

Meeting C++ 2017

Concepts Driven Design

Kris Jusiak, Quantlab Financial

Overview

- | | |
|--|---|
| <ul style="list-style-type: none">• Concepts<ul style="list-style-type: none">◦ Motivation / History• Type constraints (C++20)<ul style="list-style-type: none">◦ Requirements<ul style="list-style-type: none">▪ Design by introspection◦ Named concepts<ul style="list-style-type: none">▪ Optional interfaces• Concepts emulation (C++17) | <ul style="list-style-type: none">• Concepts based design<ul style="list-style-type: none">◦ Static polymorphism◦ Dynamic polymorphism<ul style="list-style-type: none">▪ Virtual concepts (C++2?)◦ Dependency Injection◦ Mocking (testing)• Future of concepts (C++2X) |
|--|---|

Disclaimer

This talk is from user perspective

See Andrew Sutton talks for more details about concepts!

Concepts, although, merged into C++20 draft, are still a subject for future changes

Compiler	Version	Notes
GCC	6.1+	Requires <code>-fconcepts</code> flag
MSVC	VS2017 15.5	-
Clang	In progress...	-

Examples in this talk were compiled using

```
g++7.2 -std=c++2a -fconcepts
```

Motivation - Well specified interfaces

```
template<class T>
const T& min(const T& a, const T& b) {
    return b < a ? b : a;
}
```

Precise documentation

- What's are the syntax requirements for `T` ?
 - compile-time / **concepts**
- What's are the semantics requirements for `min` ?
 - run-time / contracts, tests, manuals

Motivation - Compiler diagnostics

```
int main() {  
    auto list = std::list{1, 2, 3};  
    std::sort(std::begin(list), std::end(list));  
}
```

<https://godbolt.org/g/RTRgg2>

Motivation - Compiler diagnostics

Without concepts

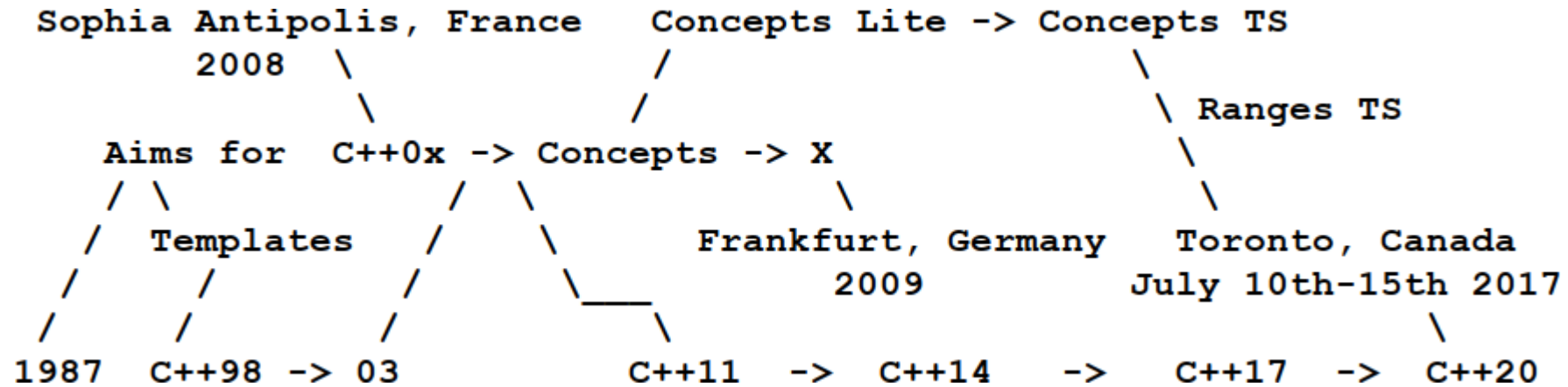
```
invalid operands to binary expression  
  ('std::_List_iterator<int>' and  
   'std::_List_iterator<int>')  
std::__lg(____last - ____first) * 2);  
           ^  
...many lines of templates instantiation call stack...
```

With concepts

```
error: cannot call std::sort  
note: concept RandomAccessIterator was not satisfied  
since: expression (b-a) will be ill formed
```

Note: Constraints are checked at the point of use

History



Requirements analysis	https://wg21.link/n3351
Concepts Lite	https://wg21.link/n3701
Concepts TS	https://wg21.link/P0734R0

Bjarne Stroustrup, Andrew Sutton, Gabriel Dos Reis
Alex Stepanow, Andrew Lumsdaine, Sean Parent, ...

C++20 draft



Requirements

Constraints

Named concepts

<http://eel.is/c++draft/temp.constr>

Requirements

Requires-clause

Specifies constraints on template arguments or on a function declaration

`std::enable_if` on steroids

```
template<class> requires false  
void foo() {}
```

```
foo<class AnyType>(); // Error: constraints not satisfied
```

```
template<bool Value> requires Value  
void bar() {}
```

```
bar<true>(); // Okay  
bar<false>(); // Error: constraints not satisfied
```

Note: requires-clause is part of the function signature

Requires-expression

Expression of type bool

```
requires ( [parameters] ) { requirements }
```

```
requires(T) { // type requirement  
  typename T::value_type; };
```

```
requires(T t) { // simple requirement  
  t[typename T::value_type{}]; };
```

```
requires(T t) { // compound requirement  
  { t.empty() } -> bool; }; // convertible to bool
```

```
requires(T t) { // nested requirement  
  requires std::is_enum_v<typename T::value_type>; };
```

Note: Parameters -> declvals

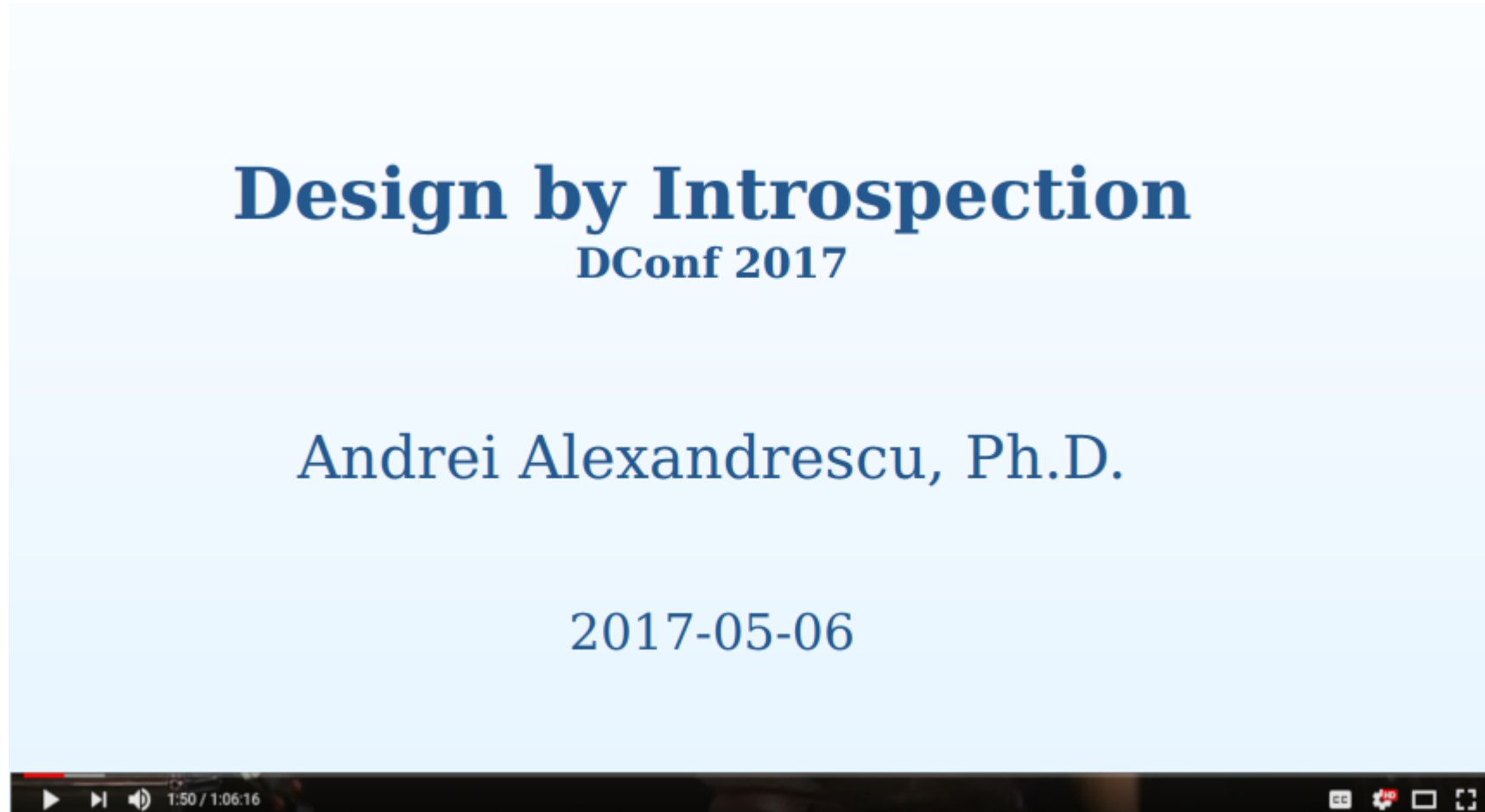
Requires-clause/Requires-expression

```
template<class T>
constexpr auto foo(T&& x) // immediate context (SFINAE friendly)
    requires requires(T t) { t.bar(); }
{ return x.bar(); }
```

```
struct { void bar() {} } bar;
foo(bar); // Okay
foo(42);  // Error: constraints not satisfied
          // note: the required expression
          //       't.bar()' would be ill-formed
```

Note: `requires requires` -> `requires-clause` followed by `requires-expression`

Design by introspection



<https://www.youtube.com/watch?v=29h6jGtZD-U>

Design by introspection - D lang

<https://dlang.org/spec/traits.html>

```
auto foo(T)(T x) {  
    static if(__traits(hasMember, x, "bar")) {  
        return x.bar;  
    } else {  
        return 0;  
    }  
}  
  
void main() {  
    assert(0 == foo(42));  
  
    struct Bar { int bar = 42; }  
    Bar bar;  
    assert(42 == foo(bar));  
}
```

<https://godbolt.org/g/wpLXzV>

Design by introspection - C++20

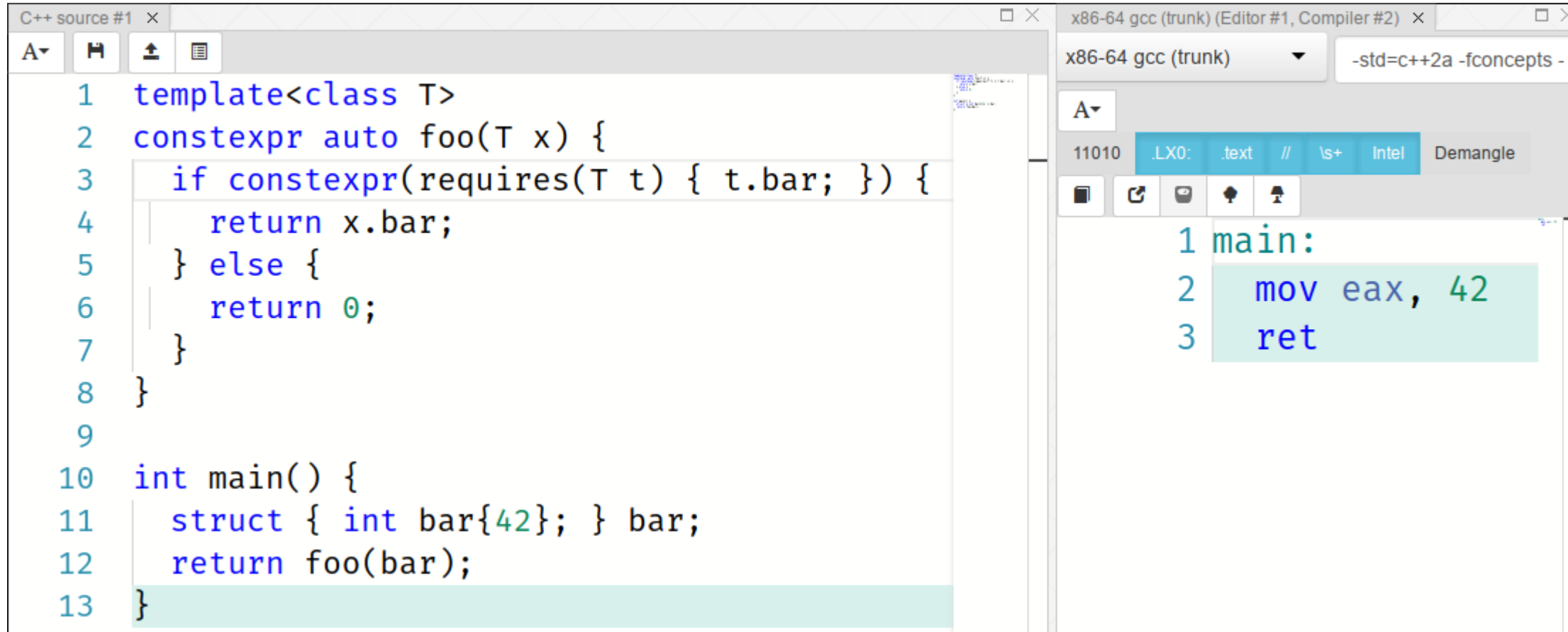
```
template<class T>
constexpr auto foo(T x) {
    if constexpr(requires(T t) { t.bar; }) { // immediate context
        return x.bar;
    } else {
        return 0;
    }
}

int main() {
    assert(0 == foo(42));

    struct { int bar{42}; } bar;
    assert(42 == foo(bar));
}
```

<https://godbolt.org/g/uot6Bv>

Design by introspection - C++



The image shows a screenshot of a C++ IDE with two panes. The left pane displays C++ source code, and the right pane displays the corresponding assembly code generated by the compiler.

C++ source code (left pane):

```
1 template<class T>
2 constexpr auto foo(T x) {
3     if constexpr(requires(T t) { t.bar; }) {
4         return x.bar;
5     } else {
6         return 0;
7     }
8 }
9
10 int main() {
11     struct { int bar{42}; } bar;
12     return foo(bar);
13 }
```

Assembly code (right pane):

The right pane shows the assembly code for the `main` function. The assembly is for x86-64 architecture, using GCC (trunk). The code is as follows:

```
1 main:
2     mov     eax, 42
3     ret
```

<https://godbolt.org/g/MFxqWu>

Requirements

Readability?

```
template<class T>
auto bar(T& t)
    requires requires(T t) {
        typename T::type;
        { t.foo() } -> void;
        requires Movable<T> or Same<T, int>;
    } {
    return foo(t);
}
```

Named concepts

Named concepts

A collection of requirements on a type (variable template)

```
template<template-parameters>  
concept concept-name = constraint-expression;
```

"If you like it then you should have put a name on it", Beyonce rule

Named concepts

Predicate constraints

```
template<class>  
concept Always = true; // always satisfied
```

```
template<class T>  
concept Size32 = sizeof(T) == 4;
```

Conjunctions, Disjunctions

```
template<class T>  
concept SignedIntegral = std::is_integral_v<T> and std::is_signed_v<T>;
```

Named concepts

Requirements

```
template<class T>
concept Fooable =           // named concept
    requires(T t) {         // collection of
        typename T::type;   // type requirement
        { t.foo() } -> void; // compound requirement
        requires Movable<T> or // nested requirement
            Same<T, int>;
    };
```

Note: Concepts are never instantiated
(therefore the `concept` keyword)

Named concepts

Unconstrained class definition

```
template<class> class Bar {};
```

Requires expression (long form)

```
template<class T> requires Fooable<T> class Bar { };
```

Abbreviated templates

```
template<Fooable T> class Bar {};
```

Note: C++17 - Non-type template arguments

```
template<auto T> class Bar {};
```

// For values -> Bar<42>

Named concepts

Example

```
struct tcp_socket { void send(std::string_view); };  
struct udp_socket { void send(std::string_view); };  
struct file       { void write(std::string_view); };
```

```
template<class T> void forward(T& t, std::string_view data) {  
    t.???(data); // send/write  
}
```

Everything Cpp

<https://www.youtube.com/watch?v=xsSYPD0v5Mg>

Named concepts

Concept definition

```
template<class T> concept Socket =  
    requires(T t, std::string_view data) {  
        { t.send(data) } -> void; // compound requirement  
    };
```

```
template<class T> concept File =  
    requires(T t, std::string_view data) {  
        { t.write(data) } -> void; // compound requirement  
    };
```


Named concepts

Concept overloading

```
template<Socket T> // requires Socket<T>  
/*1*/ void forward(T& t, std::string_view data) { t.send(data); }  
  
template<File T> // requires File<T>  
/*2*/ void forward(T& t, std::string_view data) { t.write(data); }
```

```
int main() {  
    tcp_socket tcp; forward(tcp, "tcp data"sv); // calls 1  
    udp_socket udp; forward(udp, "udp data"sv); // calls 1  
    file file; forward(file, "file data"sv); // calls 2  
}
```

Note: If multiple are satisfied the most constrained is chosen

Named concepts

Concepts overloading

`if constexpr` (C++17)

```
template<class T>
void forward(T& t, std::string_view data) {
    if constexpr(Socket<T>) { // compile-time
        t.send(data);
    } else if constexpr(File<T>) {
        t.write(data);
    }
}
```

Note: Branch which is not taken is discarded

----- Statement may not compile but syntax has to be valid

Named concepts

Lambdas

```
constexpr auto forward =  
    [](auto& t, std::string_view data) {  
        using type = std::decay_t<decltype(t)>;  
        if constexpr(Socket<type>) { t.send(data); } else  
        if constexpr(File<type>) { t.write(data); }  
    };
```

Generic lambdas (C++17) - <https://wg21.link/P0428r2>

```
constexpr auto forward =  
    [<class T>(T& t, std::string_view data) {  
        if constexpr(Socket<T>) { t.send(data); } else  
        if constexpr(File<T>) { t.write(data); }  
    };
```

Named concepts

Optional interfaces

Example -> `Stream<T>`

- Copy constructible
- Callable member function `write` which takes type `T&`
- Callable member function `read` which returns type `T`
- Callable **optional** member function `read_complete`
- Printable

Named concepts

Optional interfaces : Inheritance / virtual functions (not expressive enough)

```
/**
 * Implementation requires to be printable and
 * copy constructible
 */
template<class T>
class istream {
public:
    virtual ~istream() noexcept = default;
    virtual void write(T&) = 0;
    virtual T read() = 0;

    // ??? [[optional]] ???
    virtual void read_complete() = 0;
};
```

Named concepts

Optional interfaces : Concepts

```
template<class T, class TData>
concept Streamable =
    CopyConstructible<T> and          // CopyConstructible
    requires(T t, std::ostream& out, TData& data) {
        out << t;                     // Printable
        t.write(data);                // Writable
        { t.read() } -> TData          // Readable
    } or requires(T t, std::ostream& out, TData& data) {
        out << t;                     // Printable
        t.write(data);                // Writable
        { t.read() } -> TData          // Readable 1/2
        t.read_complete();             // Readable 2/2
    }
};
```

Named concepts

Optional interfaces : Usage

```
using data_t = std::array<std::byte, 1024>;
```

```
class FileStream {  
public:  
    void write(data_t&);  
    data_t read();  
    void read_complete();  
};
```

```
int main() {  
    Streamable<data_t> stream = FileStream{};  
    const auto data = stream.read();  
    ...  
    stream.read_complete();  
}
```

Placeholders

Placeholder	Synopsis
Unconstrained	<code>auto</code>
Constrained	<code>concept-name<[template-argument-list]></code>

```
template<class T> class Foo {};  
template<class T> concept Fooable = true;
```

```
// auto - least constrained concept  
auto    foo1 = Foo<int>{};  
// C++17 - Constructor Template Argument Deduction  
Foo     foo2 = Foo<int>{};  
// C++20 - placeholder  
Fooable foo3 = Foo<int>{};
```

Note: Placeholders can be used for functions `void f(auto);`

Concepts in the C++ ISO standard

18	EqualityComparable requirements	467
19	LessThanComparable requirements	467
20	DefaultConstructible requirements	467
21	MoveConstructible requirements	468
22	CopyConstructible requirements (in addition to MoveConstructible)	468
23	MoveAssignable requirements	468
24	CopyAssignable requirements (in addition to MoveAssignable)	468
25	Destructible requirements	468
26	NullablePointer requirements	470
27	Hash requirements	471
28	Descriptive variable definitions	471
29	Allocator requirements	472

Note: More concepts to come with Ranges TS

Concepts and the C++ ISO standard

Table 18 — EqualityComparable requirements [equalitycomparable]

Expression	Return type	Requirement
<code>a == b</code>	convertible to <code>bool</code>	<code>==</code> is an equivalence relation, that is, it has the following properties: <ul style="list-style-type: none">— For all <code>a</code>, <code>a == a</code>.— If <code>a == b</code>, then <code>b == a</code>.— If <code>a == b</code> and <code>b == c</code>, then <code>a == c</code>.

```
template<class T> // C++20 concepts
concept EqualityComparable = requires(T a, T b) {
    { a == b } -> bool;
};
```

```
class EqualityComparable a where // Haskell typeclasses
    (==) :: a -> a -> Bool
```

Concepts - Ranges TS

```
template <class I>
concept InputIterator =
    Iterator<I> &&
    Readable<I> &&
    requires(I& i, const I& ci) {
        typename iterator_category_t<I>;
        DerivedFrom<
            iterator_category_t<I>, input_iterator_tag>;
        i++;
    };
```

<https://github.com/CaseyCarter/cmcstl2>

Concepts emulation (C++17)

Concepts emulation (C++17)

Substitution Failure Is Not An Error (SFINAE)

```
template<bool, class = void>
struct enable_if {}; // no type alias

template<class T>
struct enable_if<true, T> { using type = T; };
```

- `false` predicates lead to ill-formed code that's discarded

Concepts emulation (C++17)

Deduction idiom (C++20)

```
template<class T>  
using Fooable = decltype(std::declval<T&>().foo());
```

```
struct Foo { void foo(); };  
struct Bar { };
```

```
static_assert(not std::is_detected<Fooable, Bar>{});  
static_assert( std::is_detected<Fooable, Foo>{});
```

Note: Under the hood, it uses

```
template<class...> using void_t = void;
```

<https://wg21.link/n4436>

Concepts emulation (C++17)

Does the Concepts TS Improve on C++17?

```
template <class F, class... Args, class = decltype(
    std::declval<F&&>()(std::declval<Args&&>()...))>
constexpr auto requires_impl(int) { return true; }

template <class F, class... Args>
constexpr auto requires_impl(...) { return false; }

template <class... Args, class F>
constexpr auto requires(F&&) {
    return requires_impl<F&&, Args&&...>(int{});
}
```

<https://wg21.link/P0726R0>

Concepts emulation (C++17)

requires-clause

```
struct Foo { void foo(); };  
struct Bar {};
```

```
static_assert(  
    requires<Foo>([](auto&& t) -> decltype(t.foo()) {})  
);  
  
static_assert(  
    !requires<Bar>([](auto&& t) -> decltype(t.foo()) {})  
);
```


Concepts emulation (C++17)

Design by introspection - C++17

```
template<class T> constexpr auto foo(T x) {  
    if constexpr(requires<T>( // compile-time if  
        [](auto&& t) -> decltype(t.bar) {})) {  
        return x.bar;  
    } else {  
        return 0;  
    }  
}
```

Note: Workaround for expression is not a constant expression

```
if constexpr(  
    auto bar = [](auto&& t) -> decltype(t.bar) {};  
    requires<T>(bar)  
)
```

Concepts emulation (C++17)

Named concept

```
template<class T> constexpr auto Socket =  
    requires<T>([](auto&& t, std::string_view data)->  
        decltype(t.send(data)) {}  
    );
```

```
template<class T> constexpr auto File =  
    requires<T>([](auto&& t, std::string_view data) ->  
        decltype(t.write(data)) {}  
    );
```

Concepts emulation (C++17)

Error message

```
template<class T, class = std::enable_if_t<Socket<T>>>
void forward(T& t, std::string_view data) {
    t.send(data);
}
```

```
int main() {
    tcp_socket tcp; forward(tcp, "tcp data"sv); // Okay
    file file; forward(file, "file data"sv);
    // error: no matching function for call to 'forward'
    // possibly many lines of output <- library side
}
```

Note: No details why function couldn't be called

Concepts emulation (C++17)

Concepts overloading

```
template<class T, std::enable_if_t<Socket<T>, int> = 0>
/*1*/ void forward(T& t, std::string_view data) { t.send(data); }

template<class T std::enable_if_t<File<T>, int> = 0>
/*2*/ void forward(T& t, std::string_view data) { t.write(data); }
```

```
int main() {
    tcp_socket tcp; forward(tcp, "tcp data"sv); // calls 1
    udp_socket udp; forward(udp, "udp data"sv); // calls 1
    file file; forward(file, "file data"sv);    // calls 2
}
```

Note: Default template parameters aren't part of the function signature (no SFINAE)

Concepts emulation (C++17)

Concept overloading

`if constexpr` (C++17)

```
template<class T>
void forward(T& t, std::string_view data) {
    if constexpr(Socket<T>) { // compile-time
        t.send(data);
    } else if constexpr(File<T>) {
        t.write(data);
    }
}
```

Note: Exactly the same way as with C++20 concepts

Concepts based design

Concepts based design

Goals

Expressiveness	Type constraints for better error messages
Loosely coupleled design	Inject all the things! (Policy Design)
Performance	Static dispatch by default (based on concepts)
Flexiblity	Dynamic dispatch using type erasure (based on the same concepts)
Testability	Automatic mocks injection (based on the same concepts)

Concepts based design

Static polymorphism

```
template<class T> concept Drawable =  
    requires (T t, std::ostream& out) { { t.draw(out) } -> void; };
```

```
struct Square {  
    void draw(std::ostream& out) { out << "Square"; } };  
  
struct Circle {  
    void draw(std::ostream& out) { out << "Circle"; } };  
  
template<Drawable T>  
void f(T& d) { d.draw(std::cout); }  
  
int main() {  
    f(Square{}); // prints Square  
    f(Circle{}); // prints Circle  
}
```

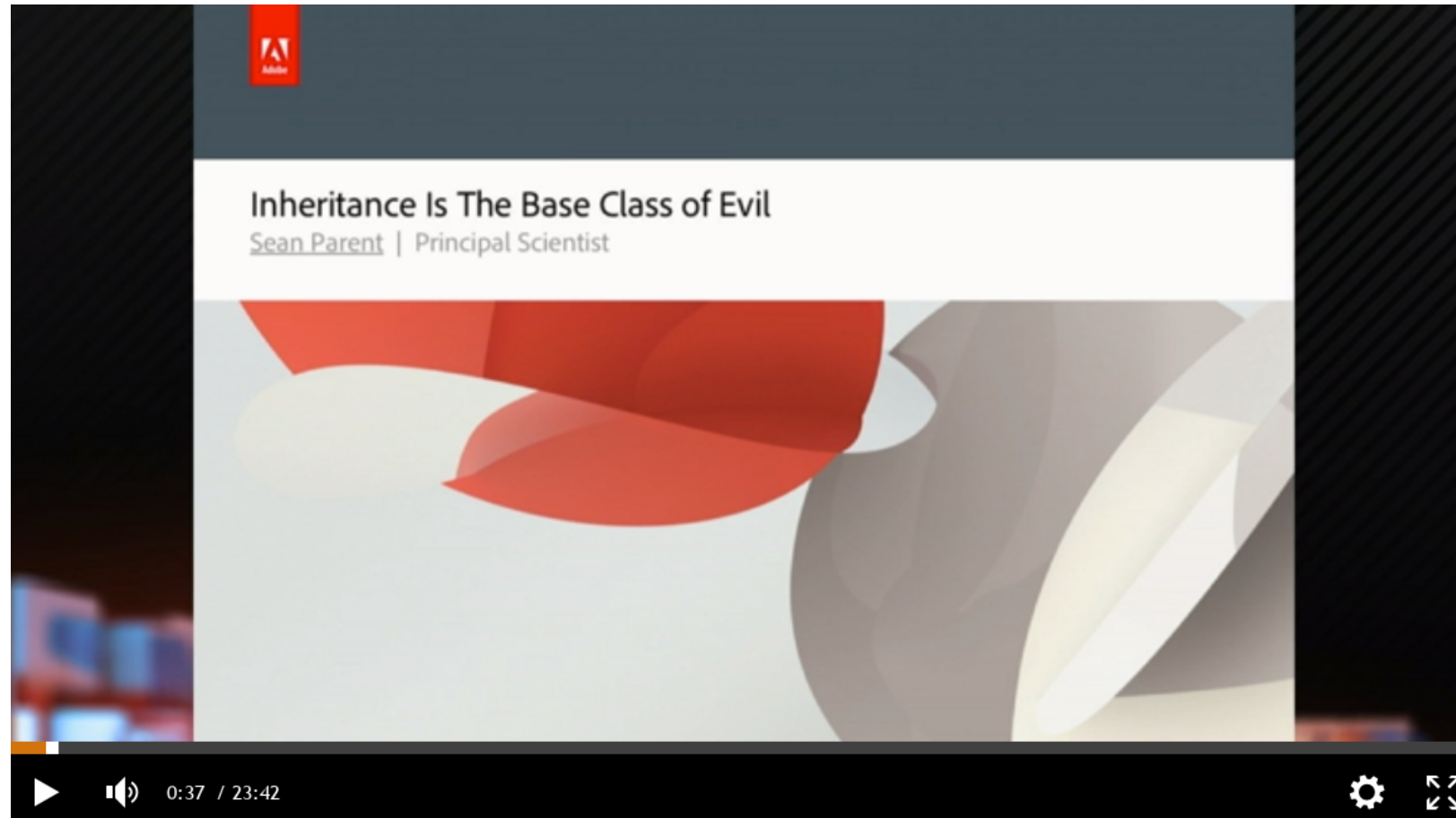

Concepts based design

Static polymorphism

```
std::vector v1 = { Square{}, Circle{} };  
// ERROR: class template argument deduction failed  
  
std::vector<auto> v2 = { Square{}, Circle{} };  
// ERROR: couldn't deduce template parameter  
  
std::vector<Drawable> v3 = { Square{}, Circle{} };  
// ERROR: couldn't deduce template parameter
```

Concepts based design

Concepts based polymorphism / Dynamic polymorphism



Inheritance-Is-The-Base-Class-of-Evil

Concepts based design

Dynamic polymorphism / type erasure

```
class Drawable {  
    void* ptr_{}; // ??? Small Buffer Optimization (SBO)  
    struct {  
        void (*draw)(std::ostream&);  
        void (*delete_ptr)(void*);  
    } const* const vptr_{};  
  
public:  
    template<class T> Drawable(T t) // non explicit  
        : ptr_{new T{t}}, vptr_{  
            [](void* self) { static_cast<T*>(self)->draw(); },  
            [](void* self) { delete static_cast<T*>(self); } }  
        { }  
    ~Drawable() { vptr_->delete_ptr(ptr_); }  
  
    void draw(std::ostream& os) { vptr_->draw(os); }  
};
```

Concepts based design

Dynamic polymorphism / Virtual concepts (C++2?)

```
template<class T> concept Drawable =  
    requires() { auto T::draw(std::ostream&) -> void; };  
    // Signature requirement
```

```
struct Square {  
    void draw(std::ostream& out) { out << "Square"; } };  
  
struct Circle {  
    void draw(std::ostream& out) { out << "Circle"; } };  
  
void f(virtual Drawable& d) { d.draw(std::cout); }  
    // Type erasure  
  
int main() {  
    f(Square{}); // prints Square  
    f(Circle{}); // prints Circle  
}
```

Concepts based design

Dynamic polymorphism / Virtual concepts (C++2?)

Signature requirement

```
template<class T>
concept Fooable = requires() {
    auto T::foo() -> void; // NOT C++20!
};
```

Dynamic Generic Programming with Virtual Concepts

Note: Virtual concepts aren't part of C++20

Concepts based design

Dynamic polymorphism / Virtual concepts (C++2?)

```
std::vector<virtual Drawable> v; // okay (type erasure)
v.push_back(Square{});           // okay
v.push_back(Circle{});          // okay
```

100% value semantics / Stack based / Small buffer optimization (SBO)

Note: Might be also generated with Metaclasses (C++2?)

<pre>template<class T> any Foo auto foo() -> void; };</pre>	<pre>template<class T> \$any { // metaclass constexpr { ... } };</pre>
--	--

<https://wg21.link/p0707r0>

Concepts based design

Dynamic polymorphism / Virtual concepts emulation (C++17)

```
template<class T> constexpr auto Drawable =  
    Callable<void(T::*)(std::ostream&)>( $(draw) );
```

```
struct Square {  
    void draw(std::ostream& out) { out << "Square"; } };  
  
struct Circle {  
    void draw(std::ostream& out) { out << "Circle"; } };  
  
void f(any<$(Drawable)>& d) { d.draw(std::cout); }  
  
int main() {  
    f(Square{}); // prints Square  
    f(Circle{}); // prints Circle  
}
```

Concepts based design

Virtual concepts emulation (C++17)

```
Callable<void(T::*)(std::ostream&)>( $(draw)) ]  
    \_      \_____ \_____ \____-> name  
    \       \         \         \  
$(name)) [](auto&& r, auto&& t, auto&&... args) {  
    struct { // base class  
        auto name(decType(args)... args) ->  
            decType(self.name(args...)){ } {  
            // static polymorphism (CRTP)  
            return static_cast<decType(t) *>(this)->template  
                call<name, typename decType(r)::type>(args...);  
        }  
    } _; return _;  
}
```

```
$(type) decltype(type<...>)
```


Concepts based design

Dependency Injection (policy design) / concepts

```
template<class T>
concept ErrorPolicy =
    requires(T t, std::string_view msg) {
        requires CopyConstructible<T>;
        { t.onError(msg) } -> void;
    };
```

```
struct ThrowPolicy {
    void onError(std::string_view msg) { throw T{msg}; }
};

struct LogPolicy {
    void onError(std::string_view msg) { std::clog << T{msg} << '\n'; }
};
```

Concepts based design

Dependency Injection (policy design) / concepts

```
template<ErrorPolicy TPolicy = class Policy> // inject
struct App {
    TPolicy policy{};
    void run() { if (...) { policy.onError("error!"); } }
};
```

Bindings / Injection

```
Creatable injector = di::injector{
    di::bind<class Policy>.to<ThrowPolicy>()
};
injector.create<App>().run();
```

Same as...

```
App{ThrowPolicy{}}.run();
```

Concepts based design

Dependency Injection (policy design) / virtual concepts (C++2?)

```
class App {  
public:  
    explicit App(virtual ErrorPolicy policy)  
        : policy{policy}  
    { }  
  
    void run() {  
        if (...) { policy.onError("error!"); }  
    }  
  
private:  
    virtual ErrorPolicy policy{};  
};
```

Concepts based design

Dependency Injection (policy design) / virtual concepts (C++2?)

```
Creatable injector = di::make_injector(  
    di::bind<virtual ErrorPolicy>.to<LogPolicy>()  
);  
injector.create<App>().run();
```

Same as...

```
App{LogPolicy{}}.run();
```

<https://github.com/boost-experimental/di>

Concepts based design

Dependency Injection (`policy design`) / `virtual concepts emulation (C++17)`

```
template<class T>
constexpr auto ErrorPolicy =
    CopyConstructible<T> and Callable<void(T::*)()>( $((onError)) ); // expose
```

```
struct App {
    any<$(ErrorPolicy)> policy{};
    void run() { if (...) { policy.onError("error!"); } }
};
```

Concepts based design

Dependency Injection (policy design) / virtual concepts emulation (C++17)

```
Creatable injector = di::make_injector(  
    di::bind<$(ErrorPolicy)>.to<LogPolicy>()  
);  
injector.create<App>().run();
```

Same as...

```
App{LogPolicy{}}.run();
```

Note: Same wiring for static/dynamic polymorphism

Concepts based design

Mocking (testing)

Interface based mocking

```
struct ErrorPolicy {  
    virtual ~ErrorPolicy() = default;  
    virtual void onError(std::string_view) = 0;  
};  
GMock<ErrorPolicy> mock{};  
EXPECT_CALL(mock, onError("interface!")).Times(1);  
mock.onError("interface!");
```

Concepts based mocking

```
GMock<$(ErrorPolicy)> mock{};  
EXPECT_CALL(mock, onError("concept!")).Times(1);  
object(mock).onError("concept!");
```

Concepts based design

Mocking (testing)

automatic mocks injection

```
"should print read text"_test = [] {  
  auto [app, mocks] = testing::make<App>();  
  
  EXPECT_CALL(mocks<$(ErrorPolicy)>, onError("error!"));  
  
  app.run();  
};
```

Note: `make` creates mocks based on concepts requirements (reflection)

<https://github.com/cpp-testing/gunit>

Future of concepts (C++2X)

Future of concepts (C++2X)

Terse template syntax

```
void forward(Socket& socket, std::string_view data);
```

long form

```
template<Socket T>  
void forward(T& socket, std::string_view data);
```

longer form

```
template<class T> requires Socket<T>  
void forward(T& socket, std::string_view data);
```

Future of concepts (C++2X)

Terse template syntax - <https://wg21.link/p0696r1>

```
void forward(Socket, Socket);
```

long form

```
template<class T> void forward(T, T);
```

vs

```
void forward(auto, auto);
```

long form

```
template<class T, class U> void forward(T, U);
```

Future of concepts (C++2X)

Template-introduction syntax

```
Socket{T} void forward(T, auto);
```

or

```
template<Socket [T]> void forward(T, auto);
```

long form

```
template<class T, class U> void forward(T, U) requires Socket<T>;
```

Future of concepts (C++2X)

```
template<class T>
struct tcp_socket {
    static_assert(Socket<tcp_socket>); // always fail, tcp_socket is incomplete
};
```

Metaclasses syntax

```
template<class T> Socket tcp_socket { };
```

<https://wg21.link/p0707r0>

Summary

Provides better diagnostics

Simplify usage of SFINAE / `enable_if`

- Introspection by design / Optional interfaces

Well specified interfaces (precised documentation)

Can be emulated in C++14/C++17

- `variable templates` / `constexpr` / `constexpr if`

C++20 is just the beginning

- `syntax improvements` / `requirements improvements`
- `virtual concepts` / `metaclasses` / `static reflection`

Questions?

C++20 draft	http://eel.is/c++draft/temp.constr
Concepts	https://wg21.link/P0734R0
Virtual Concepts	https://github.com/andyprowl/virtual-concepts/blob/master/draft/Dynamic Generic Programming with Virtual Concepts.pdf

