Function Templates vs. Function Overloading

Beispiel-Aufgabe: Finde die größere von 2 Zahlen (`int, long, double, Bruch, …`)
ziemlich trivialer Algorithmus: Vergleiche die 2 Zahlen und gib die größere zurück

In Python ist die Lösung simpel, da jede Funktion mit unterschiedlichen Datentypen aufgerufen werden kann und unterschiedliche Datentypen als Ergebnis haben kann:

```
def my_max(x, y):          # x, y können einen beliebigen Typ haben
     if x < y:
          return y;
     else:
          return x;

mx = my_max(1, 3)          # my_max(int, int) -> 3  int
mx = my_max(1, 3.7)        # my_max(int, float) -> 3.7   float
```

In C++ ist das unmöglich, da jede Funktion **ihre Argumenttypen und ihren Rückgabetyp** eindeutig deklariert und nur diese Typen akzeptiert (abweichende Argumente werden immer in diesen Typ umgewandelt). Schreibt man in C++ eine analoge Funktion `my_max()`, muss man den Typ der Argumente angeben, z.B.:

```
int my_max(int x, int y)              // Maximum von 2 int
{    return x < y ? y : x; }          // ternärer Fragezeichen-Operator
...
my_max(3, 5);                  // ok, ergibt 5
my_max(3.1, 5.7);              // ok, ergibt 5 ☹
my_max(2, 3'000'000'000);      // ok, ergibt 2 ☹☹☹
```

Der 2. Aufruf ist kompatibel und der Compiler wandelt die `double` beim Aufruf in `int` um und das Resultat ist ein `int`! Beim 3. Aufruf muss und wird C++ den `long long` in einen `int` konvertieren und dadurch einen negativen Wert erzeugen, wodurch das falsche Resultat zustande kommt.

Möchte man das Maximum von zwei `double` korrekt bestimmen, muss man eine neue Funktion ZUSÄTZLICH schreiben. In C++ darf (**soll!)** man denselben Namen dafür verwenden (=Überladen, Overloading):

```
double my_max(double x, double y)  // Überladen von my_max()
{    return x < y ? y : x; }       // der genau gleiche Code
```

Jetzt funktioniert der Vergleich und 2 korrekt, der 3. Vergleich erzeugt jedoch einen Compiler-Fehler, da er nicht weiß, welche der 2 Varianten er für `long long` nehmen soll. Also muss auch noch eine 3. Variante für `long long` her und vielleicht noch für `char, float, long, unsigned` …

Wäre es nicht schöner, wenn wir nur den Algorithmus programmieren müssten, der in allen Versionen gleichbleibt, und C++ erzeugt die einzelnen Funktionen daraus selbst? Das ist in Wahrheit schon lange möglich, z.B. ab C++14 durch ein **generisches Lambda:**

```
auto my_max = [](auto x, auto y)   { return x < y ? y : x; };

my_max(3, 5);                      // ok, ergibt 5
my_max(3.1, 5.7);                  // ok, ergibt 5.7 😊
my_max(2, 3'000'000'000);          // ok, ergibt 3'000'000'000 😊
```

Seit C++20 darf man mit der genau gleichen Syntax Funktionstemplates erstellen, die bei Lambdas erlaubt ist. Wenig überraschend waren generische Lambdas immer schon Funktionstemplates und diese Neuerung beseitigt eigentlich nur eine willkürliche Beschränkung, einen logischen Grund gab es dafür nie:

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

Merke: Enthält etwas, was wie eine Funktion aussieht, **wenigstens einmal `auto`** in einem **Parametertyp,** dann ist es in Wirklichkeit ein Funktionstemplate und akzeptiert für diesen Parameter jeden Typ. Der Compiler erzeugt aus dem Template für jeden Aufruf des Templates eine eigene Funktionsinstanz, in der `auto` durch den wirklichen Typ ersetzt wird:

```
my_max(3, 5)        -> my_max(const int&, const int&)
my_max(3.1, 1LL)    -> my_max(const double&, const long long&)
```

Jetzt hat man allenfalls ein Problem, wenn das Template mit unterschiedlichen Typen aufgerufen wird:

```
my_max(3, 2LL);                    // 3LL ☹
my_max(1., 2LL);                   // 2.  ☹
my_max(-1, 1U) ;                   // 4294967295 ☹
```

Bei allen 3 Aufrufen hat das Ergebnis den richtigen Wert, aber den falschen Typ!