Funktionsdefinition, -deklaration, -signatur

Die Funktionsdefinition:

[Zusatz vorn] Ergebnistyp Funktionsname(Parameter 1, Parameter 2, ...) [Zusatz hinten]
// kein Strichpunkt!
{
    Anweisung 1; // Anweisungen enden mit Strichpunkt
    Anweisung 2;
    ...
}

Jeder Parameter besteht aus einer Typangabe und einem Namen (und in C++ optional einem Defaultwert).

Funktionssignatur (Function Signature): der Teil **vor den { } ohne die Parameternamen**.
Funktionsrumpf (Function Body): der Teil in den { }
Funktionsdefinition (Function Definition): das Ganze
Funktionsdeklaration (Function Declaration) = Funktionssignatur **Strichpunkt**

Beispiel Funktionsdefinition:

```
[[nodiscard]] int sum(const int& a, int b = 1)  noexcept
{    return a+b; }
```

Zusatz vorn:	`[[nodiscard]]`
Ergebnistyp:	`int`
Funktionsname:	`sum`
Parameter 1 Typ:	`const int&`
Name:	`a`
Parameter 2 Typ:	`int`
Name:	`b`
Defaultwert:	`1`
Zusatz hinten:	`noexcept`

Funktionssignatur:
```
[[nodiscard]] int sum(const int&, int = 1)  noexcept
```

Funktionsdeklaration:
```
[[nodiscard]] int sum(const int&, int = 1)  noexcept;
```

Eine Funktionsdefinition (oder eine Funktionsdeklaration) sagt dem C++ Compiler, wie er diese Funktion aufrufen kann (wie viele Argumente muss er übergeben, welchen Typ haben diese, was gibt die Funktion zurück, was muss er noch beachten: Zusätze). Der Compiler ruft eine Funktion nur auf, wenn er **VORHER** die Definition oder eine Deklaration der Funktion gesehen hat. Die C++ Headerdateien enthalten hauptsächlich die Deklarationen der Bibliotheksfunktionen und manchmal auch deren Definitionen (d.h. auch den Code).

Eine Funktion muss in einem C++ Programm **genau einmal** komplett definiert werden (ODR = One Definition Rule), darf aber beliebig oft konsistent deklariert werden.

Der Compiler prüft beim Aufruf von Funktionen (d.h. der Name der Funktion gefolgt vom Aufruf-Operator `()` steht irgendwo im Programm), ob die richtige Anzahl von Argumenten und die richtigen Typen vorhanden sind (Parameter mit Defaultwert dürfen wie in Python beim Aufruf fehlen):

```
sum()                 // Fehler: kein Argument vorhanden
sum(5)                // korrekt: das Ergebnis ist 6 (5+1)
sum(1, 2)             // korrekt: das Ergebnis ist 3 (1+2)
sum(1.5, 2.1)         // korrekt: Wert ist 3 (1+2) (1.5 -> 1, 2.1 -> 2)
sum("1", 2)           // Fehler: "1" kann nicht konvertiert werden
```

Man sieht hier, dass C und C++ beim Funktionsaufruf automatisch Typumwandlungen vornehmen können, wenn solche existieren, z.B. `double -> int, double -> const int&`. Es wird immer das Argument des Aufrufers in den Typ des Parameters der Funktion gewandelt. Die **Funktion** (nicht der Aufrufer) bestimmt, welche Argument-Typen verwendet werden.

Die Parameter der Funktionsdefinition (hier `a,b`) stehen innerhalb der Funktion wie lokale Variable zur Verfügung. Ihre Anfangswerte erhalten sie vom Aufrufer der Funktion direkt beim Aufruf (entweder als Kopie oder als Referenz etc., siehe später!).