>(1)
          });
auto tb = std::jthread([&p]{
              p = std::make_shared<int>(2)
          });
```

atomic_ref

```
int normal = 42;
std::atomic_ref aref(normal);

aref.store(4);
```

constexpr all the things

consteval

Enforce that a *function* is never called at runtime

```
consteval auto always_constexpr(int n) {
                return n;
            }
constexpr int ce      = always_constexpr(100);
          int runtime = 1;
          int not_ce  = always_constexpr(runtime);
```

consteval

Enforce that a *function* is never called at runtime

```
consteval auto always_constexpr(int n) {
                return n;
            }
constexpr int ce      = always_constexpr(100);
          int runtime = 1;
          int not_ce  = always_constexpr(runtime);
```

consteval

Enforce that a *function* is never called at runtime

```
consteval auto always_constexpr(int n) {
                return n;
            }
constexpr int ce      = always_constexpr(100);
          int runtime = 1;
          int not_ce  = always_constexpr(runtime);
```

✅ OK

consteval

Enforce that a *function* is never called at runtime

```
consteval auto always_constexpr(int n) {
                return n;
            }
constexpr int ce      = always_constexpr(100);
          int runtime = 1;
          int not_ce  = always_constexpr(runtime);
```

❌ Compile Error

constinit

Forces a global *variables* to be initialized without dynamic initialization

```
          auto dynamic()         -> std::u8string_view {
                  return "dynamic";
              }
constexpr auto constant(bool b) -> std::u8string_view {
                  return b ? "constant" : dynamic();
              }

constinit std::u8string_view c = constant(true);
constinit std::u8string_view c = constant(false);
```

constinit

Forces a global *variables* to be initialized without dynamic initialization

```
          auto dynamic()         -> std::u8string_view {
                  return "dynamic";
              }
constexpr auto constant(bool b) -> std::u8string_view {
                  return b ? "constant" : dynamic();
              }

constinit std::u8string_view c = constant(true);
constinit std::u8string_view c = constant(false);
```

constinit

Forces a global *variables* to be initialized without dynamic initialization

```
          auto dynamic()         -> std::u8string_view {
                  return "dynamic";
              }
constexpr auto constant(bool b) -> std::u8string_view {
                  return b ? "constant" : dynamic();
              }

constinit std::u8string_view c = constant(true);
constinit std::u8string_view c = constant(false);
```

constinit

Forces a global *variables* to be initialized without dynamic initialization

```
          auto dynamic()         -> std::u8string_view {
                  return "dynamic";
              }
constexpr auto constant(bool b) -> std::u8string_view {
                  return b ? "constant" : dynamic();
              }

constinit std::u8string_view c = constant(true);
constinit std::u8string_view c = constant(false);
```

✅ OK

constinit

Forces a global *variables* to be initialized without dynamic initialization

```
          auto dynamic()         -> std::u8string_view {
                 return "dynamic";
               }
constexpr auto constant(bool b) -> std::u8string_view {
                 return b ? "constant" : dynamic();
               }

constinit std::u8string_view c = constant(true);
constinit std::u8string_view c = constant(false);
```

❌ Compile Error

the great constexr-ification

Lots of `std::` library features can now be used in `constexpr` contexts:

- `std::string_view`, `std::tuple`

the great constexr-ification

Lots of `std::` library features can now be used in `constexpr` contexts:

- `std::string_view`, `std::tuple`
- `std::string`, `std::vector`

the great constexr-ification

Lots of `std::` library features can now be used in `constexpr` contexts:

- `std::string_view`, `std::tuple`
- `std::string`, `std::vector`
- `std::sort`, `std::all_of`

the great constexr-ification

Lots of `std::` library features can now be used in `constexpr` contexts:

- `std::string_view`, `std::tuple`
- `std::string`, `std::vector`
- `std::sort`, `std::all_of`
- ...

class types in non-type template parameters

```
template<notstd::fixed_string Pattern>
struct match {};
```

```cpp
auto extract_number(std::string_view s)
  -> std::optional<std::string_view> noexcept
{
    if (auto m = ctre::match<"[a-z]+([0-9]+)">(s)) {
        return m.get<1>().to_view();
    }

    return std::nullopt;
}
```

News from the initialization frontier

designated initializers

```
struct color {  uint8_t r; uint8_t g; uint8_t b;
                uint8_t a = 0xff; };
color red        = { .r = 0xff };
color purple     = { .r = 0xa0, .b = 0xc0 };
color green_tint = { .g = 0xff, .a = 0x7f };
```

designated initializers

```
struct color {  uint8_t r; uint8_t g; uint8_t b;
                uint8_t a = 0xff; };
color red        = { .r = 0xff };
color purple     = { .r = 0xa0, .b = 0xc0 };
color green_tint = { .g = 0xff, .a = 0x7f };
```

initializers can not be out-of-order, array, nested or mixed.

parenthesized aggregate initialization

```
struct point { int x; int y; };
point p1{ 1.0, 1.0 }; // ❌ Compile Error
point p2( 1.0, 1.0 ); // ✅ narrowing conversion
```

more deduction guides

```
template<typename T> struct point{ T x; T y; };
point pd{1.0, 1.0};      // -> point<double> deduced
point pi{1, 1};          // -> point<int> deduced
std::pmr::vector v = {1, 2, 3};
  // -> vector<int, polymorphic_allocator<int>> deduced
```

Attributes

likely/unlikely

```
void some_fun() {
  auto err = might_fail();
  if (err) [[unlikely]] {
      return;
  }
  more_fun();
}
```

`[[likely]]` and `[[unlikely]]` help to guide the optimizer

nodiscard reasons

```
struct MyContainer {
    [[nodiscard("empty() means is_empty(), not clear()!")]]
    bool empty() const;
};
```

no_unique_address

```
template<typename T, typename Alloc = std::allocator<T>>
struct MyContainer {

  private:
      T* data_
      [[no_unique_address]] Alloc alloc_;
};
```

THANK YOU!

The C++20 Firehose talk

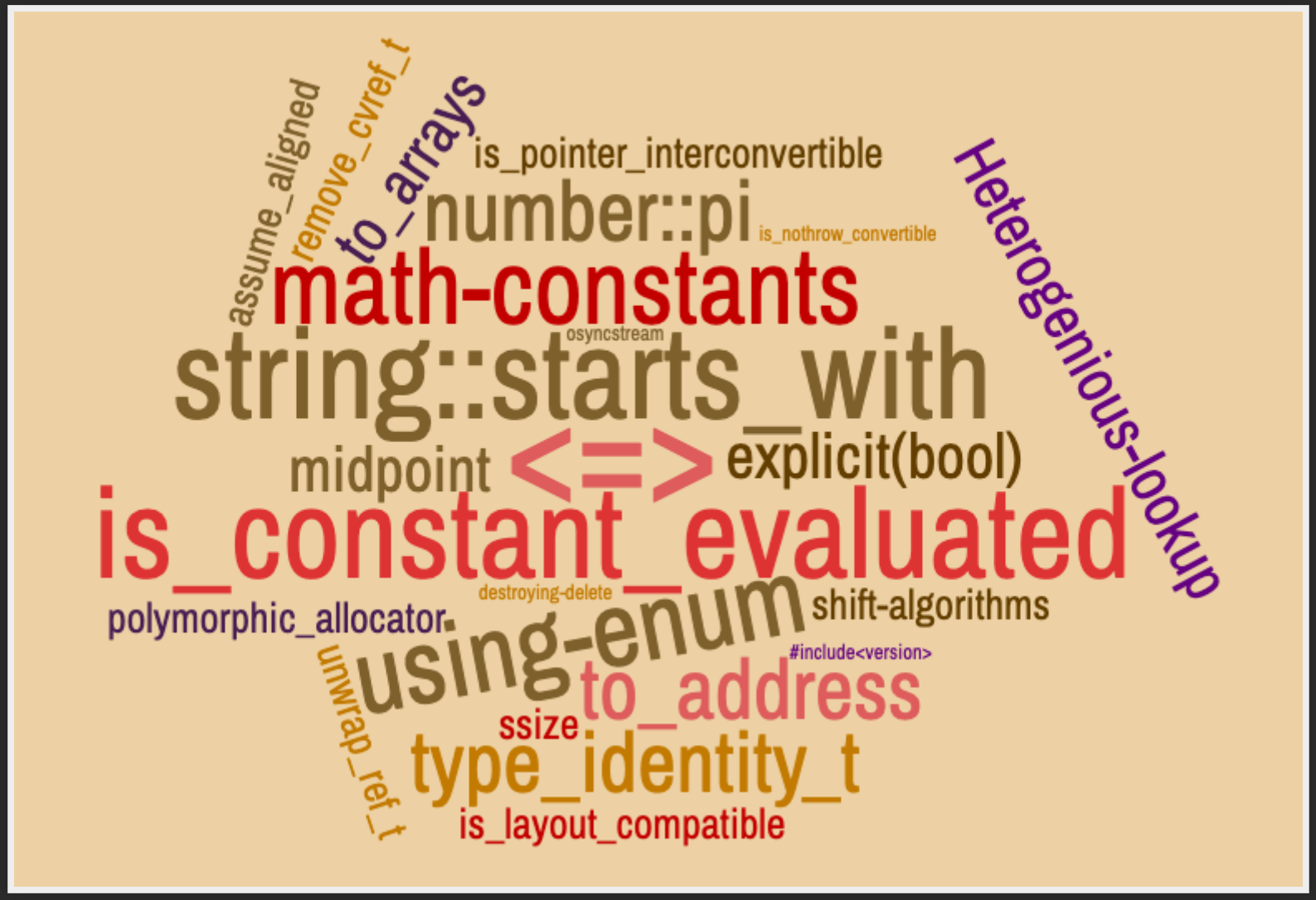
QUESTIONS?

feel free to approach me during breaks!

THANK YOU!

Fabio Fracassi	**fabio@fracassi.de**
https://code.berlin/	fabio.fracassi@code.berlin


C<>DE
UNIVERSITY
OF APPLIED
SCIENCES


assume_aligned
remove_cvref_t
to_arrays
is_pointer_interconvertible
number::pi
is_nothrow_convertible
Heterogenious-lookup
math-constants
osyncstream
string::starts_with
midpoint
<=>
explicit(bool)
is_constant_evaluated
destroying-delete
polymorphic_allocator
using-enum
shift-algorithms
unwrap_ref_t
#include<version>
to_address
ssize
type_identity_t
is_layout_compatible