

# Requirements and concepts

# Validity of templates

- ▶ Templates are checked for validity
  - ▶ On definition: syntactic correctness, correctness of independent names
    - Not required by the language specification but supported by rules
  - ▶ On instantiation: All rules of the language
- ▶ A template does not have to be correct for all combinations of arguments
  - It would be impossible in most cases
  - Compilers check the correctness only for the arguments used in an instantiation
    - Templates are difficult to test
  - Before C++20, there was no mechanism to specify requirements on template arguments
    - Trial-and-error approach (see SFINAE for advanced misuse)
    - Unreadable error messages when a template is incorrectly used
  - C++20 introduces *requires clauses* and *concepts* for constraining template arguments
    - They also assist in template function overload resolution (like SFINAE, unlike `static_assert`)
- ▶ Instantiation of a class template does not invoke instantiation of all members
  - A valid class template instance may contain invalid member functions
    - Example: copy-constructor of `vector<unique_ptr<T>>`

## ► A **requires-clause** acts as a constraint on template parameters

- Evaluated by the compiler in the moment of template instantiation

```
template< typename IT, typename F>
```

```
requires std::is_invocable_v<F, typename std::iterator_traits<IT>::reference>
```

```
F for_each( IT a, IT b, F f);
```

- In this case, the **requires** clause contains a constexpr bool expression

- [C++17] `std::is_invocable_v` is a variable template defined as

```
template< typename F, typename ... ArgTypes>
```

```
using is_invocable_v = is_invocable< F, ArgTypes...>::value;
```

- `std::is_invocable` is a class template defined to look like this:

```
template< typename F, typename ... ArgTypes> class is_invocable
```

```
{ static constexpr bool value = /*...*/; };
```

- the actual implementation uses partial specialization and other advanced tricks

- ▶ A **requires-clause** acts as a constraint on template parameters

```
template< typename IT, typename F>
```

```
requires std::is_invocable_v<F, typename std::iterator_traits<IT>::reference>
```

```
F for_each( IT a, IT b, F f);
```

- ▶ If violated, this function declaration will be ignored during overload resolution
  - Most likely, the result will be "no function declaration matches the call"
  - This indicates that the problem is not inside the implementation of `for_each`
- ▶ For non-function templates, the violation will directly trigger an error message
- ▶ The `requires` clause also acts as documentation
  - **Note:** The implementation of `for_each` probably contains the expression `f(*a)`
    - The `requires`-clause essentially checks whether this expression is correct
    - If not correct, template instantiation would fail even if the `requires` clause were not present
    - It would fail after overload resolution, not before (as with SFINAE or `requires`)
    - The error message would point to something inside the implementation

- ▶ A ***concept*** is, logically, a Boolean function whose arguments are types, templates or constants
  - ▶ In most cases, there is just one *typename* argument
  - ▶ Evaluated by the compiler
  - ▶ Note: C++14 already has a construct with the same underlying logic:

```
template< typename T> inline constexpr bool is_reference_v = /*...*/;
```

- The difference is in some syntactic sugar associated with concepts
- Concepts may be defined using bool constants but not (easily) the other way round

## ► Definition of a concept:

- A **concept** may be defined using a **requires-expression**

```
template< typename T> concept Dereferencable = requires (T x) { *x; };
```

- In this case, the **requires-expression** states that the expression `*x` must be semantically valid for any `x` of type `T`

```
template< typename F, typename ... AL> concept Callable  
= requires (F f, AL ... al) { f(al ...); };
```

- A **concept** may also be defined using other concepts or constant Boolean expressions, including combining by `&&` and `||` operators

```
template< typename T> concept Reference = std::is_reference_v<T>;
```

```
template< typename T> concept ConstReference =  
Reference<T> && std::is_const_v< std::remove_reference_t< T>>;
```

- In this context, `&&` and `||` operators are well-defined even for erroneous operands
  - If `remove_reference_t` is not defined for `T`, the result is false

# [C++20] Concepts

## ▶ Concepts used with all arguments explicit

### ▶ In the **requires-clause**

```
template< typename IT, typename F>
```

```
requires Iterator<IT> && Callable<F, typename IT::reference>
```

```
void for_each( IT a, IT b, F f);
```

### ▶ In the definition of other **concepts**

```
template< typename IT>
```

```
concept Iterator = Dereferenceable<IT> && Incrementable<IT>;
```

## ▶ Concepts used with the first argument implicitly inferred from the context

### ▶ Instead of **typename** in template parameter declaration

- The first argument of the concept is the type being declared here

```
template< Iterator IT, Callable<typename IT::reference> F>
```

```
void for_each( IT a, IT b, F f);
```

- Just a syntactic sugar equivalent to a **requires** clause

### ▶ In **auto** declarations

```
Iterator auto it = k.find(x);
```

- Triggers an error if the return type of find does not satisfy Iterator

```
[](Iterator auto it){ return *it; }
```

- Produces a **requires** clause in the generated template operator()

### ▶ In type-checking requirements inside a **requires-expression**

```
template< typename IT> concept SubtractableIterator =
```

```
requires (IT a, IT b) { {a-b} -> std::convertible_to<std::ptrdiff_t>; }
```

- Invokes the concept `std::convertible_to<decltype(a-b), std::ptrdiff_t>`

## ► Example

```
template< typename K, typename V> concept StackOf
requires (K k, V v) {
    {k.push(v)} -> std::same_as<void>;
    {k.top()} noexcept -> std::convertible_to< V>;
    {k.pop()} -> std::same_as<void>;
};

template< typename K> concept Stack
requires {
    typename K::value_type;
    requires StackOf<K, K::value_type>;
};
```



## ► Advantages of concepts

- Explicit and systematic statement of requirements
- Understandable diagnostic messages
- Requires clause participates in overload resolution (SFINAE no longer required)
  - Unlike a `static_assert` inside the template

## ► Adoption of concepts in standard library

- Previously existing parts of library are not upgraded to use concepts
- Some new parts like `std::ranges` are heavily dependent on concepts
- There are some generally usable concepts defined in `<concepts>`
  - Often equivalent to previously existing traits in `<type_traits>` etc.
  - Example: `std::same_as` does the same as `std::is_same_v`