C++20 Concepts und Funktions-Templates

Wenn man ein Funktionstemplate mit unpassenden Typen instanziert, dann bekommt man oft den Zorn des Compilers mit voller Wucht zu spüren, der Hunderte Fehlermeldungen produziert. Oder noch schlimmer: die Funktion liefert unsinnige Resultate.
Lösung: Man sagt dem Compiler, welche Datentypen erlaubt sind und welche nicht. C++20 definiert einige benannte Anforderungen (constraints) in der Datei `<concepts>`:

```
std::integral            alle Integertypen
std::signed_integral     alle Integertypen mit Vorzeichen
std::unsigned_integral   alle Integertypen ohne Vorzeichen
std::floating_point      float, double, long double
std::convertible
std::invocable
…
```

Gleichzeitig erhält man mit C++20 die Möglichkeit, eigene Concepts zu entwickeln und so die Instanzierung der Lambdas genau zu regeln. Für die Einschränkung der Instanzierung wurden 2 verschiedene Möglichkeiten geschaffen, die man auch kombinieren kann:

1) Dem `auto` ein Concept voranstellen: z.B. `std::integral auto`
2) Vor dem Funktionsrumpf eine **`requires`** Klausel einfügen, die noch allgemeinere Einschränkungen zulässt (Ausdrücke vom Typ `bool`, die vom Compiler evaluiert werden können)

Das Template vom vorigen Text kann man z.B. so auf gleiche Typen einschränken:

```
const auto& my_max(auto x, auto y)     // Template a la C++20
     requires std::same_as<decltype(x), decltype(y)>
{    return x < y ? y : x;  }
```

Den Typ eines Objekts erhält man mittels `decltype()` (seit C++11), und wir verlangen hier, dass beide Parameter denselben Typ haben. In diesem Fall können wir sogar eine konstante Referenz zurückgeben, was bei ungleichen Typen nicht geht.

Dadurch lassen alle 3 Problemfälle des letzten Textes verhindern, weil sich dieses Template nicht darauf anwenden lässt. Bei Bibliotheksfunktionen, die in vielen Anwendungen aufgerufen werden, ist diese Korrektur unbedingt sinnvoll. Schreibt man die Funktion nur für eine eigene Anwendung, wird man eher darauf verzichten, weil man ja über dieses Problem (hoffentlich) Bescheid weiß.

Templates lassen sperrige Typen verschwinden

Für eine Implementierung des Newtonverfahrens durch ein(e) Funktion(stemplate) `newton(f, x)` muss man eine reelle Funktion `f` und eine reelle Zahl `x` an diese Funktion übergeben. Aber wie drückt man dies am besten aus? Man kann reelle Zahlen durch `double` nachbilden, aber welchen Typ hat dann eine reellwertige Funktion? Lösung: Wir sagen gar nicht, was das ist und schreiben:

```
auto newton(auto f, double x)
```

Das erlaubt aber auch Aufrufe wie `newton(1, 2)` (1 ist keine Funktion) oder `newton(1., exp)` (Parameter vertauscht!). Mit den folgenden Fehlermeldungen wird ein Anwender viel Freude haben. Mit Concepts können wir unsere Absicht klar ausdrücken:

```
auto newton(auto f, std::floating_point auto x)
     requires std::invocable<decltype(f), decltype(x)>
{...}
```

Interpretation: `x` muss einen Gleitkommatyp haben und `f(x)` muss ein korrekter C++ Ausdruck sein. d.h. wir legen uns nicht fest, was `f` ist, sondern sagen, was man damit machen kann.

Alle Concepts beziehen sich immer auf die **Typen** der Parameter, nicht auf die Parameterobjekte selbst. Mit `decltype()` kommt man vom Objekt zu seinem Typ.

Damit wird obiges Template extrem flexibel, weil man nicht nur C++-Funktionen, sondern auch Lambdas oder sonstige Callables für `f` übergeben kann. Und alles nur, weil man nicht den Typ einer reellwertigen Funktion hinschreiben konnte oder wollte. Wenn man sich bei `x` auf `double` beschränken will geht sogar noch kürzer ohne `requires`:

```
auto newton(std::invocable<double> auto f, double x)
{...}
```

Wer nur schnell etwas für sich programmieren will, wird die Concepts eher weglassen. Die korrekte Antwort auf die Frage in der Aufgabenstellung ist übrigens:

```
double newton(double (*f)(double), double x)
```

erlaubt nur eine reellwertige Funktion als 1. Parameter.