Typ-Templates leicht gemacht

Typ-Templates sind der andere Baustein für das **„generische Programmieren“.** Man kann auch eine Typ-Definition mit Platzhaltertypen und Platzhalter-Konstanten versehen:

```
template<typename T>                    // z.B. Bruch<int>
struct Bruch { …};                      //      Bruch<long long>

template<typename T, int Dimension>     // 3d: Vec<double, 3>
class Vec { … }                         // 2d: Vec<float, 2>
```

Solche Typ-Templates haben wir bereits mit der STL verwendet, z.B. `std::vector<int>`

Im Unterschied zu Funktionstemplates muss man die Platzhalter-Typen meistens angeben. Seit C++-17 kann C++ bei einer **Instanzierung mit Initialisierer** den Typ oft aus dem Initialisierer bestimmen:

```
std::array  x{1, 2, 3, 4};    // -> std::array<int, 4>
std::pair p{1., "hi"};        // -> pair<double, const char*>
std::scoped_lock lg{mtx}      // -> scoped_lock<mutex>
```

In der Template-Typdefinition kann man die Platzhalternamen wie normale Typen oder Konstante verwenden. **Wenn man alle Methoden und Freunde des Template-Typen innerhalb der Typdefinition** erstellt, unterscheidet sich die Template-Programmierung nicht von einem Nicht-Template. Als kleines Beispiel: Wir wandeln unseren `Bruch` um in ein Template `Bruch_t<T>`.

1) Wir schreiben vor die Definition (und auch vor die GGT-Funktion) die Template-Klausel `template<typename T>`
2) Wir benennen alle Vorkommnisse von `Bruch` in `Bruch_t` um.
3) Wir benennen alle Vorkommnisse von `int` in `T` um.
4) Wenn man will: Man definiert den Typ `Bruch` kompatibel zur alten Definition, sodass man alte Programme unverändert mit dem neuen Header kompilieren kann: `using Bruch = Bruch_t<int>;`

```
Bruch_t{5'000'000'000, 700'000'000'000}// -> Fraction<long long>
Bruch_t{5'000'000'000, 1'000'000'000}  // Fehler : long long und int
Bruch_t<long long>{5'000'000'000, 1'000'000'000} // ok,  da explizit
Bruch_t {'a', 'c'}  // -> Bruch_t<char>  + Warnungen oder Fehler (narrowing conversion von int nach char )
```