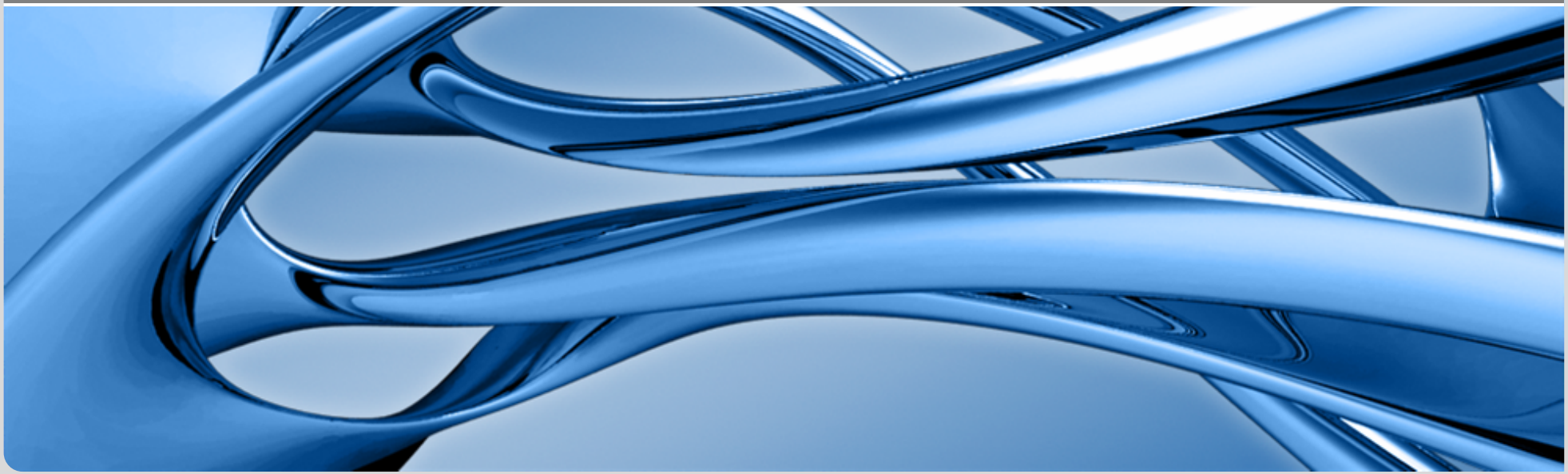
Karlsruher Institut für Technologie

Fortgeschrittene Templatemetaprogrammierung

Florian Weber



Definition von `std::enable_if`

```
template<bool, typename T = void> struct enable_if{};

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

template<bool B, typename T = void>
using enable_if_t = typename std::enable_if<B,T>::type;
```

Verwendungsmöglichkeit 1

```
template<typename T>
enable_if_t<pred<T>, returntype> fun(T arg) {
    ...
}
```

Verwendungsmöglichkeit 2

```
template<typename T>
returntype fun(T arg,
               enable_if_t<pred<T>, void*> = nullptr) {
    ...
}
```

Verwendungsmöglichkeit 3

```
template<
    typename T,
    typename = enable_if_t<pred<T>, void>
>
returntype fun(T arg) {
    ...
}
```

Verwendungsmöglichkeit 4

```
template<
    typename T,
    enable_if_t<pred<T>, int> = 0
>
returntype fun(T arg) { ... }
```

Verwendungsmöglichkeit 4

```
template<
    typename T,
    enable_if_t<pred<T>, int> = 0
>
returntype fun(T arg) { ... }
```

```
template<bool B>
using requires = enable_if_t<B, int>;

template<
    typename T,
    requires<pred<T>> = 0
>
returntype fun(T arg) { ... }
```

Makros

```
#define MYLIB_REQUIRES(...)\
    std::enable_if_t<(__VA_ARGS__), int> = 0


template<
    typename T,
    MYLIB_REQUIRES(pred<T>)
>
returntype fun(T arg) { ... }
```

Übersicht

- Möglichkeit 1: u.U. bei variadischen Funktionstemplates
- Möglichkeiten 2&3: nie
- Möglichkeit 4: sofern möglich

Übersicht

- Möglichkeit 1: u.U. bei variadischen Funktionstemplates
- Möglichkeiten 2&3: nie
- Möglichkeit 4: sofern möglich

- Makros sind widerlich und hier mittlerweile eigentlich unnötig

Übersicht

- Möglichkeit 1: u.U. bei variadischen Funktionstemplates
- Möglichkeiten 2&3: nie
- Möglichkeit 4: sofern möglich

- Makros sind widerlich und hier mittlerweile eigentlich unnötig

- Existierende Alternativen nach Möglichkeit nutzen
  - Tag-Typen
  - Concepts

Tag-Typen als Alternative

```cpp
template<typename Int>
Int sto(std::string_view s, std::true_type) {
    /* parse s as signed integer */
}

template<typename Int>
Int sto(std::string_view s, std::false_type) {
    /* parse s as unsigned integer */
}

template<typename Int>
Int sto(std::string_view s) {
    return sto<Int>(s, std::is_signed_v<Int>);
}
```

Einsatz bei Iteratoren

```
template<typename It>
constexpr bool is_ra_it = std::is_base_of<
    std::random_access_iterator_tag,
    typename std::iterator_traits<It>::iterator_category
>::value;

template<typename It, requires<is_ra_it<It>> = 0>
void some_fun(It) { /* version for random-access-iterators */ }


template<typename It,
    requires<is_in_it<It> = 0,
    requires<!is_ra_it<It>> = 0>
void some_fun(It) { /* version for other input-iterators */ }
```

Einsatz bei Iteratoren

```
template<typename It>
void some_fun(It, str::random_access_iterator) {
    /* version for random-access-iterators */
}

template<typename It>
void some_fun(It, std::input_iterator) {
    /* version for other input-iterators */
}

template<typename It>
void some_fun(It it) {
    some_fun(it,
        typename std::iterator_traits<It>::iterator_category{});
}
```

Kombination mit Enums

```
class enum bla { foo, bar, baz };

template<typename T>
constexpr blub() {
    if (...) {return bla::foo;}
    ...
}

template<bla B>
struct bla_t {};

using foo_t = bla_t<bla::foo>;

void fun(..., foo_t) { ... }

fun(..., bla_t<blub<...>()>);
```

Concepts

- nach Jahrzehnten immer noch nicht so richtig fertig.

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

template<typename T> requires EqualityComparable<T>
void f(T arg) { ... }

template<EqualityComparable T>
void f(T arg) { ... }

void f(EqualityComparable arg) { ... }
```