Funktions-Templates vor C++20

Die klassischen Templates gehören schon seit ca. 1990 zum C++-Sprachumfang und erlauben seither die Programmierung generischer Funktionen und Datentypen. Diese greifen auf denselben Code zurück, wenden diesen aber auf veränderbare Datentypen an. Der Programmierer erstellt nur einen Bauplan für eine Funktion (= **Funktions-Template**) oder für eine Struktur/Klasse (= Klassen-Template) und der Compiler generiert daraus bei Bedarf die benötigten speziellen Varianten (= **Instanzierungen).** Vor C++20 **musste jedes Template** durch eine Template-Klausel eingeleitet werden, z.B.

```
template<typename T>        // Template-Argument  T  ist Typplatzhalter
T my_max(T x, T y)          // Funktions-Template: Argumente  x, y vom Typ T
{    return x < y? y : x; }

template<typename T, int Dim> // Typ  T  und  int Dim  sind Platzhalter
struct Vec {                  // Class-Template  eines 2D, 3D, nD Vektors
{    std::array<T, Dim> coords; … }
```

```
my_max(3, 5);                   // ok, ergibt 5
my_max(3.1, 5.7);               // ok, ergibt 5.7
Vec<double, 2> p{1., 2.};       // 2D-Vektor mit double Koordinaten
```

Mit dem Schlüsselwort `template` wird der „Bauplan“ eingeleitet, in den Spitzklammern stehen die Platzhalter, die in diesem Template variiert werden (Template-Argumente), meistens Platzhalter für variable Typen (es sind aber auch Integerkonstante möglich). Im Code der Templates können diese Platzhalter wie echte Typen und echte Integer verwendet werden. Das Funktions-Template besitzt also sowohl Template-Argumente (in den Spitzklammern) als auch Funktions-Argumente (in den runden Klammern).

Bei der Verwendung eines Templates muss der Compiler herausfinden, welchen konkreten Typ er statt der Platzhaltertypen (meist `T`) verwenden soll. Das kann auf 2 Arten geschehen:

1) man gibt alle oder nur die ersten Template-Argumente in Spitzklammern explizit an: explizite Instanzierung. Das ist bei Funktions-Templates eher unüblich, bei Class-Templates eher die Norm):

   ```
   my_max<int>(3.1, 5.7);          // ok, ergibt 5
   my_max<double>(3.1, 5.7);       // ok, ergibt 5.7
   ```

2) vom Compiler automatisch erkennen lassen (=**TAD Template Argument Deduction** bzw. **CTAD = Class Template Argument Deduction**): Der Compiler versucht aus den Funktionsargumenten **des Aufrufs** die Template-Platzhalter zu erkennen. Der Template-Aufruf sieht in diesem Fall wie ein normaler Funktionsaufruf aus:

my_max(3, 5) : x hat den Typ T in der Definition, im Aufruf steht dafür 3 (int), also ist T = int. y hat den Typ T in der Definition, im Aufruf steht dafür 5 (int), also ist T = int. Da kein Widerspruch existiert, instanziert (erzeugt) der Compiler die folgende Funktion (er ersetzt dabei jedes T durch int):

```
int my_max<int>(int x, int y) { return x < y? y : x; }
```

my_max(3.1, 5.7) : Der Compiler erkennt analog T = double und instanziert :

```
double my_max<double>(double x, double y) {…}
```

Aus einem Funktionstemplate können daher gar keine oder sehr viele Instanzierungen entstehen, je nachdem, wie oft es mit verschiedenen Typen verwendet wird. Es handelt sich hier um **kein Überladen** der Funktion my_max(), sondern es entstehen tatsächlich unterschiedliche Funktionsnamen my_max<int>, my_max<double> etc. Die Teile in Spitzklammern gehören zum Funktionsnamen! Allerdings darf nur der Template-Compiler Funktionen mit solchen Funktionsnamen benennen (nicht der Programmierer!).

my_max(2, 5.7) x hat den Typ T in der Definition, im Aufruf steht dafür 2 (int), also ist **T = int.** y hat den Typ T in der Definition, im Aufruf steht dafür 5.7, also ist **T = double**: **Substitutionsfehler!**

Bei Substitutionsfehlern kann der Compiler nicht widerspruchsfrei erkennen, welchen Typ er für T einsetzen soll. Es findet bei TAD **KEINE Typumwandlung** statt! Sobald der Compiler auf Widersprüche hinsichtlich des Platzhaltertyps stößt, ignoriert er diese Template-Definition, erzeugt aber keine Fehlermeldung (SFINAE: Substitution Failure Is Not An Error). Auch kleinste Unterschiede reichen für einen Widerspruch aus z.B. const und nicht const.

Zu einer (oder eher sehr vielen) Fehlermeldung(en) kommt es erst, wenn der Compiler anschließend keine andere Möglichkeit findet, diesen Funktionsaufruf zu erfüllen (z.B. durch ein anderes Funktions-Template my_max() oder eine andere Funktion my_max()) .

Wo werden Templates definiert?

Der C++-Compiler benötigt **immer den kompletten Quellcode** des Templates, um die Instanzierungen zu erstellen. Deshalb muss dieser Code in **jeder .cpp** Datei des Projekts vorhanden sein, in der das Template verwendet wird. Ist das mehr als eine Datei, so schreibt man am besten die gesamte Template-Definition in eine Headerdatei und inkludiert diese.