


# C++20:

# The small things

Version 1.4

**Timur Doumler**

 @timur\_audio

CoreHard Autumn Conference  
30 November 2019



## IS schedule

The following is the current schedule for the C++ IS, approved by WG21 unanimous consent in Jacksonville (2018-03).

2017.2 – Toronto	First meeting of C++20
2017.3 – Albuquerque	<i>Try to front-load “big” language features including ones with broad library impact</i>
2018.1 – Jacksonville	
2018.2 – Rapperswil	
2018.3 – San Diego	<i>(incl. try to merge TSes here)</i>
	<i>EWG: Last meeting for new C++20 language proposals we haven’t seen before</i>
	<i>EWG → LEWG: Last meeting to approve C++20 features needing library response</i> <i>LEWG: Focus on progressing papers on how to react to new language features</i>
2019.1 – Kona	<i>* → CWG,LWG: Last meeting to send proposals to wording review (incl. TS merges)</i> C++20 design is feature-complete
2019.2 – Cologne	CWG+LWG: Complete CD wording EWG+LEWG: Working on C++23 features + CWG/LWG design clarification questions C++20 draft wording is feature complete, <b>start CD ballot</b>
2019.3 – Belfast	CD ballot comment resolution
2020.1 – Prague	CD ballot comment resolution C++20 technically finalized, <b>start DIS ballot</b>

## IS schedule

The following is the current schedule for the C++ IS, approved by WG21 unanimous consent in Jacksonville (2018-03).

2017.2 – Toronto	First meeting of C++20
2017.3 – Albuquerque	<i>Try to front-load “big” language features including ones with broad library impact</i>
2018.1 – Jacksonville	
2018.2 – Rapperswil	
2018.3 – San Diego	<i>(incl. try to merge TSes here)</i>
	<i>EWG: Last meeting for new C++20 language proposals we haven’t seen before</i>
	<i>EWG → LEWG: Last meeting to approve C++20 features needing library response</i> <i>LEWG: Focus on progressing papers on how to react to new language features</i>
2019.1 – Kona	<i>* → CWG,LWG: Last meeting to send proposals to wording review (incl. TS merges)</i> C++20 design is feature-complete
2019.2 – Cologne	CWG+LWG: Complete CD wording EWG+LEWG: Working on C++23 features + CWG/LWG design clarification questions C++20 draft wording is feature complete, <b>start CD ballot</b>
2019.3 – Belfast	CD ballot comment resolution
2020.1 – Prague	CD ballot comment resolution C++20 technically finalized, <b>start DIS ballot</b>



## IS schedule

The following is the current schedule for the C++ IS, approved by WG21 unanimous consent in Jacksonville (2018-03).

2017.2 – Toronto	First meeting of C++20
2017.3 – Albuquerque	<i>Try to front-load “big” language features including ones with broad library impact</i>
2018.1 – Jacksonville	
2018.2 – Rapperswil	
2018.3 – San Diego	<i>(incl. try to merge TSes here)</i>
	<i>EWG: Last meeting for new C++20 language proposals we haven’t seen before</i>
	<i>EWG → LEWG: Last meeting to approve C++20 features needing library response</i> <i>LEWG: Focus on progressing papers on how to react to new language features</i>
2019.1 – Kona	<i>* → CWG,LWG: Last meeting to send proposals to wording review (incl. TS merges)</i> C++20 design is feature-complete
2019.2 – Cologne	CWG+LWG: Complete CD wording EWG+LEWG: Working on C++23 features + CWG/LWG design clarification questions C++20 draft wording is feature complete, <b>start CD ballot</b>
2019.3 – Belfast	CD ballot comment resolution
2020.1 – Prague	CD ballot comment resolution C++20 technically finalized, <b>start DIS ballot</b>

## IS schedule

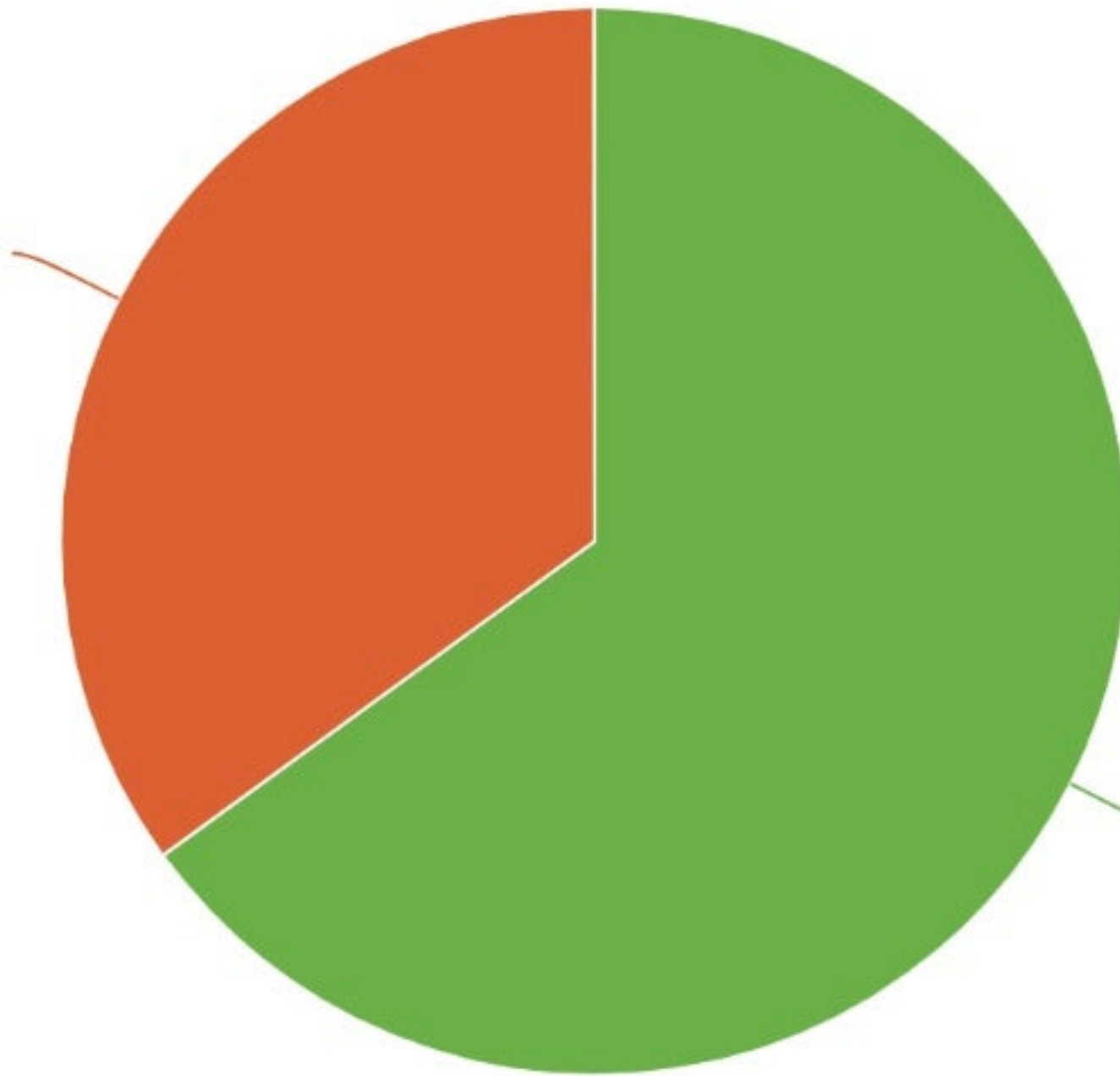
The following is the current schedule for the C++ IS, approved by WG21 unanimous consent in Jacksonville (2018-03).

2017.2 – Toronto	First meeting of C++20
2017.3 – Albuquerque	<i>Try to front-load “big” language features including ones with broad library impact</i>
2018.1 – Jacksonville	
2018.2 – Rapperswil	
2018.3 – San Diego	<i>(incl. try to merge TSes here)</i>
	<i>EWG: Last meeting for new C++20 language proposals we haven’t seen before</i>
	<i>EWG → LEWG: Last meeting to approve C++20 features needing library response</i> <i>LEWG: Focus on progressing papers on how to react to new language features</i>
2019.1 – Kona	<i>* → CWG,LWG: Last meeting to send proposals to wording review (incl. TS merges)</i> C++20 design is feature-complete
2019.2 – Cologne	CWG+LWG: Complete CD wording EWG+LEWG: Working on C++23 features + CWG/LWG design clarification questions C++20 draft wording is feature complete, <b>start CD ballot</b>
2019.3 – Belfast	CD ballot comment resolution
2020.1 – Prague	CD ballot comment resolution C++20 technically finalized, <b>start DIS ballot</b>



**we are here**

**Core Language**

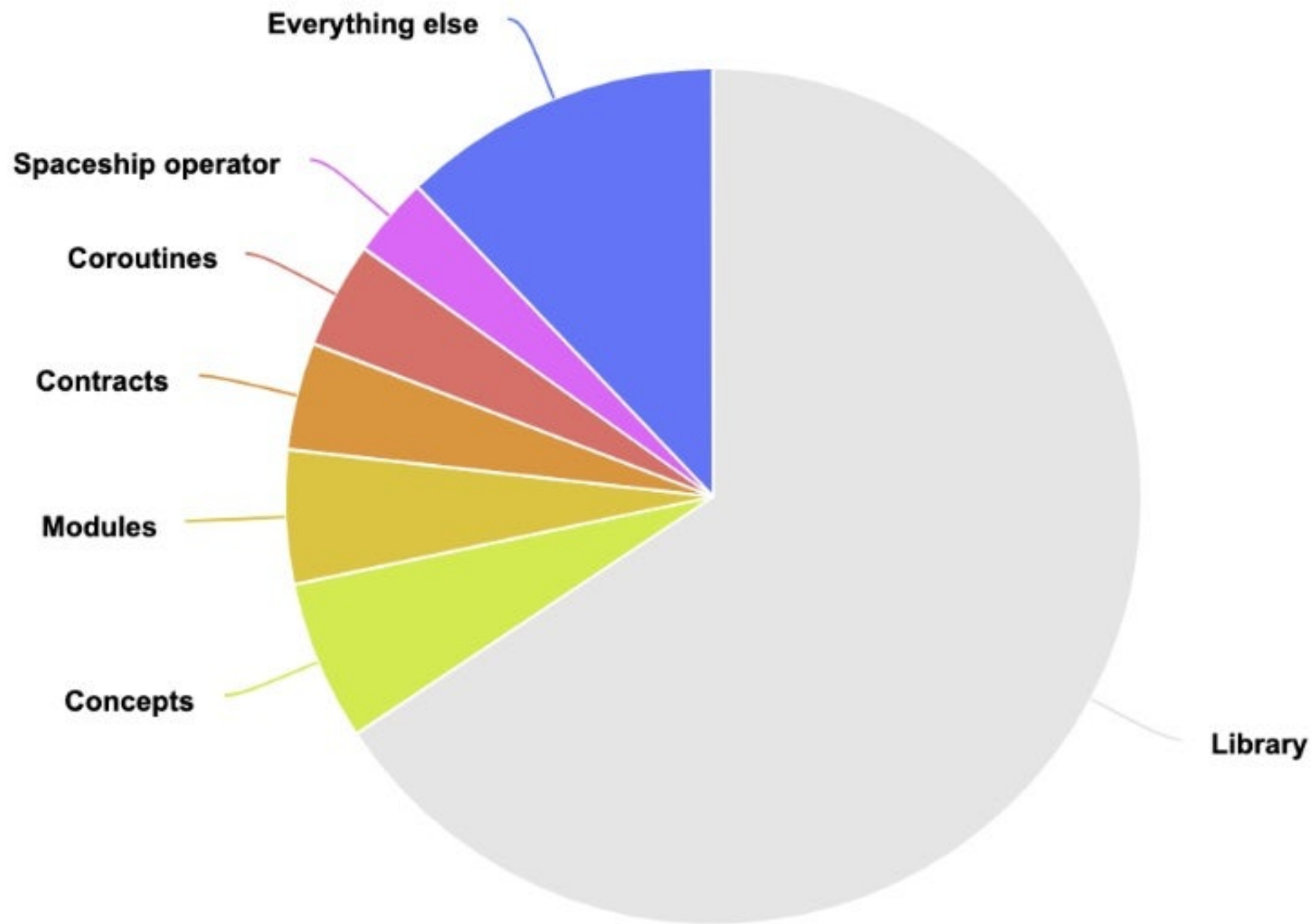


**Library**

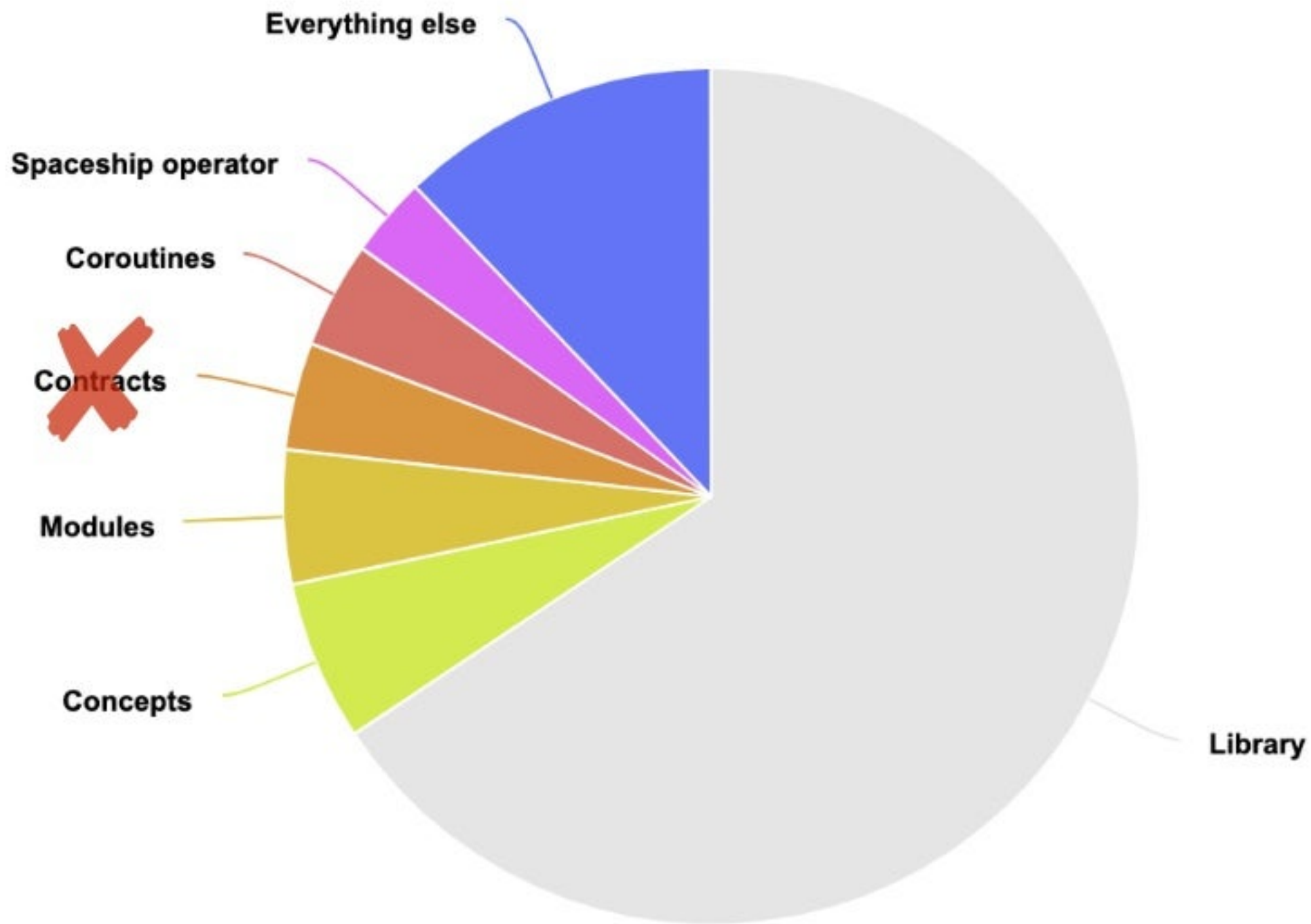
**Core Language**

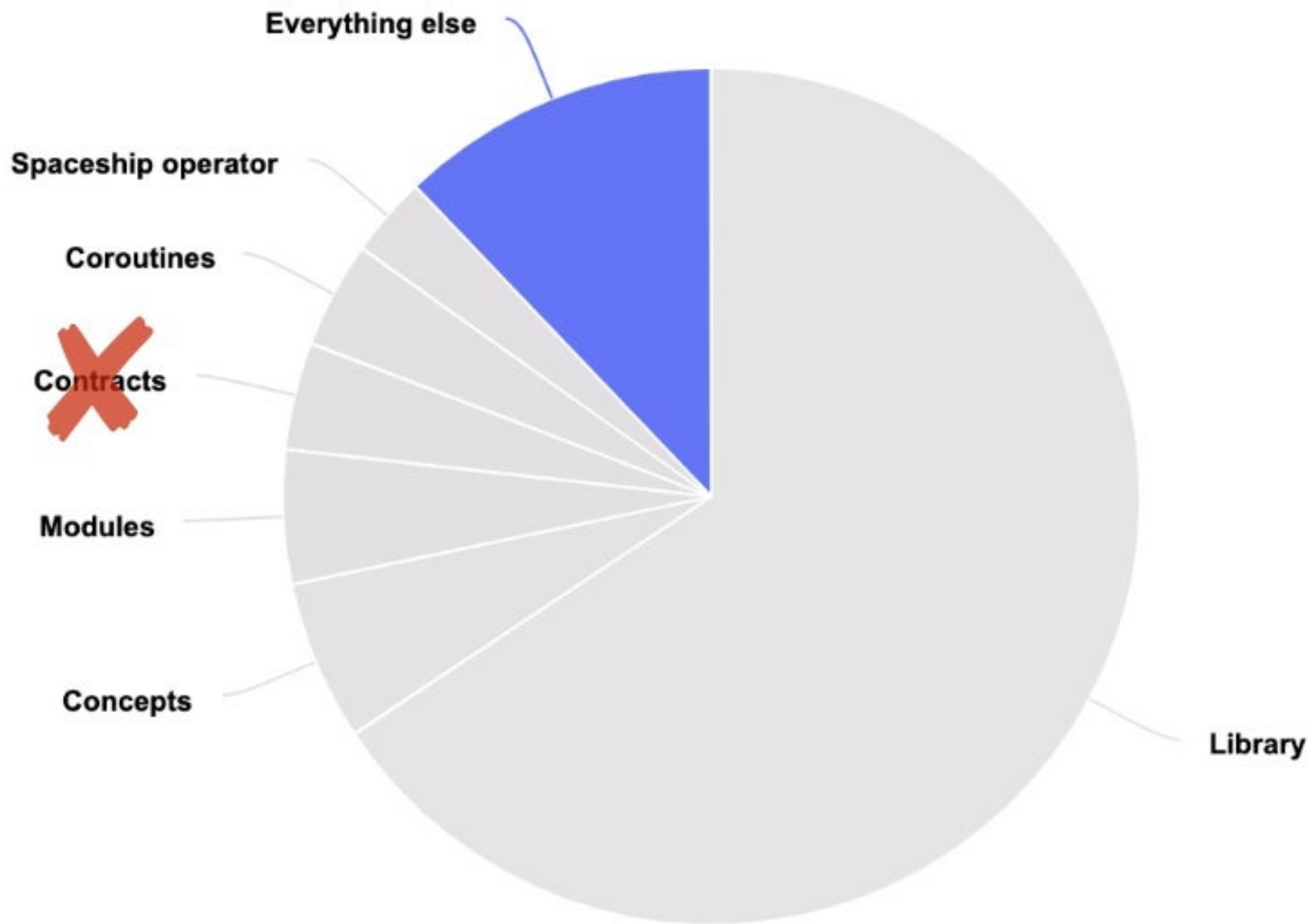


**Library**









- 1 Initialisation
- 2 Structured bindings
- 3 Lambdas
- 4 Templates
- 5 constexpr
- 6 Miscellaneous



- 1 Initialisation
- 2 Structured bindings
- 3 Lambdas
- 4 Templates
- 5 constexpr
- 6 Miscellaneous

# Aggregates

```
struct Widget {  
    int a;  
    bool b;  
    int c;  
};
```

# Aggregates

```
struct Widget {  
    int a;  
    bool b;  
    int c;  
};
```

```
int main() {  
    Widget widget = {3, true};  
}
```



# Designated initialisers

```
struct Widget {  
    int a;  
    bool b;  
    int c;  
};  
  
int main() {  
    Widget widget{.a = 3, .c = 7};  
}
```

# Designated initialisers

```
struct Widget {  
    int a;  
    bool b;  
    int c;  
};  
  
int main() {  
    Widget widget{.a = 3, .c = 7};  
}
```

Only for aggregate types.

C compatibility feature.

Works like in C99, except:

- not out-of-order  
Widget widget{.c = 7, .a = 3} // Error
- not nested  
Widget widget{.c.e = 7} // Error
- not mixed with regular initialisers  
Widget widget{.a = 3, 7} // Error
- not with arrays  
int arr[3]{.[1] = 7} // Error

An *aggregate* is an array or a class (Clause 12) with

- no ~~user-provided, explicit,~~user-declared or inherited constructors (15.1),
- no private or protected non-static data members (Clause 14),
- no virtual functions (13.3), and
- no virtual, private, or protected base classes (13.1).



Aggregates can no longer declare constructors

# Aggregates can no longer declare constructors

```
struct Widget {  
    Widget() = delete;  
};  
  
Widget w1;    // Error
```

# Aggregates can no longer declare constructors

```
struct Widget {  
    Widget() = delete;  
};
```

```
Widget w1;    // Error  
Widget w2{};  // OK in C++17!
```



# Aggregates can no longer declare constructors

```
struct Widget {  
    Widget() = delete;  
};
```

```
Widget w1;    // Error
```

```
Widget w2{};  // OK in C++17! Will be error in C++20
```

# C++17 problems with aggregate initialisation:

- Does not work with macros:

```
assert(Widget(2, 3));    // OK
```

# C++17 problems with aggregate initialisation:

- Does not work with macros:

```
assert(Widget(2, 3));    // OK  
assert(Widget{2, 3});    // Error: this breaks the preprocessor :(
```

# C++17 problems with aggregate initialisation:

- Does not work with macros:

```
assert(Widget(2, 3));    // OK  
assert(Widget{2, 3});    // Error: this breaks the preprocessor :(
```

- Can't do perfect forwarding in templates
  - can't write `emplace` or `make_unique` that works for aggregates :(

# C++20: Direct-initialisation of aggregates

```
struct Widget {  
    int i;  
    int j;  
};
```

```
Widget widget(1, 2);    // will work in C++20!
```



# C++20: Direct-initialisation of aggregates

```
struct Widget {  
    int i;  
    int j;  
};
```

```
Widget widget(1, 2);    // will work in C++20!  
int arr[3](0, 1, 2);    // will work in C++20!
```

# C++20: Direct-initialisation of aggregates

```
struct Widget {  
    int i;  
    int j;  
};
```

```
Widget widget(1, 2);    // will work in C++20!  
int arr[3](0, 1, 2);    // will work in C++20!
```

So in C++20, (args) and {args} will do the same thing!

Except:

- ( ) does not call std::initializer\_list constructors
- {} does not allow narrowing conversions

# constinit

```
struct Colour  
{  
    Colour(int r, int g, int b) noexcept;  
};
```

# constinit

```
struct Colour
{
    Colour(int r, int g, int b) noexcept;
};
```

```
namespace Colours
{
    const Colour red = {255, 0, 0};
}
```

# constinit

```
struct Colour
{
    Colour(int r, int g, int b) noexcept;
};

namespace Colours
{
    const Colour red = {255, 0, 0}; // dynamic initialisation
}
```



# constinit

```
struct Colour
{
    Colour(int r, int g, int b) noexcept;
};

namespace Colours
{
    const Colour red = {255, 0, 0}; // dynamic initialisation
                                    // -> initialisation order fiasco -> UB :(
}
```

# constinit

```
struct Colour  
{  
    constexpr Colour(int r, int g, int b) noexcept;  
};
```

```
namespace Colours  
{  
    constexpr Colour red = {255, 0, 0};  
}
```

# constinit

```
struct Colour
{
    constexpr Colour(int r, int g, int b) noexcept;
};

namespace Colours
{
    constexpr Colour red = {255, 0, 0}; // constant initialisation :)
}
```

# constinit

```
struct Colour
{
    constexpr Colour(int r, int g, int b) noexcept;
};

namespace Colours
{
    Colour backgroundColour = getBackgroundColour();
}
```

# constinit

```
struct Colour
{
    constexpr Colour(int r, int g, int b) noexcept;
};

namespace Colours
{
    Colour backgroundColour = getBackgroundColour(); // const or dynamic init?
}
```



# constexpr

```
struct Colour
{
    constexpr Colour(int r, int g, int b) noexcept;
};

namespace Colours
{
    constexpr Colour backgroundColour = getBackgroundColour();
}
```

# constexpr

```
struct Colour
{
    constexpr Colour(int r, int g, int b) noexcept;
};

namespace Colours
{
    constexpr Colour backgroundColour = getBackgroundColour();
} // ^^^^^^ only compiles if init happens at compile time :)
```

# Range-based for with initialiser

## C++17

```
Database getDatabase();
```

```
for (auto&& user : getDatabase().getUsers())  
{  
    registerUser(user);  
}
```

# Range-based for with initialiser

## C++17

```
Database getDatabase();
```

```
for (auto&& user : getDatabase().getUsers()) // maybe undefined behaviour!  
{  
    registerUser(user);  
}
```

# Range-based for with initialiser

## C++17

```
Database getDatabase();
```

```
auto db = getDatabase();  
for (auto&& user : db.getUsers())  
{  
    registerUser(user);  
}
```

# Range-based for with initialiser

## C++17

```
Database getDatabase();
```

```
{  
    auto db = getDatabase();  
    for (auto&& user : db.getUsers())  
    {  
        registerUser(user);  
    }  
}
```



# Range-based for with initialiser

## C++20

```
Database getDatabase();
```

```
for (auto db = getDatabase(); auto&& user : db.getUsers())  
{  
    registerUser(user);  
}
```

- 1 Initialisation
- 2 Structured bindings**
- 3 Lambdas
- 4 Templates
- 5 constexpr
- 6 Miscellaneous

```
struct Widget  
{  
    int i;  
    bool b;  
};
```

```
auto [a, b] = getWidget();
```

```
struct Widget
{
    int i;
    bool b;
};
```

```
auto [a, b] = getWidget();
static [a, b] = getWidget(); // Error in C++17
thread_local [a, b] = getWidget(); // Error in C++17
```

```
struct Widget
{
    int i;
    bool b;
};
```

```
auto [a, b] = getWidget();
static [a, b] = getWidget(); // OK in C++20
thread_local [a, b] = getWidget(); // OK in C++20
```

```
struct Widget
{
    int i;
    bool b;
};
```

```
auto [a, b] = getWidget();
```

```
auto f = [a]{ return a > 0; }; // Error in C++17:
                               // capture 'a' does not name a variable
```

```
struct Widget  
{  
    int i;  
    bool b;  
};
```

```
auto [a, b] = getWidget();
```

```
auto f = [a]{ return a > 0; }; // OK in C++20
```



```
struct Widget
{
    int i;
    bool b;
};
```

```
auto [a, b] = getWidget();
```

```
auto f = [a]{ return a > 0; }; // OK in C++20  
                                // copies 'a', not the whole object
```

- 1 Initialisation
- 2 Structured bindings
- 3 Lambdas**
- 4 Templates
- 5 constexpr
- 6 Miscellaneous

## C++20: pack expansion allowed in lambda init capture

```
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...);
    };
}
```

## More C++20 lambda features:

- Lambdas are allowed in unevaluated contexts
- Lambdas (without captures) are default-constructible and assignable

## More C++20 lambda features:

- Lambdas are allowed in unevaluated contexts
- Lambdas (without captures) are default-constructible and assignable

```
decltype( [] {} )
```

## More C++20 lambda features:

- Lambdas are allowed in unevaluated contexts
- Lambdas (without captures) are default-constructible and assignable

```
decltype([]{} ) f;
```

## More C++20 lambda features:

- Lambdas are allowed in unevaluated contexts
- Lambdas (without captures) are default-constructible and assignable

```
class Widget
{
    decltype([]{}) f;
};
```



```
template <typename T>  
using MyPtr = std::unique_ptr<  
    T, decltype([](T* t) { myDeleter(t); })>;
```

```
MyPtr<Widget> ptr;
```

```
template <typename T>  
using MyPtr = std::unique_ptr<  
    T, decltype([](T* t) { myDeleter(t); })>;
```

```
MyPtr<Widget> ptr;
```

```
using WidgetSet = std::set<  
    Widget,  
    decltype([](Widget& lhs, Widget& rhs) { return lhs.x < rhs.x; })>;
```

```
WidgetSet widgets;
```

# Generic lambdas / functions

```
auto f = [](auto a){  
    return a * a;  
};
```

# Generic lambdas / functions

```
auto f = [](auto a){  
    return a * a;  
};
```

```
auto f(auto a) {    // Generic *functions* – OK since C++20 :)  
    return a * a;  
}
```

# Generic lambdas / functions

```
template <typename T>  
void f(std::vector<T> vector) {  
    // ...  
}
```

# Generic lambdas / functions

```
template <typename T>  
void f(std::vector<T> vector) {  
    // ...  
}
```

```
// C++20:  
auto f = []<typename T>(std::vector<T> vector) {  
    // ...  
};
```

- 1 Initialisation
- 2 Structured bindings
- 3 Lambdas
- 4 Templates**
- 5 constexpr
- 6 Miscellaneous



# Non-type template parameters (NTTPs)

# Non-type template parameters (NTTPs)

```
template <int size>
struct Widget
{
    std::array<int, size> a;
};
```

# Non-type template parameters (NTTPs)

```
template <int size>
struct Widget
{
    std::array<int, size> a;
};
```

## C++20: floating-point NTTPs

```
template <double x>
struct Filter
{
    std::array<double, 2> coefficients = {x, 0.5 * x * x};

    // stuff...
};
```

## C++20: class-type NTTPs

```
struct Coefficients  
{  
    double x;  
    double y;  
};
```

## C++20: class-type NTTPs

```
struct Coefficients
{
    double x;
    double y;
};

template <Coefficients coeffs>
struct Filter
{
    // stuff :)
};
```

## C++20: class-type NTTPs

```
struct Coefficients
{
    double x;
    double y;
};

template <Coefficients coeffs>
struct Filter
{
    // stuff :)
};

constexpr Filter<Coefficients{1, 0.125}> f;
```



CTAD

# CTAD

```
std::vector v = {1, 2, 3};           // std::vector<int>
```

# CTAD

```
std::vector v = {1, 2, 3};           // std::vector<int>
```

```
std::tuple t = {42, 0.5, true};      // std::tuple<int, double, bool>
```

# CTAD

```
std::vector v = {1, 2, 3};           // std::vector<int>

std::tuple t = {42, 0.5, true};      // std::tuple<int, double, bool>

std::scoped_lock lock(rtmutex);      // std::scoped_lock<std::recursive_timed_mutex>
```

## C++20 adds:

- CTAD for aggregates
- CTAD for alias templates

# C++17

```
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};
```

```
aggr_pair p = {1, true}; // Error: no deduction candidate found
```

# C++17

```
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};
```

```
template <typename T, typename U>
aggr_pair(T, U) -> aggr_pair<T, U>;
```

```
aggr_pair p = {1, true}; // OK
```

## C++17

```
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};

template <typename T, typename U>
aggr_pair(T, U) -> aggr_pair<T, U>;

aggr_pair p = {1, true}; // OK
```

## C++20

```
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};

aggr_pair p = {1, true}; // OK
```



```
template<typename... Bases>  
struct overloaded : Bases...  
{  
    using Bases::operator()...;  
};
```

```
template<typename... Bases>  
struct overloaded : Bases...  
{  
    using Bases::operator()...;  
};
```

```
template<typename... Bases>  
overloaded(Bases...) -> overloaded<Bases...>;
```

```
template<typename... Bases>
struct overloaded : Bases...
{
    using Bases::operator()...;
};
```

```
template<typename... Bases>
overloaded(Bases...) -> overloaded<Bases...>;
```

```
overloaded printer = {
    [](auto arg) { std::cout << arg << ' '; },
    [](double arg) { std::cout << std::fixed << arg << ' '; },
    [](const char* arg) { std::cout << std::quoted(arg) << ' '; }
};
```

```
template<typename... Bases>
struct overloaded : Bases...
{
    using Bases::operator()...;
};
```

```
template<typename... Bases>
overloaded(Bases...) -> overloaded<Bases...>;
```

```
overloaded printer = {
    [](auto arg) { std::cout << arg << ' '; },
    [](double arg) { std::cout << std::fixed << arg << ' '; },
    [](const char* arg) { std::cout << std::quoted(arg) << ' '; }
};
```

```
int main()
{
    printer("Hello, World!");
}
```

```
template<typename... Bases>
struct overloaded : Bases...
{
    using Bases::operator()...;
};
```

```
template<typename... Bases>
overloaded(Bases...) -> overloaded<Bases...>;
```

```
overloaded printer = {
    [](auto arg) { std::cout << arg << ' '; },
    [](double arg) { std::cout << std::fixed << arg << ' '; },
    [](const char* arg) { std::cout << std::quoted(arg) << ' '; }
};
```

```
int main()
{
    printer("Hello, World!");
}
```

```
template<typename... Bases>
struct overloaded : Bases...
{
    using Bases::operator()...;
};
```

```
overloaded printer = {
    [](auto arg) { std::cout << arg << ' '; },
    [](double arg) { std::cout << std::fixed << arg << ' '; },
    [](const char* arg) { std::cout << std::quoted(arg) << ' '; }
};
```

```
int main()
{
    printer("Hello, World!");
}
```



```
namespace pmr {  
    template <class T>  
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;  
}
```

## C++17

```
std::pmr::vector<int> v{1, 2, 3};
```

```
namespace pmr {  
    template <class T>  
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;  
}
```

## C++17

```
std::pmr::vector<int> v{1, 2, 3};
```

## C++20

```
std::pmr::vector v{1, 2, 3};
```



- 1 Initialisation
- 2 Structured bindings
- 3 Lambdas
- 4 Templates
- 5 constexpr**
- 6 Miscellaneous

In C++20, in a constexpr function you can:

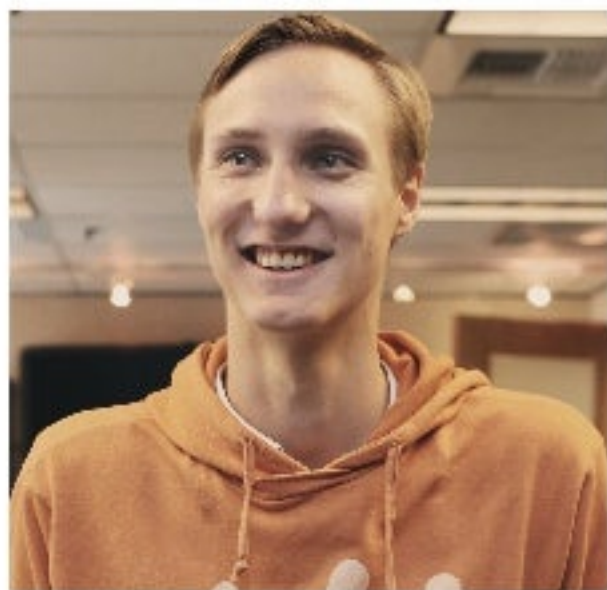
- have a try-block
- have an unevaluated asm block
- use a union
- call virtual functions
- dynamic\_cast and typeid
- new and delete



*Daveed Vandevoorde*

**“C++ Constants”**

C++Now 2019 keynote



*Louis Dionne*

**"Compile-time programming and  
reflection in C++20 and beyond"**

CppCon 2018 talk

“running” code at compile time

```
int square(int i) {  
    return i * i;  
}
```

## “running” code at compile time

```
constexpr int square(int i) {  
    return i * i;  
}
```

```
square(3); // compile time  
square(x); // runtime
```

## “running” code at compile time

```
constexpr int square(int i) {  
    return i * i;  
}
```

```
square(3); // compile time  
square(x); // Error - x is not a compile-time constant!
```

## compile time or runtime?

```
int square(int i) {  
    return __magic_fast_square(i); // contains runtime magic  
}
```

```
square(3); // runtime, fast magic  
square(x); // runtime, fast magic
```

# compile time or runtime?

```
constexpr int square(int i) {  
    return i * i;  
}
```

```
square(3); // compile time  
square(x); // runtime, no fast magic :(
```



# compile time or runtime?

```
constexpr int square(int i) {  
    if (std::is_constant_evaluated()) {  
        return i * i;  
    }  
    else {  
        return __magic_fast_square(i);  
    }  
}
```

```
square(3); // compile time  
square(x); // runtime, fast magic :)
```

# compile time or runtime?

```
constexpr int square(int i) {  
    if (std::is_constant_evaluated()) {  
        return i * i;  
    }  
    else {  
        return __magic_fast_square(i);  
    }  
}
```

```
square(3); // compile time  
square(x); // runtime, fast magic :)
```

- 1 Initialisation
- 2 Structured bindings
- 3 Lambdas
- 4 Templates
- 5 constexpr
- 6 Miscellaneous**

```
template <typename Container>
auto findFirstValid(const Container& c) -> Container::iterator
{
    return std::find_if(c.begin(), c.end(), [](auto elem){ return elem.is_valid(); });
}
```

```
template <typename Container>  
auto findFirstValid(const Container& c) -> Container::const_iterator  
{  
    return std::find_if(c.begin(), c.end(), [](auto elem){ return elem.is_valid(); });  
}
```

*// Error: missing 'typename' prior to dependent type name 'Container::const\_iterator'*

```
template <typename Container>
auto findFirstValid(const Container& c) -> Container::const_iterator
{
    return std::find_if(c.begin(), c.end(), [](auto elem){ return elem.is_valid(); });
}

// OK in C++20 :)
```

# New attributes in C++20

## New attributes in C++20

- `[[likely]]`, `[[unlikely]]`



## New attributes in C++20

- `[[likely]]`, `[[unlikely]]`
- `[[no_unique_address]]`

## New attributes in C++20

- `[[likely]]`, `[[unlikely]]`
- `[[no_unique_address]]`
- `[[nodiscard]]` on constructors

## New attributes in C++20

- `[[likely]]`, `[[unlikely]]`
- `[[no_unique_address]]`
- `[[nodiscard]]` on constructors
- `[[nodiscard("can have a message")]]`

# Using enum

```
enum class rgba_color_channel {  
    red,  
    green,  
    blue,  
    alpha  
};
```

# Using enum

```
enum class rgba_color_channel {  
    red,  
    green,  
    blue,  
    alpha  
};  
  
std::string_view to_string(rgba_color_channel channel) {  
    switch (channel) {  
        case rgba_color_channel::red:    return "red";  
        case rgba_color_channel::green:  return "green";  
        case rgba_color_channel::blue:   return "blue";  
        case rgba_color_channel::alpha:  return "alpha";  
    }  
}
```

# Using enum

```
enum class rgba_color_channel {  
    red,  
    green,  
    blue,  
    alpha  
};  
  
std::string_view to_string(rgba_color_channel channel) {  
    switch (channel) {  
        case rgba_color_channel::red:    return "red";  
        case rgba_color_channel::green:  return "green";  
        case rgba_color_channel::blue:   return "blue";  
        case rgba_color_channel::alpha:  return "alpha";  
    }  
}
```

# Using enum

```
enum class rgba_color_channel {  
    red,  
    green,  
    blue,  
    alpha  
};  
  
std::string_view to_string(rgba_color_channel channel) {  
    switch (channel) {  
        using enum rgba_color_channel;  
  
        case red:    return "red";  
        case green:  return "green";  
        case blue:   return "blue";  
        case alpha:  return "alpha";  
    }  
}
```

# Using enum

```
enum class Suit {  
    diamonds,  
    hearts,  
    spades,  
    clubs  
};  
  
class Card {  
    using enum Suit;  
    Suit suit = spades;  
};
```



## Built-in UTF-8 char type

```
int main() {  
    const char* str = u8"🦄🌈"; // C++17...  
}
```

# Built-in UTF-8 char type

```
int main() {  
    const char* str = u8"🦄🌈"; // compile error in C++20!  
}
```

## Built-in UTF-8 char type

```
int main() {  
    const char8_t* str = u8"🦄🌈";  
}
```

## IS schedule

The following is the current schedule for the C++ IS, approved by WG21 unanimous consent in Jacksonville (2018-03).

2017.2 – Toronto	First meeting of C++20
2017.3 – Albuquerque	<i>Try to front-load “big” language features including ones with broad library impact</i>
2018.1 – Jacksonville	
2018.2 – Rapperswil	
2018.3 – San Diego	<i>(incl. try to merge TSes here)</i>
	<i>EWG: Last meeting for new C++20 language proposals we haven’t seen before</i>
	<i>EWG → LEWG: Last meeting to approve C++20 features needing library response</i> <i>LEWG: Focus on progressing papers on how to react to new language features</i>
2019.1 – Kona	<i>* → CWG,LWG: Last meeting to send proposals to wording review (incl. TS merges)</i> C++20 design is feature-complete
2019.2 – Cologne	CWG+LWG: Complete CD wording EWG+LEWG: Working on C++23 features + CWG/LWG design clarification questions C++20 draft wording is feature complete, <b>start CD ballot</b>
2019.3 – Belfast	CD ballot comment resolution
2020.1 – Prague	CD ballot comment resolution C++20 technically finalized, <b>start DIS ballot</b>



**we are here**




# C++20:

# The small things

Version 1.4

**Timur Doumler**

 @timur\_audio

CoreHard Autumn Conference  
30 November 2019