

C++ Slack

<https://cpplang-inviter.cppalliance.org>

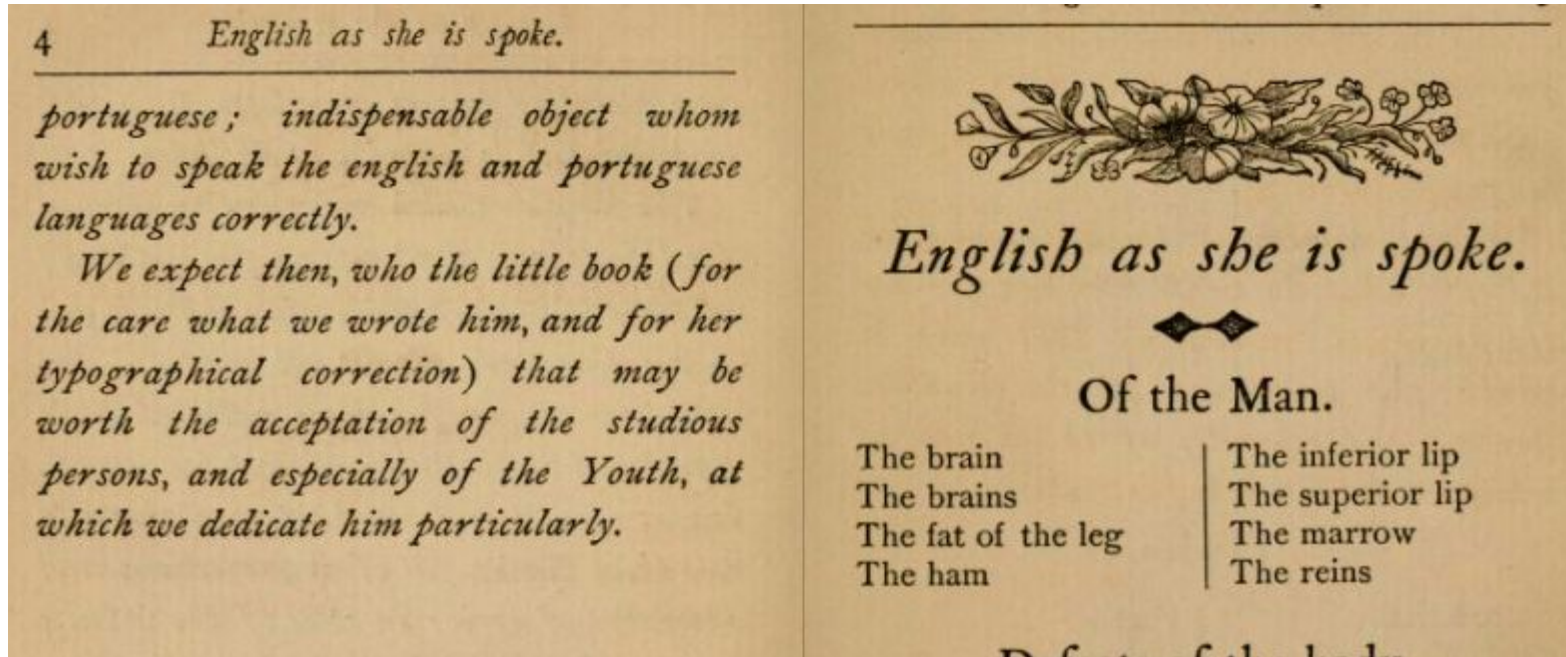
Channel #ug_us_nyc

Concepts as she is spoke

Concepts in the C++2a Working Draft

Arthur O'Dwyer
2019-01-10

What's with the title?



<https://publicdomainreview.org/collections/english-as-she-is-spoke-1884/>

What's with the title?

Apprendi ô francêz.

Dõe-me à cabeça.

Não tenho tẽmpo.

Não pôsso demorár-me.

Conhêço-o dê vista.

Tenho dê ôu dêvo sair.

Tenho dê ôu dêvo recebêr dinhêiro nõ
fim dô mêz.

Dár-vôs- hêi ôu dár-lhê-hêi ó sêu ende-
rêço.

Refiro-me ao quê dizêis ôu díz.

Não sôu tôlo.

Não entẽdo ôu intendo isso.

I have learned the french language.

My head is sick.

I have no time.

I cannot to stayme.

I know him by sight.

I have to go out.

*I have any money to receive at last
month.*

I will give you on's adress.

I report me at this you tell me.

I am not to silly.

I not understand that

<https://books.google.com/books?id=1Ud5AAAAIAAJ&pg=PA75>

Outline

- High-level overview of C++17 TMP and C++2a Concepts
 - SFINAE by any other name [5–24]
- Subsumption and normal form [25–54]
- Syntax of a `requires-expression` [55–71]
 - Some traps for the unwary [72–81]
- Terse/abbreviated syntaxes and future directions [82–97]
 - “Abbreviated Function Templates” (“AFTs”)
 - Guidelines for using C++2a Concepts [92–94]
 - What C++2a Concepts still can’t do [95–97]

Hey look!
Slide numbers!

What's the state of Concepts?

- The Concepts Technical Specification (“Concepts TS”)
- The part of it merged into the C++2a working draft in Toronto (July 2017)
- A novel terse syntax (“AFTs”) merged in San Diego (Nov 2018)

GCC supports almost all of Concepts TS, including an obsolete form of AFTs.

```
g++ -std=c++2a -fconcepts -Dconcept="concept bool" test.cc
```

Saar Raz's fork of Clang supports (almost all of) C++2a Concepts pre-San Diego, but not AFTs.

```
clang++* -std=c++2a test.cc
```

MSVC 2018 can kinda parse Concepts TS syntax but basically treats it as whitespace; it doesn't “support” Concepts in any meaningful sense.

```
cl -std:c++latest -experimental:concepts test.cc
```

<https://godbolt.org>

What is a *concept*, anyway?

```
template<class T>
std::string stringify(const T& t) {
    std::ostringstream oss;
    oss << t;
    return std::move(oss).str();
}
```

What *kinds* of arguments can we pass to this function?

- “Anything with an << operator”
- Just that one requirement? Probably doesn’t need Concepts.

What is a *concept*, anyway?

```
template<class Body>
struct http_request {
    int version;
    string method, target;
    map<string, string> fields;
    Body::value_type body;

    void read(istream& is) {
        read_header(is, *this);
        Body::read(is, this->body);
    }
    void write(ostream& os) const {
        write_header(os, *this);
        Body::write(os, this->body);
    }
};
```

This example comes from
Vinnie Falco's CppCon 2017 session
"Make Classes Great Again"

<https://youtu.be/WsUnnYEKPnI>

What is a *concept*, anyway?

```
template<class Body>
struct http_request {
    int version;
    string method, target;
    map<string, string> fields;
    Body::value_type body;

    void read(istream& is) {
        read_header(is, *this);
        Body::read(is, this->body);
    }
    void write(ostream& os) const {
        write_header(os, *this);
        Body::write(os, this->body);
    }
};
```

Many requirements:

- Body::value_type
- Body::read
- Body::write

Let's also say we want to prevent callers from using http_request with a “half-implemented” Body type. Implementing Body::write() but not Body::read(), for example. Let's forbid that.

This example comes from
Vinnie Falco's CppCon 2017 session
“Make Classes Great Again”

<https://youtu.be/WsUnnYEKPNi>

Defining a concept (old-school)

```
template<class B, class = void> struct is_body : false_type;

template<class B>
struct is_body<B, void_t<
    typename B::value_type,
    decltype(
        B::read(declval<istream&>(), declval<typename B::value_type&>())
    ),
    decltype(
        B::write(declval<ostream&>(), declval<typename B::value_type const&>())
    )
>> : true_type {};

static_assert(is_body<file_body>::value);
static_assert(is_body<string_body>::value);
static_assert(not is_body<int>::value);
```

This example again comes from
Vinnie Falco's CppCon 2017 session
“Make Classes Great Again”
<https://youtu.be/WsUnnYEKPnI>

Defining a concept (C++2a)

```
template<class B>
concept Body = requires(
    istream& is,
    ostream& os,
    typename B::value_type& b,
    typename B::value_type const& cb)
{
    typename B::value_type; // redundant
    B::read(is, b);
    B::write(is, cb);
};

static_assert(Body<file_body>);
static_assert(Body<string_body>);
static_assert(not Body<int>);
```

Asserting and SFINAEing (old-school)

```
template<class Body>
struct http_request {
    static_assert(is_body<Body>::value, "Body requirements are not satisfied");
};
```

```
template<class Body, enable_if_t<is_body<Body>::value, int> = 0>
void do_something() { puts("yes"); }
```

```
template<class NonBody, enable_if_t<not is_body<NonBody>::value, int> = 0>
void do_something() { puts("no"); }
```

```
do_something<file_body>(); // prints "yes"
do_something<int>();       // prints "no"
```



This enable_if prevents ambiguity during the overload resolution of do_something<file_body>().

Asserting and SFINAEing (C++2a)

```
template<class B>
struct http_request {
    static_assert(Body<B>, "Body requirements are not satisfied");
};
```

```
template<class B> requires Body<B>
void do_something() { puts("yes"); }
```



This template is
“constrained.”

```
template<class B>
void do_something() { puts("no"); }
```



This template is
“unconstrained,”
and therefore a
“less good” match
during overload
resolution.

```
do_something<file_body>(); // prints "yes"
do_something<int>();       // prints "no"
```

Something else about the old way...

```
template<class Body, class = enable_if_t<is_body<Body>::value>>  
void do_something() { puts("yes"); }
```

We're basically abusing template parameters here — we have a template type *parameter* that is not being used to *parameterize*!

A caller who knows we're doing this can “hijack” our implementation.

```
do_something<int>();           // correctly does not compile
```

```
do_something<int, char**>();  // prints "yes" (yikes!)
```

requires clauses aren't template parameters

```
template<class Body> requires is_body<Body>::value  
void do_something() { puts("yes"); }
```

The constraint is properly relegated to a requires clause, which is not a template parameter and thus cannot be hijacked.

```
do_something<int>();           // correctly does not compile
```

```
do_something<int, char**>();  // also does not compile
```

requires-clauses are the single best feature of C++2a Concepts.

**Q: How do requires-clauses
interact with name-mangling?**

A: They don't. (Yet.)

requires doesn't affect mangling*

```
template<class T, class = enable_if_t<is_integral_v<T>>>
T foo(T t) { return t + 1; }
```

```
template int foo<int>(int);           // _Z3fooIiVET_S0_
template short foo<short>(short);     // _Z3fooIsVET_S0_
template float foo<float>(float);     // does not compile
```

}

old-school
enable_if

```
template<class T> requires is_integral_v<T>
T bar(T t) { return t + 1; }
```

```
template int bar<int>(int);           // _Z3barIiET_S0_
template short bar<short>(short);     // _Z3barIsET_S0_
template float bar<float>(float);     // does not compile
```

}

new-school
requires

requires doesn't affect mangling*

```
template<class T, enable_if_t<is_integral_v<T>, int> = 0>  
T foo(T t) { return t + 1; }  
template<class T, enable_if_t<not is_integral_v<T>, int> = 0>  
T foo(T t) { return t + 2; }
```

} old-school
enable_if

```
template int foo<int>(int);           // _Z3fooIiLi0EET_S0_  
template short foo<short>(short);    // _Z3fooIsLi0EET_S0_  
template float foo<float>(float);    // _Z3fooIfLi0EET_S0_
```

```
template<class T> requires is_integral_v<T>  
T bar(T t) { return t + 1; }  
template<class T> requires not is_integral_v<T>  
T bar(T t) { return t + 2; }
```

} new-school
requires

```
template int bar<int>(int);           // _Z3barIiET_S0_  
template short bar<short>(short);    // _Z3barIsET_S0_  
template float bar<float>(float);    // _Z3barIfET_S0_
```

Duplicate function templates are OK

We need a uniquely mangled name for each *template function*, but not necessarily for each *function template*.

```
template<class T> requires is_integral_v<T>
T bar(T t) { return t + 1; }           // "A"
```

```
template<class T> // unconstrained
T bar(T t) { return t + 2; }           // "B"
```

```
template int bar<int>(int);             // _Z3barIiET_S0_ specializes template "A"
template short bar<short>(short);        // _Z3barIsET_S0_ specializes template "A"
template float bar<float>(float);         // _Z3barIfET_S0_ specializes template "B"
```

Different story with class templates

A constraint on a class, variable, or alias template is forever.
But you can place different constraints on different partial specializations.

```
template<class T>
class Widget { ... };

template<class U> requires is_pointer_v<U>
class Widget { ... };           // Error: redefinition of class template "Widget"

template<class V> requires is_pointer_v<V>
class Widget<V> { ... };       // OK: constrained partial specialization
```

GCC doesn't fully support
constrained alias templates.

Remember this old-school trap?

Recall a trap I discussed in “Template Normal Programming, Part 1.”

```
template<class T>
T baz(T t) { return t + 1; }           // "A"

template<>
int *baz(int *t) { return t + 2; }    // _Z3bazIPiET_S1_ specializes template "A"

template<class T> // "better match"
T *baz(T *t) { return t + 3; }        // "B"
```

At this point, if we call `baz(&i)`, overload resolution will consider two candidates:

- the template function `baz<int*>(int*)` that could be instantiated from template "A"
- the template function `baz<int>(int*)` that could be instantiated from template "B"

"B" is a better match than "A".

So the compiler uses template "B" to instantiate `baz<int>(int*)` a.k.a. `_Z3bazIiEPT_S1_ ...` and our explicitly specialized `baz<int*>(int*)` a.k.a. `_Z3bazIPiET_S1_` is never called!

With requires, the stakes are higher*

```
template<class T>
T baz(T t) { return t + 1; }           // "A"

template<>
int *baz(int *t) { return t + 2; }    // _Z3bazIPiET_S1_ specializes template "A"

template<class T> requires is_pointer_v<T> // "better match"
T baz(T t) { return t + 3; }           // "B"
```

At this point, if we call `baz(&i)`, overload resolution will consider two candidates:

- the template function `baz<int*>(int*)` that could be instantiated from template "A"
- the template function `baz<int*>(int*)` that could be instantiated from template "B"

"B" is a better match than "A", because "B" is constrained and "A" is not.

So the compiler uses template "B" to instantiate `baz<int*>(int*)` a.k.a. `_Z3bazIPiET_S1_ ...` which has exactly the same mangled name as our explicitly specialized `baz<int*>(int*)` a.k.a. `_Z3bazIPiET_S1_!` This is a violation of the One Definition Rule, and causes undefined behavior.

Two distinct “better match” criteria

```
template<class T>  
T foo(T t) { return t + 1; }           // "A"
```

```
template<class T> requires is_pointer_v<T>  
T foo(T t) { return t + 2; }         // "B"
```

```
template<class T>  
T *foo(T *t) { return t + 3; }       // "C"
```

When we call `foo(&i)`,

- "C" is a better match than "A" because its function parameter list is more highly specialized.
- "C" is a better match than "B" because its function parameter list is more highly specialized.
- "B" is a better match than "A" because it is more constrained.

Two distinct “better match” criteria

```
template<class T>  
T foo(T t) { return t + 1; }           // "A"
```

```
template<class T> requires is_pointer_v<T>  
T foo(T t) { return t + 2; }         // "B"
```

```
template<class T>  
T *foo(T *t) { return t + 3; }       // "C"
```

The old-school rule has highest priority! Look first at the parameter list; **then** (and only then) consider the requires clause.

When we call `foo(&i)`,

- "C" is a better match than "A" because its function parameter list is more highly specialized.
- "C" is a better match than "B" because its function parameter list is more highly specialized.
- "B" is a better match than "A" because it is more constrained.

What is “more constrained” exactly?

```
template<class T>  
T foo(T t) { return t + 1; }           // "A"
```

```
template<class T> requires is_integral_v<T>  
T foo(T t) { return t + 2; }           // "B"
```

- "B" is a better match than "A" because it is more constrained.

What is “more constrained” exactly?

```
template<class T>  
T foo(T t) { return t + 1; }           // "A"
```

```
template<class T> requires is_integral_v<T>  
T foo(T t) { return t + 2; }         // "B"
```

- "B" is a better match than "A" because it is more constrained.

```
template<class T> requires (sizeof(T) == 4)  
T foo(T t) { return t + 3; }         // "C"
```

- Intuitively, neither "B" nor "C" is “more constrained” than the other.

What is “more constrained” exactly?

```
template<class T>  
T foo(T t) { return t + 1; }           // "A"
```

```
template<class T> requires is_integral_v<T>  
T foo(T t) { return t + 2; }           // "B"
```

- "B" is a better match than "A" because it is more constrained.

```
template<class T> requires (sizeof(T) == 4)  
T foo(T t) { return t + 3; }           // "C"
```

- Intuitively, neither "B" nor "C" is “more constrained” than the other.

```
template<class T> requires (sizeof(T) != 0)  
T foo(T t) { return t + 4; }           // "D"
```

??

What is “more constrained” exactly?

```
template<class T>  
T foo(T t) { return t + 1; }           // "A"
```

```
template<class T> requires is_integral_v<T>  
T foo(T t) { return t + 2; }           // "B"
```

- "B" is a better match than "A" because it is more constrained.

```
template<class T> requires (sizeof(T) == 4)  
T foo(T t) { return t + 3; }           // "C"
```

- Intuitively, neither "B" nor "C" is “more constrained” than the other.

```
template<class T> requires (sizeof(T) != 0)  
T foo(T t) { return t + 4; }           // "D"
```

- Logically "D" is less constrained than "C", but we don't seriously expect the compiler to realize that. So intuitively it makes sense that all of "B", "C", and "D" are *equally much* constrained.

What is “more constrained” exactly?

So do we only have two levels — “constrained” and “unconstrained”?
Let’s try some things we think the compiler might be able to figure out.

```
template<class T> requires is_integral_v<T>
T foo(T t) { return t + 6; }           // "F"

template<class T> requires is_integral_v<T> && is_signed_v<T>
T foo(T t) { return t + 7; }           // "G"

int x = foo(0);
```

What do we expect to happen here? Will the compiler pick "G", or say there’s an ambiguity?

What is “more constrained” exactly?

So do we only have two levels — “constrained” and “unconstrained”?
Let’s try some things we think the compiler might be able to figure out.

```
template<class T> requires is_integral_v<T>
T foo(T t) { return t + 6; }           // "F"

template<class T> requires is_integral_v<T> && is_signed_v<T>
T foo(T t) { return t + 7; }          // "G"

int x = foo(0);
```

Trick question! The Concepts TS (GCC) is actually happy with this. But the C++2a version of Concepts (Clang) says “no”:

```
<source>:12:9: error: call to 'foo' is ambiguous
int x = foo(0);
      ^~~
```

What is “more constrained” exactly?

```
template<class T>  
concept Integral = is_integral_v<T>;
```

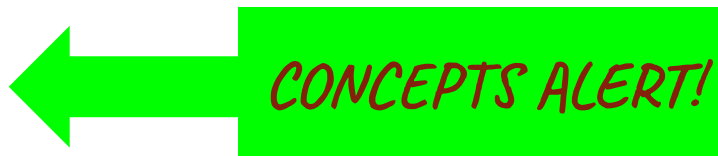
```
template<class T> requires Integral<T>  
T foo(T t) { return t + 10; }
```

// "J"

```
template<class T> requires Integral<T> && is_signed_v<T>  
T foo(T t) { return t + 11; }
```

// "K"

```
int x = foo(0);
```



What do we expect to happen **here**? Will the compiler pick "K", or say there's an ambiguity?

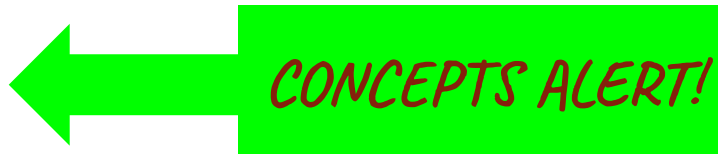
What is “more constrained” exactly?

```
template<class T>  
concept Integral = is_integral_v<T>;
```

```
template<class T> requires Integral<T>  
T foo(T t) { return t + 10; }
```

```
template<class T> requires Integral<T> && is_signed_v<T>  
T foo(T t) { return t + 11; }
```

```
int x = foo(0);
```



// "J"

// "K"

The compiler picks "K"!

Subsumption

Constraints are logical formulas!



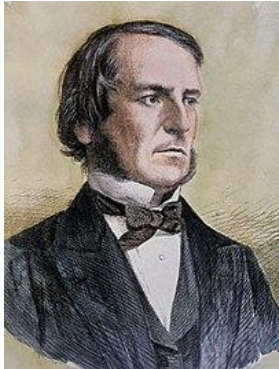
requires-clause :

requires constraint-logical-or-expression

constraint-logical-or-expression :

constraint-logical-and-expression

constraint-logical-or-expression | | constraint-logical-and-expression



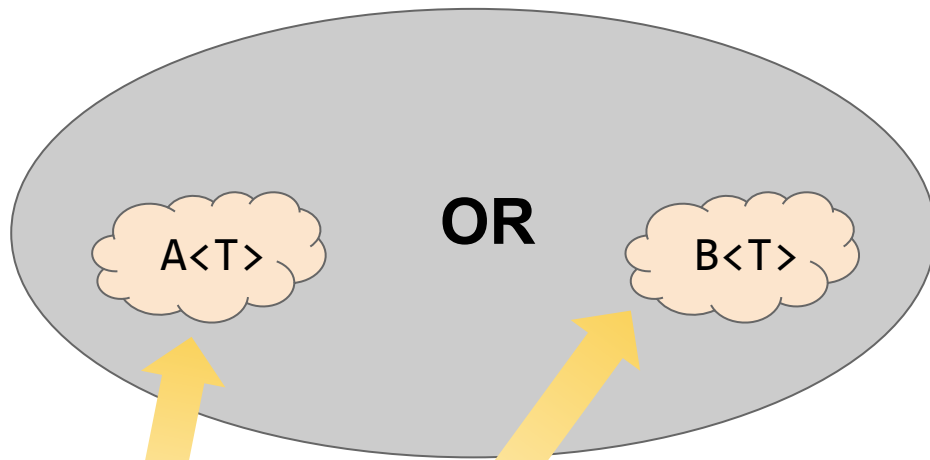
constraint-logical-and-expression :

primary-expression

constraint-logical-and-expression && primary-expression

Both Clang and GCC accept *inclusive-or-expression* where the wording currently says “*primary-expression*.” I expect the wording to catch up soon.

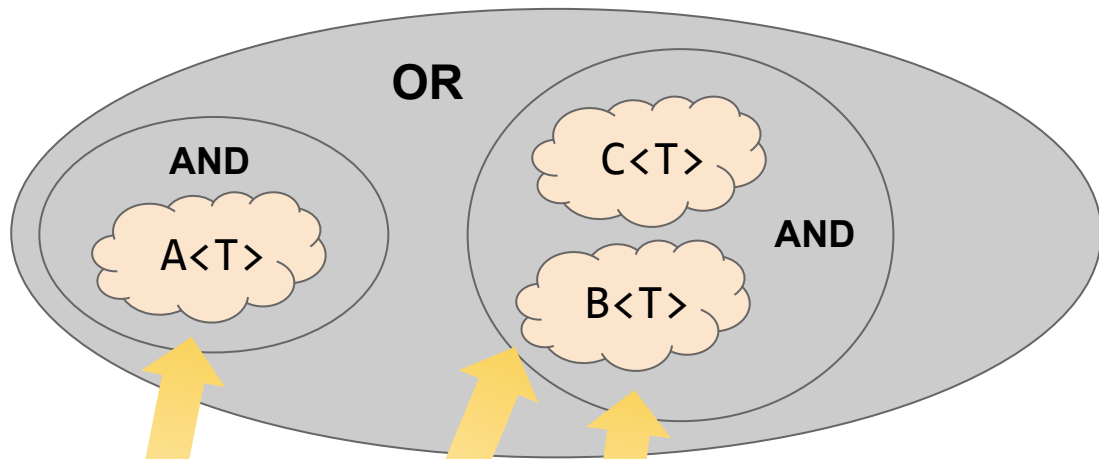
Normalization and normal form



This cloudy Platonic diagram is called the constraint's "normal form." It is a mathematical construct, not a C++ source code construct.

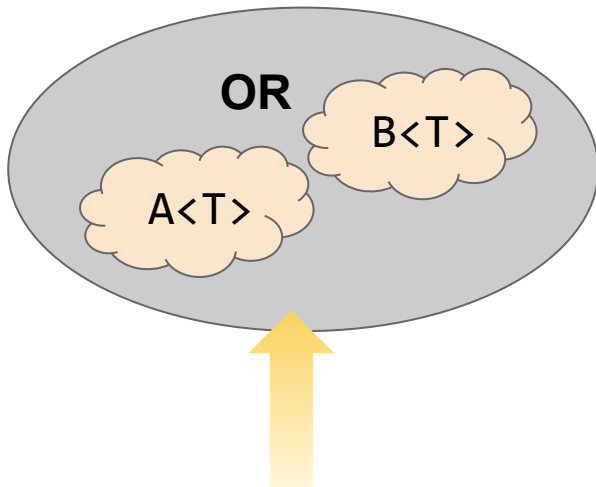
```
template<class T> void foo(T)
  requires A<T> || B<T>;
```

Normalization and normal form



```
template<class T> void foo(T)
  requires A<T> || (B<T> && C<T>);
```

Logical operations are commutative



```
template<class T> void foo(T)
    requires A<T> || B<T>;
```

```
template<class T> void foo(T)
    requires B<T> || A<T>;
```

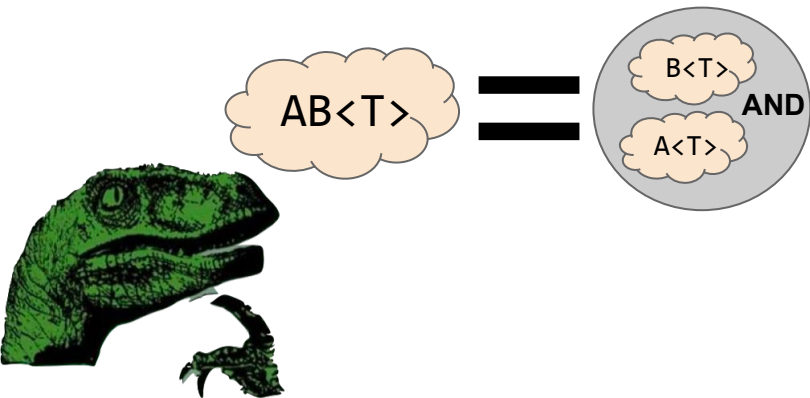
These two
requires-clauses have
exactly the same *normal*
form.

Non-atomic concepts

```
template<class T> concept AB =  
    A<T> && B<T>;
```

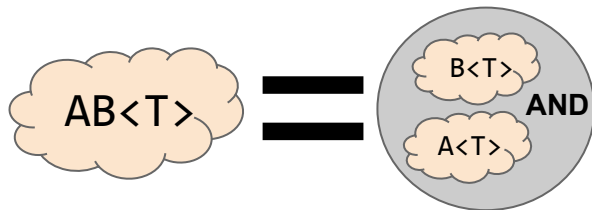
Non-atomic concepts

```
template<class T> concept AB =  
    A<T> && B<T>;
```

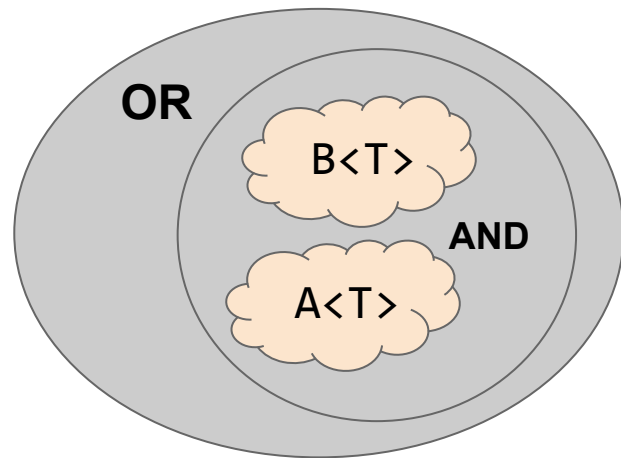


Non-atomic concepts

```
template<class T> concept AB =  
    A<T> && B<T>;
```



```
template<class T> void foo(T)  
    requires AB<T>;  
template<class T> void foo(T)  
    requires B<T> && A<T>;
```



These two
requires-clauses have
exactly the same *normal*
form.

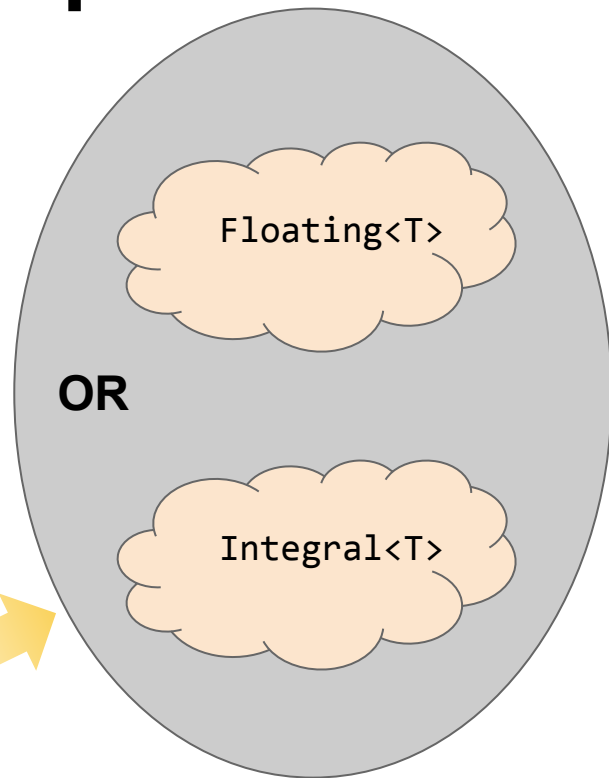
We always bottom out at expressions

```
template<class T> concept Scalar =  
    is_scalar_v<T>;
```

```
template<class T> concept Integral =  
    Scalar<T> && is_integral_v<T>;
```

```
template<class T> concept Floating =  
    Scalar<T> && is_floating_point_v<T>;
```

```
template<class T> void foo(T)  
    requires Integral<T> || Floating<T>;
```



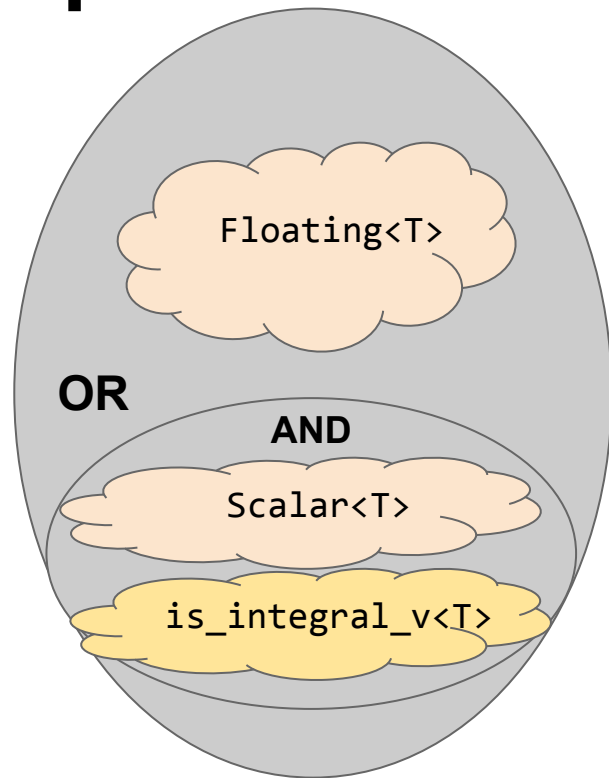
We always bottom out at expressions

```
template<class T> concept Scalar =  
    is_scalar_v<T>;
```

```
template<class T> concept Integral =  
    Scalar<T> && is_integral_v<T>;
```

```
template<class T> concept Floating =  
    Scalar<T> && is_floating_point_v<T>;
```

```
template<class T> void foo(T)  
    requires Integral<T> || Floating<T>;
```



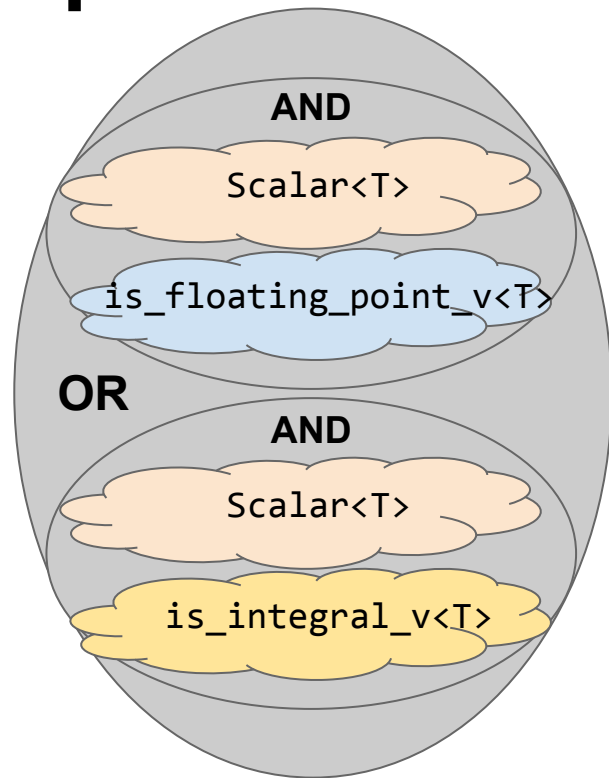
We always bottom out at expressions

```
template<class T> concept Scalar =  
    is_scalar_v<T>;
```

```
template<class T> concept Integral =  
    Scalar<T> && is_integral_v<T>;
```

```
template<class T> concept Floating =  
    Scalar<T> && is_floating_point_v<T>;
```

```
template<class T> void foo(T)  
    requires Integral<T> || Floating<T>;
```



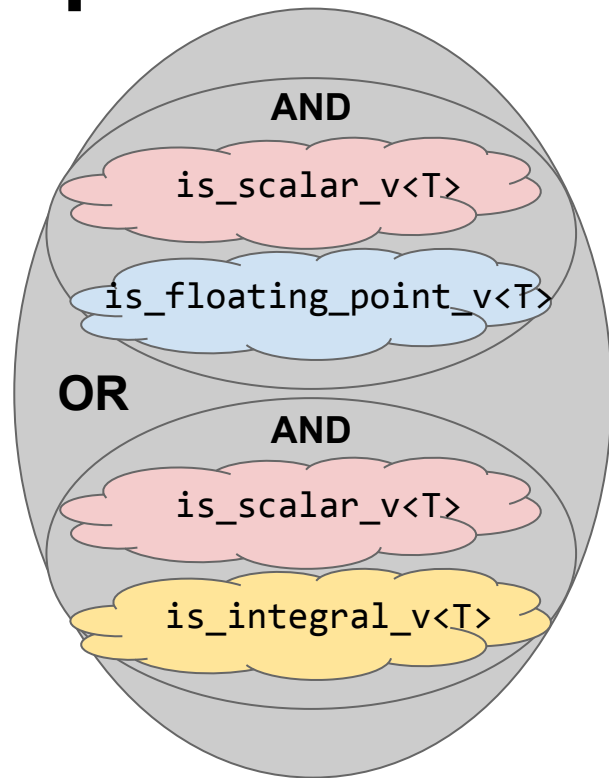
We always bottom out at expressions

```
template<class T> concept Scalar =  
    is_scalar_v<T>;
```

```
template<class T> concept Integral =  
    Scalar<T> && is_integral_v<T>;
```

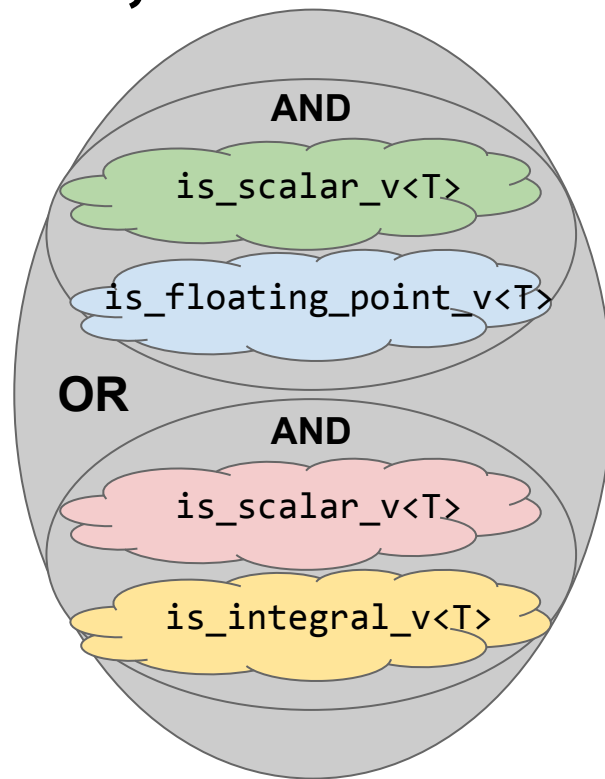
```
template<class T> concept Floating =  
    Scalar<T> && is_floating_point_v<T>;
```

```
template<class T> void foo(T)  
    requires Integral<T> || Floating<T>;
```



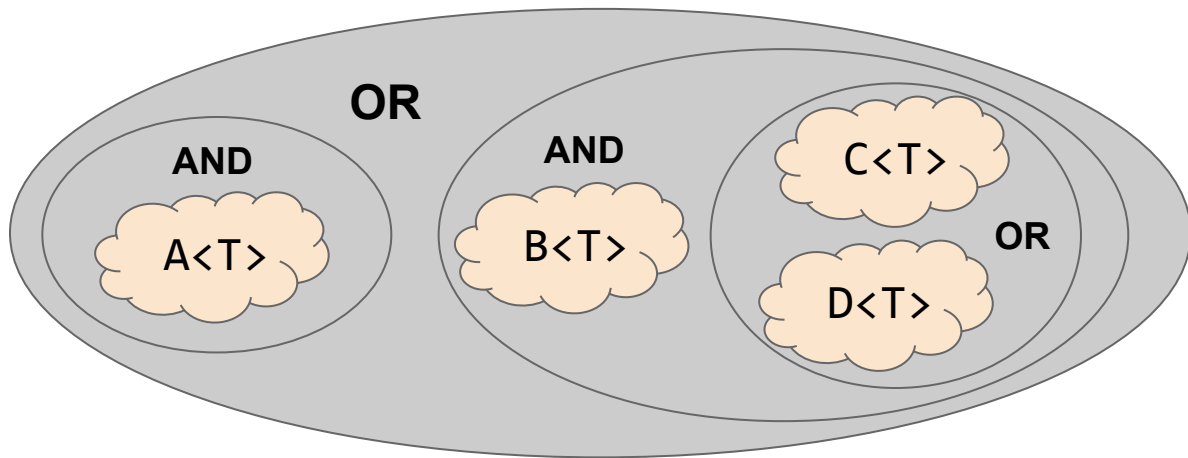
...indexed by source position, not text*

```
template<class T> concept Integral =  
    is_scalar_v<T> &&  
        is_integral_v<T>;  
  
template<class T> concept Floating =  
    is_scalar_v<T> &&  
        is_floating_point_v<T>;  
  
template<class T> void foo(T)  
    requires Integral<T> || Floating<T>;
```



This is slide 31's trick question. The Concepts TS actually did try to deduplicate expressions based on textual spelling, not source position.

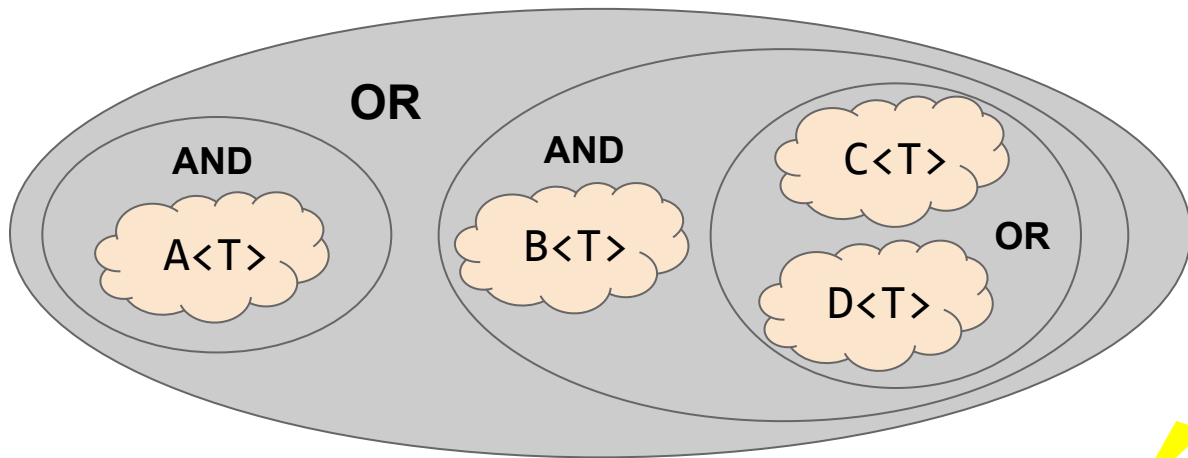
Only two levels are needed



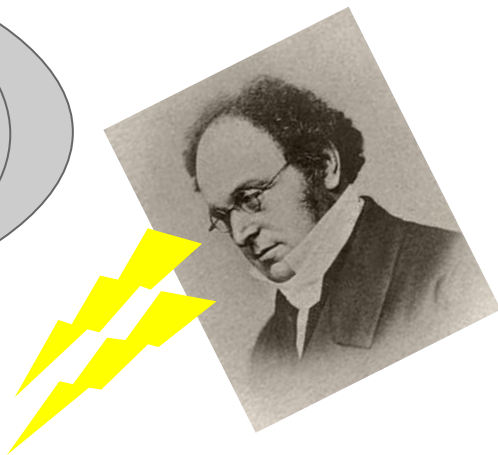
??

```
template<class T> void foo(T)
    requires A<T> || (B<T> && (C<T> || D<T>));
```

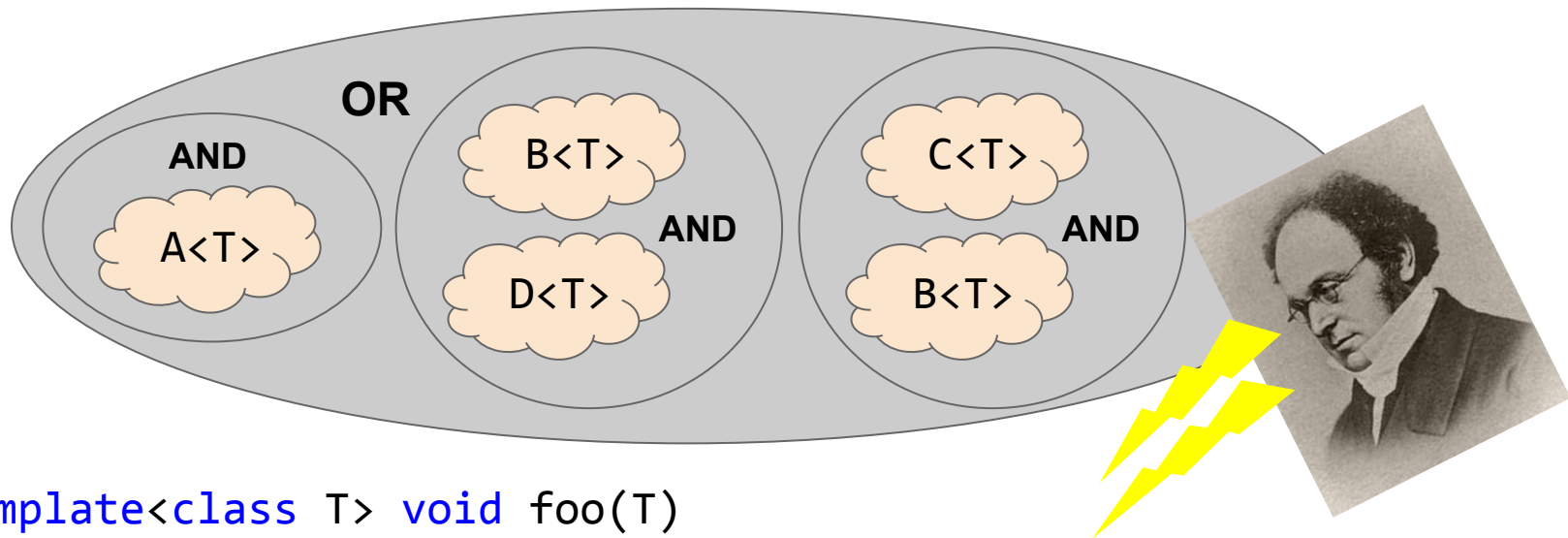
Only two levels are needed



```
template<class T> void foo(T)
    requires A<T> || (B<T> && (C<T> || D<T>));
```



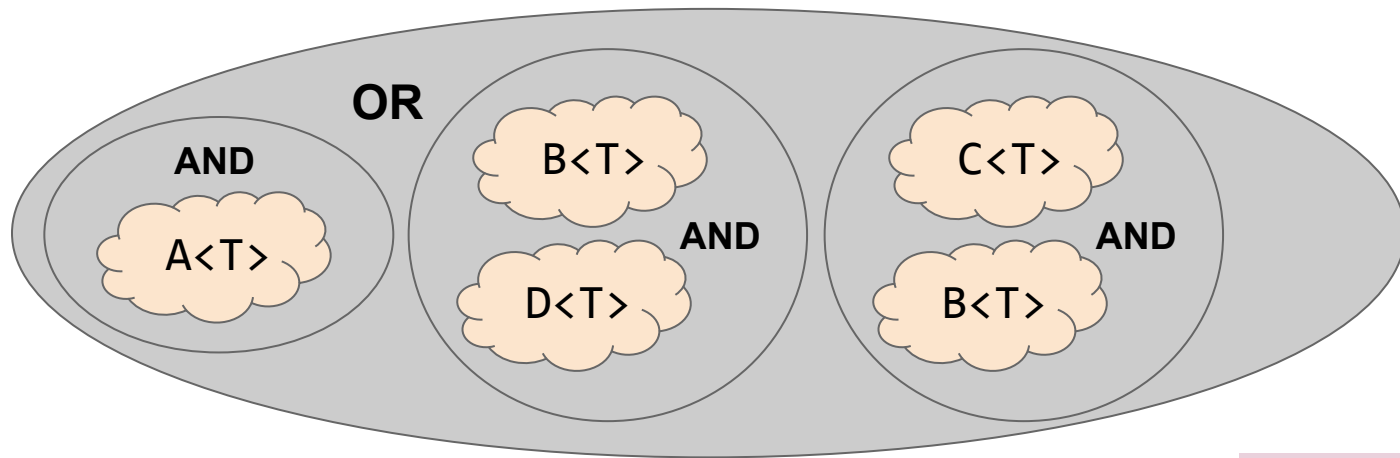
Only two levels are needed



```
template<class T> void foo(T)
    requires A<T> || (B<T> && (C<T> || D<T>));
```

```
template<class T> void foo(T)
    requires A<T> || (B<T> && C<T>) || (B<T> && D<T>);
```

Only two levels are needed

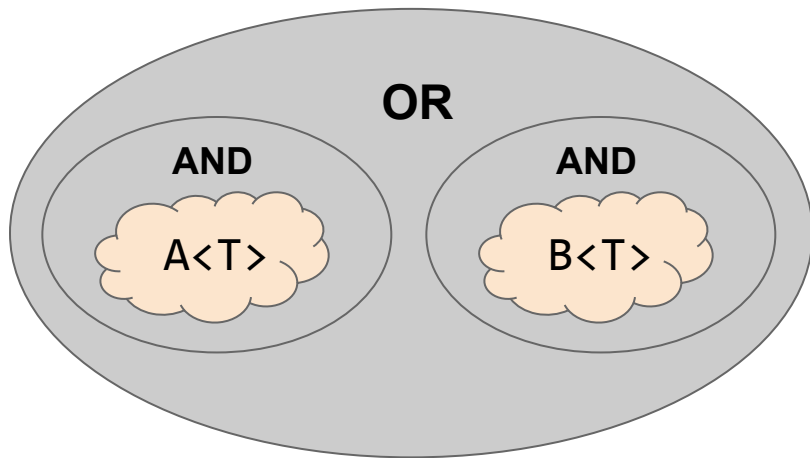


```
template<class T> void foo(T)
    requires A<T> || (B<T> && (C<T> || D<T>));
```

```
template<class T> void foo(T)
    requires A<T> || (B<T> && C<T>) || (B<T> && D<T>);
```

These two
requires-clauses have
exactly the same *normal*
form.

What about negation?

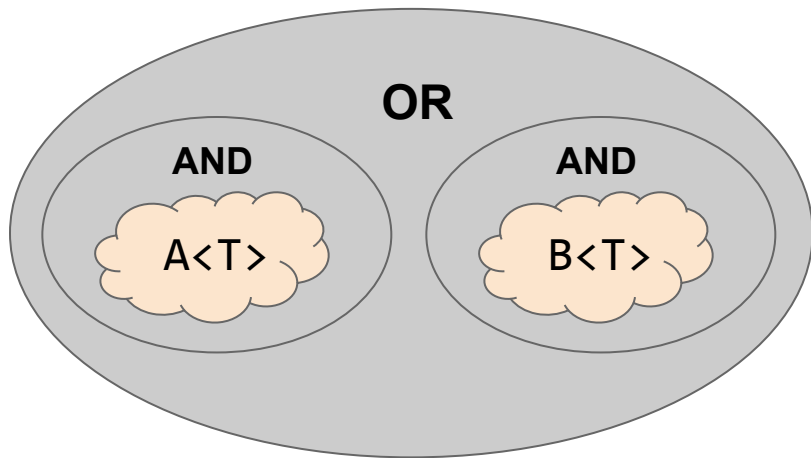


??

```
template<class T> void foo(T)
    requires not (not A<T> || not B<T>);
```

```
template<class T> void foo(T)
    requires (A<T> && B<T>);
```

What about negation?

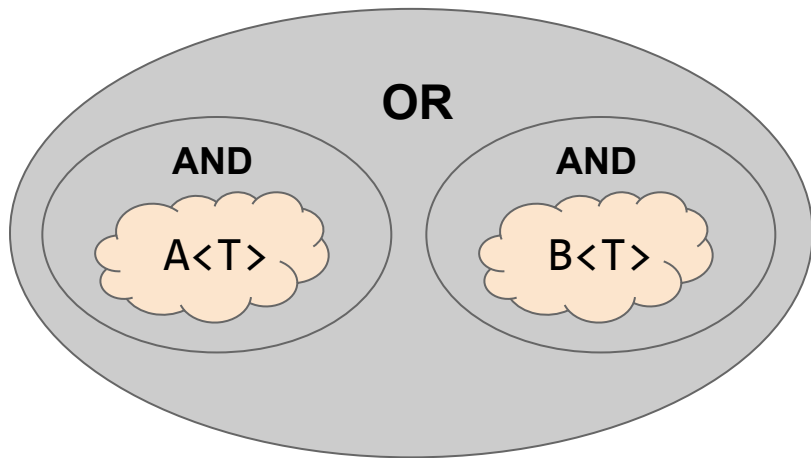


```
template<class T> void foo(T)
    requires not (not A<T> || not B<T>);
```

```
template<class T> void foo(T)
    requires (A<T> && B<T>);
```



What about negation? NO!

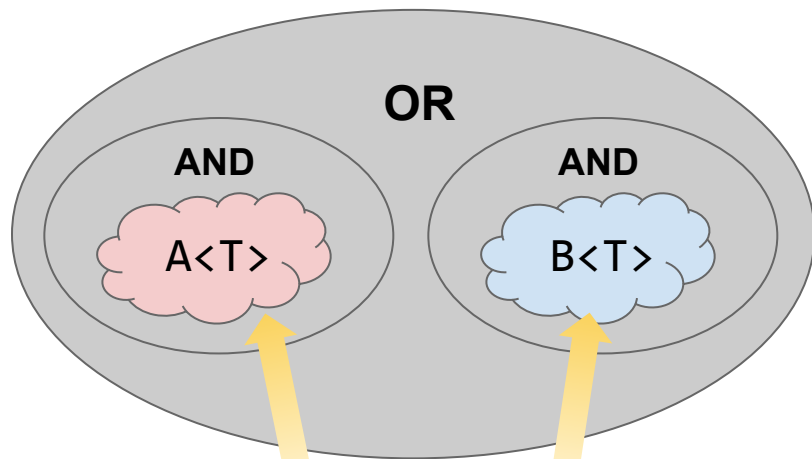


```
template<class T> void foo(T)
    requires not (not A<T> || not B<T>);
```

```
template<class T> void foo(T)
    requires (A<T> && B<T>);
```

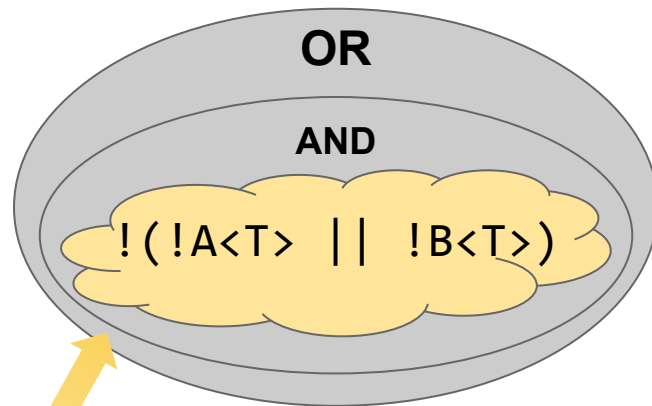


What about negation? NO!



```
template<class T> void foo(T)
  requires (A<T> && B<T>);
```

```
template<class T> void foo(T)
  requires not (not A<T> || not B<T>);
```



The grammar has special cases for top-level “||” and “&&”, but not for “!”. These two requires-clauses are **functionally equivalent**, but neither one **subsumes** the other.

What about negation? NO!



The situation is similar to “covariance and contravariance” with virtual methods. Just as in that case, C++2a supports covariance but not contravariance.

```
template<class T> void foo(T) requires Scalar<T>;  
    // ...is subsumed by the more constrained template...  
template<class T> void foo(T) requires Integral<T>;
```

```
template<class T> void foo(T) requires !Integral<T>;  
    // ...has no subsumption relationship at all with...  
template<class T> void foo(T) requires !Scalar<T>;
```

Defining a Concept

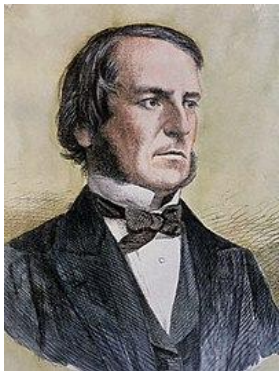
Defining a concept

```
template <template-parameter-list>  
concept concept-name = constraint-expression;
```

Kinds of *constraint-expressions*



- Disjunction — `a || b`
- Conjunction — `a && b`
- Primary expression of type `bool`
 - Type trait — `is_integral_v<T>`
 - A `requires`-expression



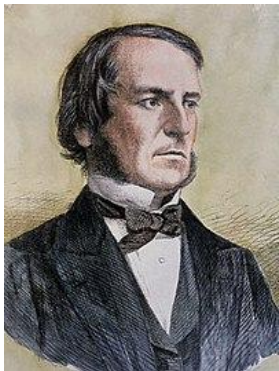
Kinds of *constraint-expressions*



- Disjunction — `a || b`
- Conjunction — `a && b`
- Primary expression of type `bool`
 - Type trait — `is_integral_v<T>`
 - A `requires`-expression



A `requires`-expression is **not the same** as a `requires`-clause.



Noexcept-expression vs. clause

Noexcept-clause is part of a declaration:

“This is noexcept when...”

```
template<class T>  
void xyzzy() noexcept ( boolean-expression );
```

Noexcept-expression is a constant expression of type bool:

“Is this expression noexcept?”

```
constexpr bool plugh =  
    noexcept ( unevaluated-expression );
```

Requires-expression vs. clause

Requires-clause is part of a declaration:

“This participates in overload resolution when...”

```
template<class T>  
void xyzzy() requires boolean-expression ;
```

Requires-expression is a constant expression of type bool:

“Is this (set of) requirement(s) satisfied?”

```
constexpr bool plugh =  
    requires ( parameter-list ) { requirement-seq };
```

Noexcept-expression vs. clause

Noexcept-clause sometimes contains a noexcept-expression:

```
template<class T>  
void xyzzy() noexcept(noexcept( unevaluated-expression ));
```

Or we separate the two:

```
template<class T>  
bool is_nothrow_foable =  
    noexcept ( unevaluated-expression );
```

```
template<class T>  
void xyzzy() noexcept( is_nothrow_foable_v<T> );
```

Requires-expression vs. clause

Requires-clause sometimes contains a requires-expression:

```
template<class T>
void xyzzy()
    requires requires ( parameter-list ) { requirement-seq };
```

Or we separate the two:

```
template<class T>
concept Fooable =
    requires ( parameter-list ) { requirement-seq };
```

```
template<class T>
void xyzzy() requires Fooable<T>;
```

Requires-expression vs. clause

Requires-clause sometimes contains a requires-expression:

```
template<class T>
void xyzzy()
    requires requires ( parameter-list ) { requirement-seq };
```

Or we separate the two:

```
template<class T>
concept Fooable =
    requires ( parameter-list ) { requirement-seq };
```

```
template<class T>
void xyzzy() requires Fooable<T>;
```


Kinds of requirements

```
template<class T>
concept Fooable = requires ( parameter-list ) {

    typename type-expression ;

    { value-expression };

    { value-expression } noexcept ;

    { value-expression } -> type-or-concept ;

    requires constraint-expression ;

};
```

Kinds of requirements

```
template<class T>
concept Fooable = requires ( parameter-list ) {

    typename T::value_type;

    { value-expression };

    { value-expression } noexcept ;

    { value-expression } -> type-or-concept ;

    requires constraint-expression ;

};
```

Kinds of requirements

```
template<class T>
concept Fooable = requires (const T ct, int i) {

    typename T::value_type;

    { ct + i };

    { ct += i } noexcept;

    { ct += i } -> T&;

    { ct - ct } -> Integral;
};
```

Kinds of requirements

```
template<class T>
concept Fooable = requires (const T ct, int i) {

    typename T::value_type;

    { ct + i };

    { ct += i } noexcept;

    { ct += i } -> T&;

    { ct - ct } -> Integral;

};
```

Exactly equivalent to

```
requires noexcept(ct += i);
```

Kinds of requirements

```
template<class T>
concept Fooable = requires (const T ct, int i) {

    typename T::value_type;

    { ct + i };

    { ct += i } noexcept;

    { ct += i } -> T&;

    { ct - ct } -> Integral;

};
```

Exactly equivalent to

```
requires is_convertible_v<
    decltype(ct += i), T&
>;
```

Kinds of requirements

```
template<class T>
concept Fooable = requires (const T ct, int i) {

    typename T::value_type;

    { ct + i };

    { ct += i } noexcept;

    { ct += i } -> T&;

    { ct - ct } -> Integral;

};
```

Exactly equivalent to*

```
requires Integral<decltype(ct - ct)>;
```

Kinds of requirements

Sometimes, requirements can be chained together on a single line of code.

```
template<class T>
concept Fooable = requires (const T ct, int i, T::value_type) {

    { ct += i } noexcept -> T&;

    { ct - ct } noexcept -> Integral;

};
```

GCC (not C++2a) supports auto

```
template<class T>  
concept Fooable = requires (const T ct) {
```

```
    { *ct } -> auto&&;
```

```
    { +ct } -> auto*;
```

```
};
```

Equivalent to

```
requires !is_void_v<decltype(*ct)>;
```

Equivalent to

```
requires is_pointer_v<  
    remove_reference_t<decltype(+ct)>>;
```


Fun traps for the unwary

```
template<class T>
concept Negatable = requires (T t) {

    -t -> T;

};

static_assert(Negatable<char>); // FAILS
```



??

Fun traps for the unwary

```
template<class T>
concept Negatable = requires (T t) {

    -t -> T;

};

struct S {
    int T;
};

static_assert(Negatable<S*>);  // OK
```



??

Fun traps for the unwary



```
template<class T>  
concept Negatable = requires (T t) {
```

```
    -t -> T;
```

```
};
```

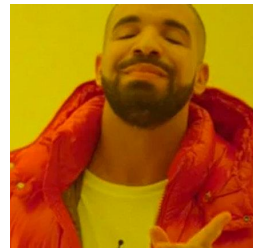
```
struct S {  
    int T;
```

```
};
```

```
static_assert(Negatable<S*>); // OK
```

Forgot the curly
braces!

Fun traps for the unwary



```
template<class T>
concept Negatable = requires (T t) {

    { -t } -> T;

};
```

Fun traps for the unwary

```
template<class T>
concept IntSized = requires {

    sizeof(T) == 4;

};

static_assert(IntSized<char>); // OK
```



??

Fun traps for the unwary

```
template<class T>  
concept IntSized = requires {
```

```
    sizeof(T) == 4;
```

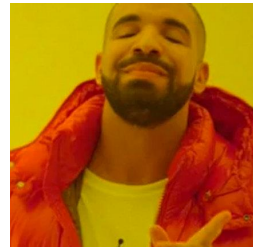
```
};
```

```
static_assert(IntSized<char>); // OK
```

Forgot the
“requires”
keyword!



Fun traps for the unwary



```
template<class T>
concept IntSized = requires {

    requires sizeof(T) == 4;

};
```

// Or even better...

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

Fun traps for the unwary

```
template<class T>
concept NothrowAddable = requires (T t) {

    noexcept ( t += t );

};

static_assert(NothrowAddable<int>);    // OK

static_assert(NothrowAddable<std::string>);    // OK
```



??

Fun traps for the unwary



```
template<class T>
concept NothrowAddable = requires (T t) {

    noexcept ( t += t );

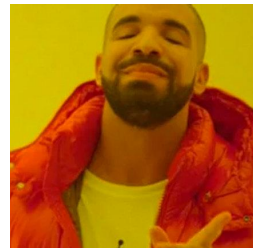
};
```

```
static_assert(NothrowAddable<int>); // OK
```

```
static_assert(NothrowAddable<std::string>); // OK
```

Forgot the
“requires”
keyword!

Fun traps for the unwary



```
template<class T>
concept NothrowAddable = requires (T t) {

    requires noexcept( t += t );

    // Or even better...

    { t += t } noexcept;

};
```


Almost done!

- High-level overview of C++17 TMP and C++2a Concepts
 - SFINAE by any other name [5–24]
- Subsumption and normal form [25–54]
- Syntax of a `requires-expression` [55–71]
 - Some traps for the unwary [72–81]
- Terse/abbreviated syntaxes and future directions [82–97]
 - “Abbreviated Function Templates” (“AFTs”)
 - Guidelines for using C++2a Concepts [92–94]
 - What C++2a Concepts still can’t do [95–97]

Terse syntax(es)

C++2a supports a terser syntax, but I currently recommend not to use it.

```
template<Integral T>  
void foo(T t) {  
    // ...  
}
```



“Constrained parameter” (July 2017)

means the same thing as

```
template<class T> requires Integral<T>  
void foo(T t) {  
    // ...  
}
```

Terse syntax(es)

C++2a supports a terser syntax, but I currently recommend not to use it.

```
template<class T> concept Reference = is_reference_v<T>;  
template<class T> concept Function = is_function_v<T>;
```

```
template<Reference R>  
void foo(R r) { ... }
```

```
template<Function F>  
void foo(F f) { ... }
```



“My attitude is never to be satisfied.” –Duke Ellington

Terse syntax(es)

Concepts TS supports an *even terser* syntax, commonly known as “*the* terse syntax.” This syntax was ***not*** adopted into C++2a.

```
void foo(Integral t) {    // Look ma, no “template”!  
    // ...  
}
```

would mean the same thing as


```
template<class T> requires Integral<T>  
void foo(T t) {  
    // ...  
}
```

However...

Terse syntax(es)

In November 2018, at San Diego, the Committee adopted this compromise approach to abbreviated function templates (“AFTs”).

```
void foo(Integral auto t) {    // look ma, no “template”!  
    // ...  
}
```



in the C++2a Working Draft means the same thing as

```
template<class T> requires Integral<T>  
void foo(T t) {  
    // ...  
}
```

Terse syntax(es)

The San Diego “AFT” compromise also extends *constrained return types* and variable type deduction. The Concepts TS supports this syntax minus the `auto`.

```
template<class T>
Integral auto foo(T i) { // if the deduced type is not Integral, static-assert
    Integral auto j = i; // if the deduced type is not Integral, static-assert
    return j;
}
```

Exactly analogous to C++11’s constrained return types:

```
template<class T>
auto* foo(T i) { // if the deduced type is not a pointer, static-assert
    auto* j = i; // if the deduced type is not a pointer, static-assert
    return j;
}
```


Non-type concepts

C++2a rejects concepts that are **not** templates:

```
concept True = true;    // error
```

But C++2a permits concepts that are **weird** templates:

```
template<template<class...> class Ctr>  
concept BigContainer = sizeof(Ctr<int>) >= 24;    // OK!  
static_assert(BigContainer<vector>);
```

```
template<int I>  
concept EvenValue = (I % 2 == 0);    // OK!  
static_assert(EvenValue<42>);
```

Non-type concepts

As of San Diego (November 2018), C++2a no longer lets you use “constrained parameter” syntax with non-type concepts.

```
template<int I>
concept EvenValue = (I % 2 == 0);  // OK!
static_assert(EvenValue<42>);
```

```
template<class T, int N> requires Object<T> && EvenValue<N>
struct EvenArray1 { ... };  // OK!
```

```
template<Object T, EvenValue N>
struct EvenArray2 { ... };  // ERROR
```

Multi-parameter concepts

C++2a permits concepts with multiple template parameters.

```
template<class A, class B, class C>  
concept BiggerThan =  
    sizeof(A) >= sizeof(B) && sizeof(B) >= sizeof(C);  
  
static_assert(BiggerThan<int[3], int[2], int[1]>);
```

Multi-parameter concepts

C++2a permits concepts with multiple template parameters.

```
template<class A, class B, class C>  
concept BiggerThan =  
    sizeof(A) >= sizeof(B) && sizeof(B) >= sizeof(C);
```

Combined with C++2a's constrained-parameter syntax, this gets weird.



```
template<BiggerThan<int[2], int[1]> D>  
void foo();
```

Equivalent to

```
template<class D> requires BiggerThan<D, int[2], int[1]>  
void foo();
```

Guidelines for naming concepts

Even though I say not to use constrained-parameter syntax, its existence suggests some style guidelines for naming of concepts:

- Your first template parameter is magic. Make sure your name reflects that.
 - `ConvertibleTo<Src, Dst>`  `ConvertibleTo<D>` `auto S`
 - `GeneratorOf<F, ReturnType>`  `GeneratorOf<R>` `auto F`
- But don't strain yourself. Maybe this concept just doesn't **need** to play well with constrained-parameter syntax.
 - `requires Mergeable<InIter1, InIter2, OutIter>`
 - Does it even need to be a concept at all?
 - `requires is_mergeable<I1, I2, O>`

Concepts enable subsumption. We'll never use `Mergeable` in a context requiring subsumption. So it needn't be a concept.

Guidelines for naming concepts

I strongly recommend to name non-type concepts with the `-Value` suffix. Non-type concepts should be extremely rare, anyway.

```
template<class T>  
concept Arithmetic = std::is_arithmetic_v<T>;
```

```
template<auto V>  
concept ArithmeticValue = Arithmetic<decltype(V)>;
```

Off-topic exercise for motivated students: Does `ArithmeticValue<T{}>` subsume `Arithmetic<T>`? Why or why not?

Guidelines for naming concepts



Probably the most important variable that your naming scheme needs to convey is whether your concept is “perfect-forwardable” or “type-trait-like.” **Avoid Concepts syntax for anything not “perfect-forwardable.”**

```
template<class T> concept Integral = is_integral_v<T>;
```

Type-trait-like. Bad.

```
void oops(Integral auto&& value) { ... }  
int y; oops(y); // does not compile; int& is not Integral
```

```
template<class T> concept Range = requires(T t) {  
    static_cast<T&&>(t).begin(); static_cast<T&&>(t).end();  
};
```

Perfect-forwardable.
A good fit for
C++2a Concepts.

```
void okay(Range auto&& value) { ... }  
vector<int> r; okay(r); // okay; vector<int>& is still a Range
```

Guidelines for naming concepts

“Avoid Concepts syntax for anything not perfect-forwardable”?

```
template<class T> concept Regular = ...;  
static_assert(Regular<int>);  
static_assert(not Regular<int&>);
```

Fundamentally
type-trait-like.

```
template<Regular T>  
class vector { ... }; // OK  
vector<int> v; // OK  
vector<int&> v; // “correctly” does not compile
```

A good fit for
C++2a Concepts in
some situations...

```
template<Regular T>  
void oops(Regular auto&& value); // “incorrect”  
int y; oops(y); // does not compile; int& is not Regular
```

...but not others.

Metaprogramming with concepts

Concepts can be nested inside namespaces.

Concepts **cannot** be nested inside classes or structs.

Concepts **cannot** be passed as template parameters.

```
namespace N { template<class T> concept NestedC = ...; } // OK
```

```
struct S { template<class T> concept MemberC = ...; } // ERROR
```

```
template <template<class T> concept ParamC> // ERROR
struct ST { ... };
```

Constrained non-template functions?

On the surface, constrained non-template functions seem to provide an alternative to `#ifdef / if constexpr`.

But neither Clang nor GCC supports this idiom. Don't try it.

```
unsigned byteswap(unsigned x) requires (sizeof(int) == 2) {  
    return (x << 8) | (x >> 8);  
}
```

```
unsigned byteswap(unsigned x)  
    requires (sizeof(int) == 4) {  
    return (x << 24) | (x >> 8 << 24 >> 8)  
        | (x >> 16 << 24 >> 16) | (x >> 24);  
}
```

Because name-mangling.
requires means “does not participate in overload resolution”; it doesn't mean “does not get codegenned.”

More information on Concepts

- “Constraints and concepts (since C++20)”
<https://en.cppreference.com/w/cpp/language/constraints>
- Cpplang Slack channel [#concepts](#)
<https://cpplang-inviter.cppalliance.org>
- For examples of usage in practice:
P0896 “The One Ranges Proposal” — *warning! 233-page PDF alert!*
<http://wg21.link/p0896>
- Herb Sutter’s P0745 “Concepts in-place syntax” <http://wg21.link/p0745>
- Corentin Jabot’s “The tightly-constrained design space of convenient syntaxes for generic programming” https://cor3ntin.github.io/posts/concepts_syntax/
- The compromise AFT paper: <http://wg21.link/p1141>



Questions?

Sidebar: Linkers hate return-type SFINAE

```
template<class T>
enable_if_t<is_integral_v<T>, T> foo(T t) { return t + 1; }
template<class T>
enable_if_t<not is_integral_v<T>, T> foo(T t) { return t + 2; }
```

} enable_if on the function's return type

```
template int foo<int>(int);           // _Z3fooIiENSt9enable_ifIX13is_integral_vIT_EES1_E4typeES1_
template short foo<short>(short);    // _Z3fooIsENSt9enable_ifIX13is_integral_vIT_EES1_E4typeES1_
template float foo<float>(float);     // _Z3fooIfENSt9enable_ifIXnt13is_integral_vIT_EES1_E4typeES1_
```



```
template<class T> requires is_integral_v<T>
T bar(T t) { return t + 1; }
template<class T> requires not is_integral_v<T>
T bar(T t) { return t + 2; }
```

} new-school requires

```
template int bar<int>(int);           // _Z3barIiET_S0_
template short bar<short>(short);    // _Z3barIsET_S0_
template float bar<float>(float);     // _Z3barIfET_S0_
```