# 1 Traits- und Tag Dispatching

## Traits

Wir haben bereits gelernt, was Traits sind, möchte aber trotzdem nochmal daran erinnern, wofür man diese nutzen kann. Im Allgemeinen kann man sich Traits als Interface für Typen vorstellen. Das bedeutet, sie sind Klassen oder Klassen Templates, die einen Typen charakterisieren und dazu verwendet werden, Templates genauer zu spezifizieren. Die folgende Funktion kann erst ein mal mit jedem Argument aufgerufen werden, da `T` nicht genauer spezifiziert ist:

```
template <class T>
int getValue(T arg) {
    return arg.value;
}
```

Geht etwas in der Funktion schief (also insbesondere, wenn `arg.value` nicht definiert ist oder nicht zum Typ `int` gecastet werden kann), dann sind die Fehlermeldungen meist undurchdringlich und weiteres Debugging sehr zeitaufwendig. Aus diesem Grund nutzt man Traits, um schon zur Compilezeit Templates strenger zu typisieren und damit Fehler vorzubeugen. Ein Trait könnt also folgendermaßen implementiert werden:

```
#include <iostream>

struct A { int value; };
struct B { char value; };
struct C { int data; };

template <typename T>   struct has_value {
    static constexpr bool value = false;
};
template <>             struct has_value<A> {
    static constexpr bool value = true;
};
template <>             struct has_value<B> {
    static constexpr bool value = true;
};

int main() {
    std::cout << has_value<A>::value << std::endl; // 1
    std::cout << has_value<B>::value << std::endl; // 1
    std::cout << has_value<C>::value << std::endl; // 0
}
```

Diese Art von Implementierung verstößt jedoch gegen das SOLID Prinzip, denn wenn wir eine neue Klasse `D` hinzufügen, müssen wir daran denken auch die Spezifikation vom `has_value` Trait zu erweitern. Mit den in C++ 20 eingeführten Concepts können wir das Problem lösen, indem wir Typ-Überprüfungen kombinieren und elegant zusammenschreiben. Nun können wir ohne weiteres neue Klassen erzeugen, ohne uns Gedanken um Traits bzw. Constraints zu machen.

```
#include <concepts>
```

```
#include <iostream>
#include <type_traits>

struct A { int value; };
struct B { char value; };
struct C { int data; };

template <typename T>
concept has_value = requires(T t) { { t.value }; };

int main() {
    std::cout << has_value<A> << std::endl; // 1
    std::cout << has_value<B> << std::endl; // 1
    std::cout << has_value<C> << std::endl; // 0
}
```

## Tag Dispatching

Um verschiedene Verhalten eines Konstruktors zu implementieren, haben wir die Möglichkeit diesen zu Überladen, um Signaturen voneinander unterscheiden zu können. Da Konstruktoren keinen Namen haben, kann der Compiler Konstruktoren mit gleichen Signaturen jedoch nicht unterscheiden. Das Problem wird bei der Erzeugung eines Objekts vom Typ `Kreis` klar:

```
struct Kreis {
    double radius;
    Kreis(double radius) : radius(radius){};
    Kreis(double durchmesser) : radius(durchmesser/2){};
    // error: ‘Kreis::Kreis(double)’ cannot be overloaded
    //         with ‘Kreis::Kreis(double)’
};
int main {
    Kreis c(5); // Woher soll der Compiler wissen, was wir meinen?
}
```

Eine Lösung für dieses Problem ist das Tag Dispatching. Dabei wird die Signatur so verändert, dass sie eine zusätzliche leere Struktur (den Tag) beinhaltet, sodass der Compiler den richtigen Konstruktor wählt. Dieses Prinzip haben wir bereits im SDP "Tag Dispatching" kennengelernt. Den Konstruktor des Kreises könnten wir also mittels zwei Tags implementieren:

```
struct radius_tag {};
struct durchmesser_tag {};

struct Kreis {
    double radius;
    Kreis(double radius, radius_tag _) : radius(radius){};
    Kreis(double durchmesser, durchmesser_tag _) : radius(durchmesser / 2){};
};

int main() {
```

```
    Kreis k1(3, radius_tag{});
    Kreis k2(6, durchmesser_tag{});
};
```

Nun stoßen wir auf ein weiteres Problem: wir müssen diese Tags explizit angeben, um das Verhalten des Konstruktors beeinflussen zu können. Außerdem tragen die Tags keine Information, sondern beeinflussen nur das Verhalten von Funktionen, weswegen Code mit vielen Tag Definitionen schnell unübersichtlich wird und verwirren kann.

## Trait Dispatching

Die Lösung des Problems ist das Trait Dispatching. Statt leere Strukturen zu definieren, nutzen wir Traits, um Signaturen so zu typisieren, dass der Compiler automatisch die richtige Implementation wählt.

### Ein Beispiel

Wir haben einen Algorithmus, der für jede Klasse, welche ein Attribut `value` besitzt, definiert ist. Der Algorithmus rechnet mit Doubles, weswegen wir auch nur Attribute erlauben, die sich zu Double caste lassen. Zusätzlich gibt es einen optimierten Algorithmus, der jedoch nur für Klassen funktioniert, die über ein zusätzliches Attribut `value2` verfügen. Nun möchten wir natürlich nicht bei jeden Aufruf des Algorithmus selber überprüfen, ob wir die optimierte Version statt der normalen Version wählen können. Wir nutzen Trait Dispatching und definieren zwei Traits. Der `has_value` Trait ist wahr für alle Klassen, die ein `value` Attribut besitzen, welches sich zu einem Double casten lässt. Der `is_optimizable` Trait ist wahr für alle Klassen, die das zusätzliche Attribut `value2` besitzen. Nun können wir die Funktion `algorithm` so implementieren, dass sie mit allen Klassen aufgerufen werden kann, die `has_value` erfüllen. Der Compiler wird dann automatisch den optimierten Algorithmus aufrufen, sofern dies möglich ist.

```
#include <concepts>
#include <iostream>
#include <type_traits>
#include <typeinfo>

struct A { int value; };
struct B { double value; };
struct C { char value; int value2; };

template <typename T>
concept has_value = requires(T t) {
                        { t.value } -> std::convertible_to<double>;
                    };

template <typename T>
concept is_optimizable = requires(T t) {
                             { t.value2 } -> std::convertible_to<double>;
                         };

template <has_value T>
```

```
void algorithm(const T& u) {
    if constexpr (is_optimizable<T>) {
        std::cout << typeid(T).name() << " -> optimiert" << std::endl;
        // ...
    } else {
        std::cout << typeid(T).name() << " -> unoptimiert" << std::endl;
        // ...
    }
}

int main() {
    A a(1);
    B b(2.);
    C c('a', 4);

    algorithm(a);  // A -> unoptimiert
    algorithm(b);  // B -> unoptimiert
    algorithm(c);  // C -> optimiert
}
```

Wir sehen sofort die Stärke des Trait Dispatching: als Benutzer der Funktion `algorithm` müssen wir uns keine Gedanken machen, ob wir die optimierte Version davon aufrufen können. Der Compiler wird das automatisch für uns entscheiden. Zusätzlich haben wir eine sehr starke Typsicherheit, da der Compiler einen nachvollziehbaren Fehler wirft, wenn wir versuchen, die Funktion mit Parametern aufzurufen, welche unsere Constraints nicht erfüllen:

```
    algorithm(10);  // error: required for the satisfaction
                    //        of ‘has_value<T>’ [with T = int]
```

## Wann sollte man Trait-Dispatching nutzen?

Generell gilt die Regel, dass man Trait-Dispatching nur nutzt, um zusätzliche Informationen über das Verhalten einer Funktion zu liefern und nicht zusätzliche Information über Daten. Letzteres sollte man immer mit strengen Typen implementieren.