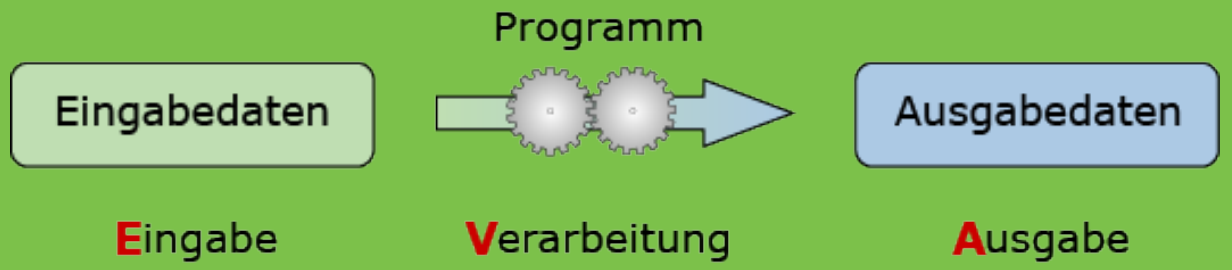
Hochschule Fulda
University of Applied Sciences

Programmierung 1 – Funktionen in C



Yvonne Jung

Wiederholung

Hochschule Fulda
University of Applied Sciences

- Bisher haben wir einige Funktionen kennengelernt und verwendet
  - Z.B. Funktionen für Ein- u. Ausgabe und insbesondere die *main()* Funktion
- *main()* kommt genau einmal im C-Programm vor und kann als sog. "Hauptprogramm" betrachtet werden
  - Aufbau:

Zähler für Kommdozeilen-Argumente

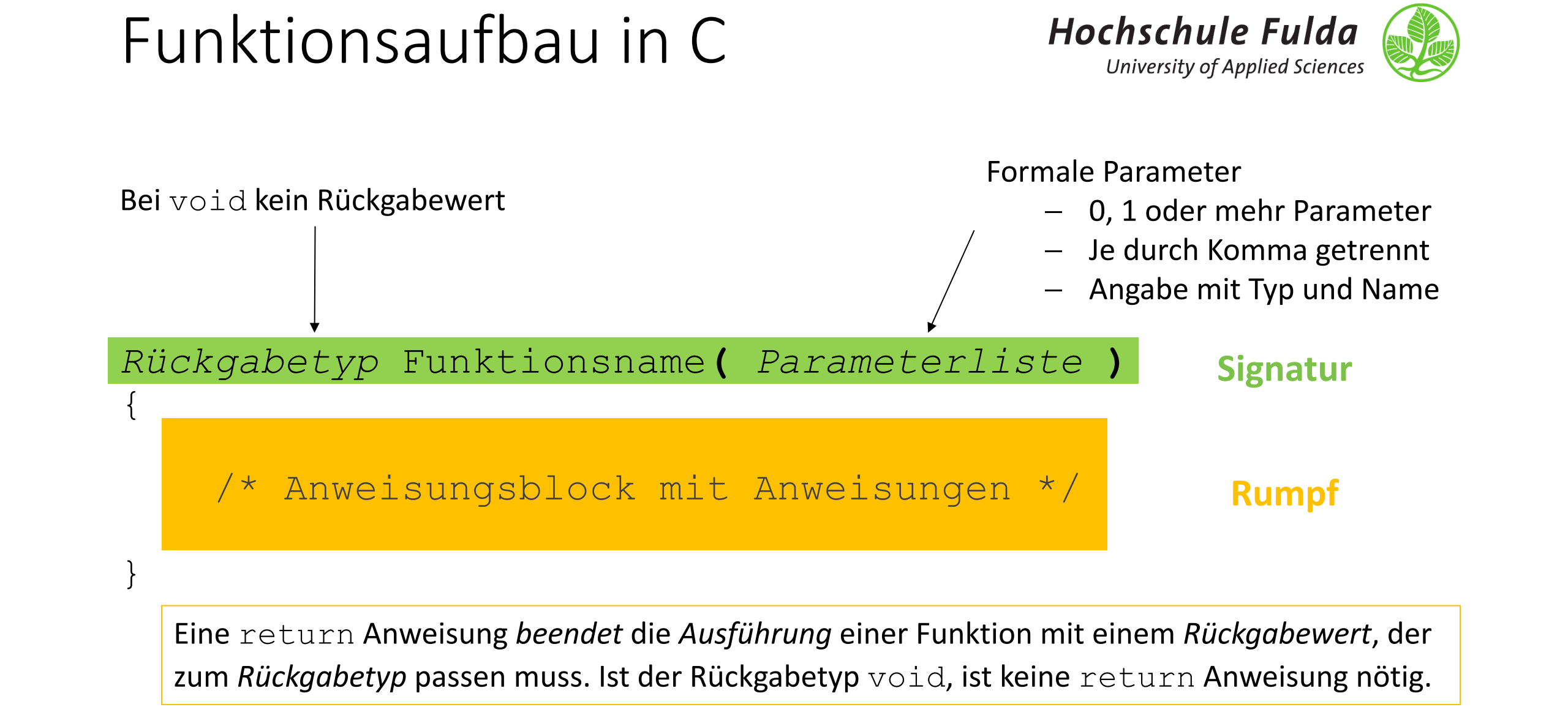
Parameter als String interpretiert
(Feld mit Zeigern auf „C-Strings“)

```
int main(int argc, char* argv[])
{
    return 0;  //Kann nach C99 entfallen
}
```

  - Parameter *argc* und *argv* können weggelassen werden, wenn nicht benötigt

Funktionsaufbau in C

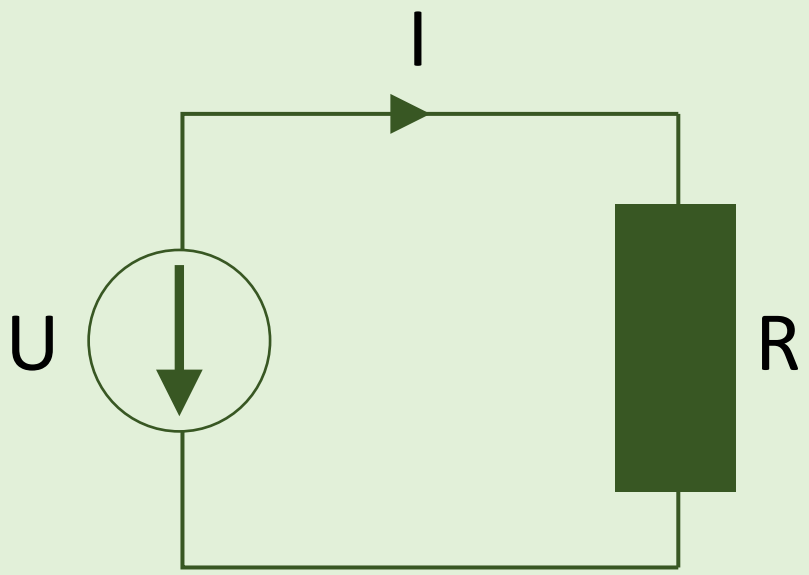
Hochschule Fulda
University of Applied Sciences



Eine `return` Anweisung *beendet* die *Ausführung* einer Funktion mit einem *Rückgabewert*, der zum *Rückgabetyp* passen muss. Ist der Rückgabetyp `void`, ist keine `return` Anweisung nötig.

Bsp.: Ohmsches Gesetz

Hochschule Fulda
University of Applied Sciences



$$R = \frac{U}{I} \Rightarrow U = R \cdot I$$

Aufgabe:

Implementieren Sie in C ein Programm, welches für einen einzugebenden Strom *I* und Widerstand *R* die Spannung *U* berechnet und ausgibt.

Formulieren Sie den Algorithmus zunächst pseudocodemäßig!

Strukturierte Programmierung

Hochschule Fulda
University of Applied Sciences

- Schrittweises Verfeinern (→ Top-Down)
  - Ziel: Programmiersprache sollte uns beim Aufschreiben (komplexerer) Lösungen entgegenkommen
  - Lösungsansatz zu vorigem Beispiel

Pseudocode:

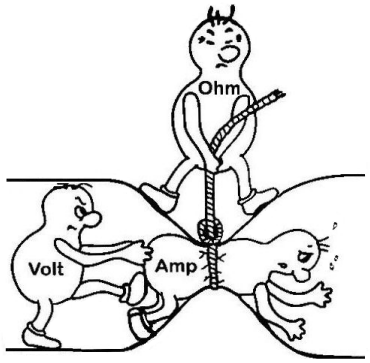
```
hole Stromstaerke
hole Widerstand
berechne Spannung
zeige Ergebnis
```

```
printf("I = ");
scanf("%f", &I);

printf("R = ");
scanf("%f", &R);

float U = R * I;

printf("U = %f\n", U);
```



Strukturierte Programmierung

Hochschule Fulda
University of Applied Sciences

- Schrittweises Verfeinern (→ Top-Down)
  - Ziel: Programmiersprache sollte uns beim Aufschreiben (komplexerer) Lösungen entgegenkommen
  - Lösung: Verwendung von Funktionen für zusammengehörende Programmteile
- Modularisierung und Strukturierung
  - Verbergen von Implementierungsdetails
  - Lokale Änderungen haben *lokale Auswirkungen*!
- Schnittstellen-Spezifikation
  - Wie soll sich der Algorithmus *nach außen verhalten*?
  - Formulierung von *Vor- und Nachbedingungen* an Startwerte und Ergebnisse

Implementierung (Teil 1a)

Hochschule Fulda
University of Applied Sciences

- Erster Entwurf (jeden Teilschritt mit eigener Funktion umsetzen):

```
int main(void) {
   float U, R, I;
   I = holeStromstaerke();
   R = holeWiderstand();
   U = berechneSpannung(R, I);
   zeigeErgebnis(U);
}
```

- Oft möchte man Teilprobleme separat lösen und (ggfs. mühevoll erarbeitete bzw. gut getestete) Lösungen mehrfach verwenden

Implementierung (Teil 1b)

Hochschule Fulda
University of Applied Sciences

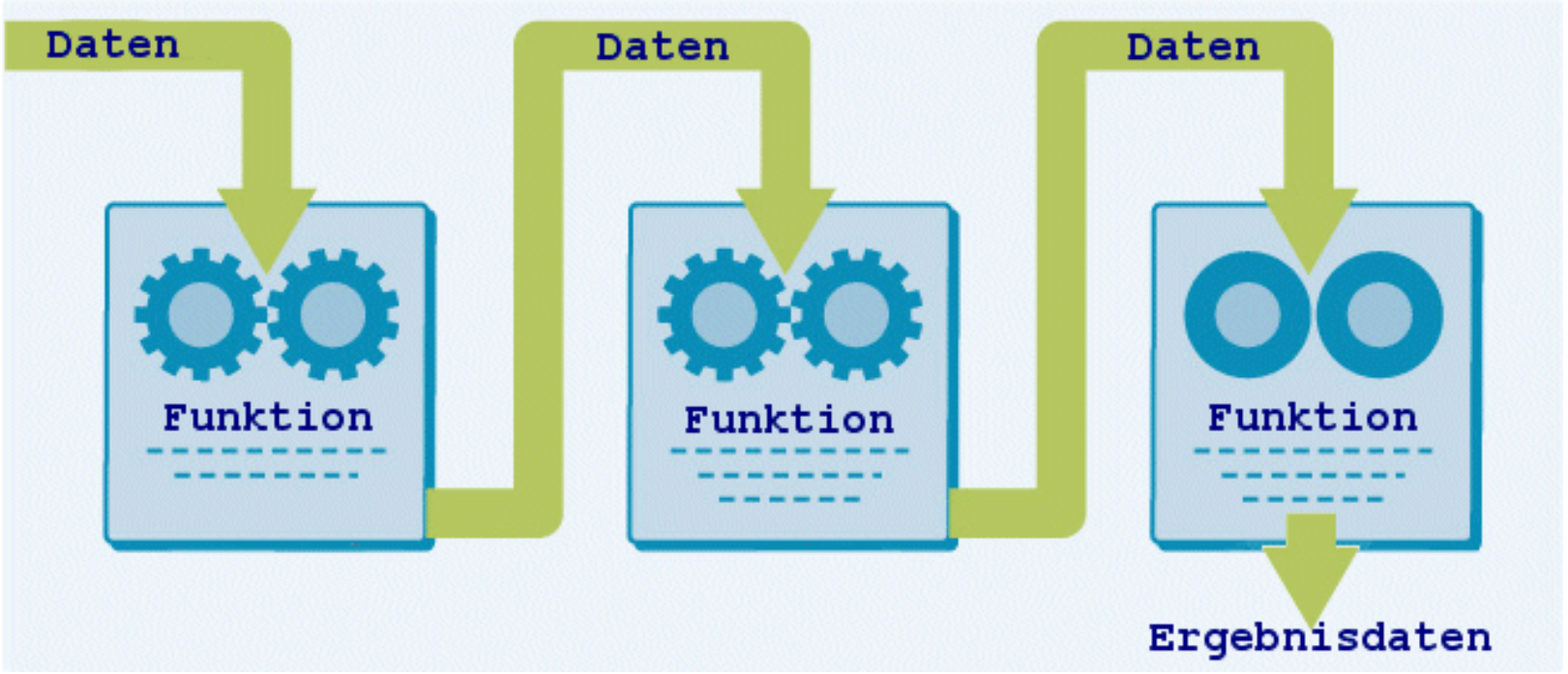
- Zweiter Entwurf (Verallgemeinerung der Ein-/Ausgabefunktionen):

```
int main() {
   float u, r, i;
   i = holeWert('I');
   r = holeWert('R');
   u = berechneSpannung(r, i);
   zeigeErgebnis('U', u);
}
```

- Werte einlesen kann z.B. als eine Funktionalität betrachtet werden
  - Abhängig vom übergebenen Zeichen wird Strom oder Widerstand geholt

Typischer Programmablauf

Hochschule Fulda
University of Applied Sciences



- Spannungsberechnung damit sogar als Einzeiler möglich:

```
zeigeErgebnis('U', berechneSpannung(holeWert('I'), holeWert('R')));
```

Definition von Funktionen

Hochschule Fulda
University of Applied Sciences

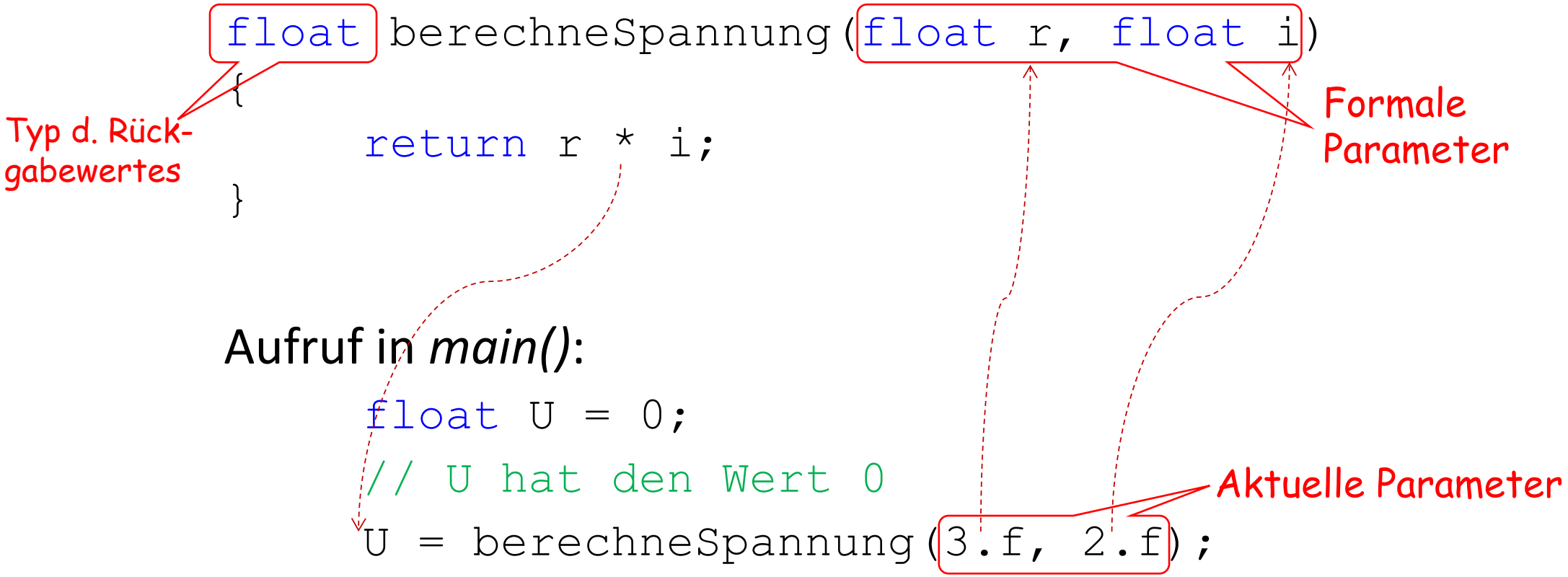
Funktionsname

```
float berechneSpannung(float r, float i)
{
      float u;          // Lokale Variable
      u = r * i;        // Spannungsberechnung
      return u;         // Rückgabe von Ergebnis
}
```

- Beim Aufschreiben der Funktion sind die tatsächlichen Variablen (bzw. Werte) noch nicht bekannt, mit denen sie später aufgerufen wird
- Aufruf (und Ausführung) einer Funktion ist zu unterscheiden von der Definition, was sie grundsätzlich tun soll

Ablauf bei Verwendung

Hochschule Fulda
University of Applied Sciences



Implementierung (Teil 2)

Hochschule Fulda
University of Applied Sciences

```c
float holeWert(char c)
{
  float value;
  printf("%c eingeben: ", c);
  scanf("%f", &value);
  return value;
}
```

```c
float berechneSpannung(float r, float i)
{
   return r * i;     //Spannung U = R * I
}
```

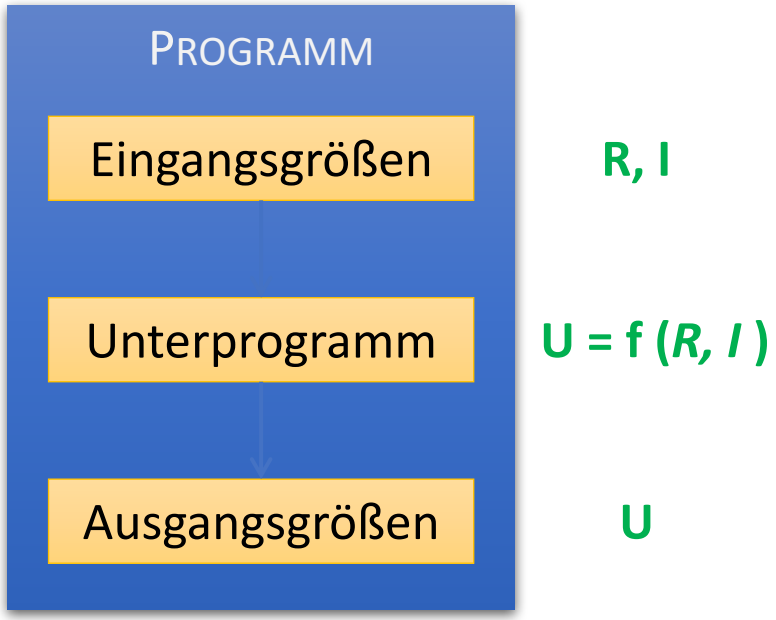
```c
void zeigeErgebnis(char c, float x)
{
   printf("%c betraegt: %f\n", c, x);
}
```

Schlüsselwort **void** bedeutet:

- Es gibt keinen Rückgabewert, **return** nicht nötig
- …außer Funktion soll bei gewissen Bedingungen vorzeitig verlassen werden
  - Dann steht **return** aber alleine, ohne Ausdruck dahinter

Schnittstellen

Hochschule Fulda
University of Applied Sciences



$$R = \frac{U}{I} \Rightarrow U = R \cdot I$$

- Zwischen Funktion und Umgebung muss definierte Schnittstelle existieren
  - Übergabe der Eingangsdaten
  - Rückgabe des Ergebnisses
- Vorbedingung?

```
float I, R;
```

  - I und R haben je positiven Wert
- Nachbedingung?

```
float U;
```

  - Es gilt U = R · I

Sichtbarkeit von Variablen

Hochschule Fulda
University of Applied Sciences

- Variablen, die innerhalb einer Funktion oder einem Block deklariert werden, heißen *lokale* Variablen
  - Andere Funktionen können *nicht* darauf zugreifen!
  - *Lebensdauer* lokaler Variablen ist auf die Dauer der Ausführung des Anweisungsblocks *beschränkt*
  - (Formale Parameter verhalten sich wie lokale Variablen)

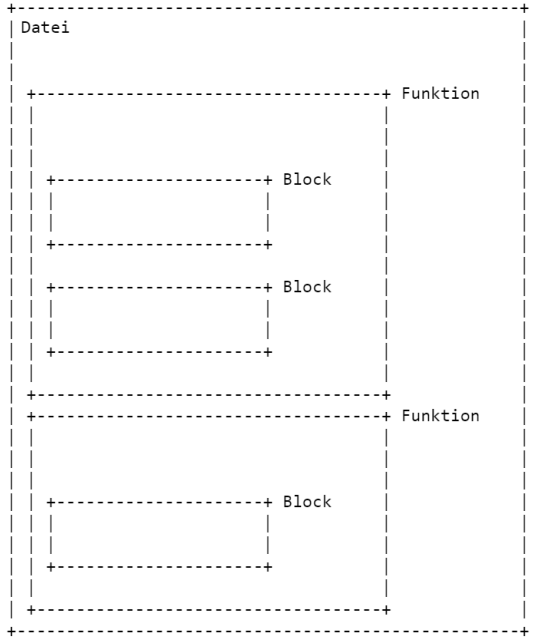
```
float berechneSpannung(float r, float i)
{
  float v;      // Lokale Variable
  v = r * i;    // Spannungsberechnung
  return v;     // Rückgabe von Ergebnis
}
```

```
int main()
{
  float U, R, I;
  // some code
  float U = berechneSpannung(R, I);
  // more code
}
```

Sichtbarkeit von Variablen

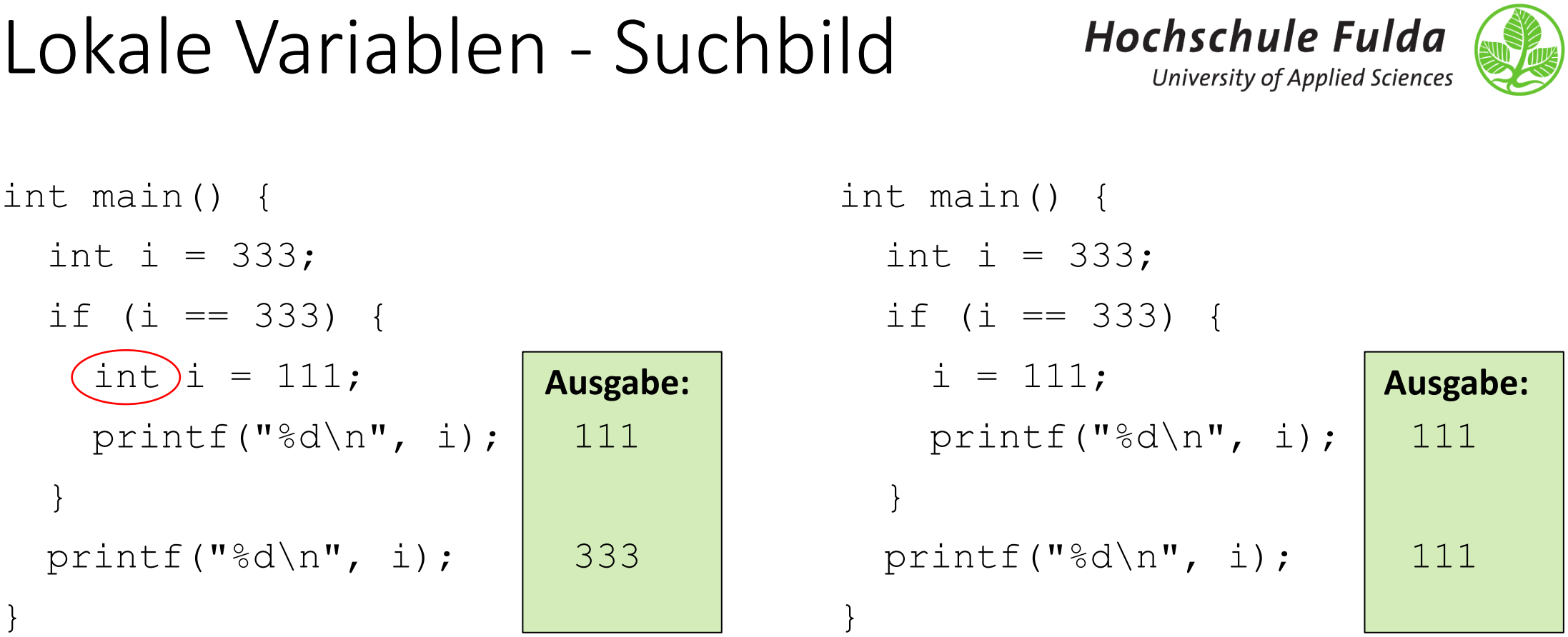
Hochschule Fulda
University of Applied Sciences

- Variablen (auch formale Parameter) sind nur in dem Anweisungsblock sichtbar (d.h. verwendbar), in dem sie deklariert wurden
  - Formale Parameter damit nur im Rumpf der sie deklarierenden Funktion sichtbar
- Variablen haben Gültigkeitsbereich (sog. „Scope“)
  - Lokal (zur Umgebung, die Variable umgibt)
    - Nach Verlassen von Rumpf / Anweisungsblock endet *Lebensdauer*!
    - Deklariert man in Funktion eine Variable mit Spezifizierer `static`, bleibt Wert nach Funktionsende erhalten (→ i.d.R. schlechter Stil)
      - Bsp.: `static int count = 0;`
  - Global (außerhalb aller Funktionen deklariert)
    - Durchkreuzt leider das Konzept modularer Programmierung ☹



Lokale Variablen - Suchbild

Hochschule Fulda
University of Applied Sciences



```
int main() {
  int i = 333;
  if (i == 333) {
    int i = 111;
    printf("%d\n", i);
  }
  printf("%d\n", i);
}
```

Ausgabe:

```
111

333
```

```
int main() {
  int i = 333;
  if (i == 333) {
    i = 111;
    printf("%d\n", i);
  }
  printf("%d\n", i);
}
```

Ausgabe:

```
111

111
```

- Woher kommt der Unterschied in der Ausgabe?
  - Innere Variablen verdrängen äußere (und lokale verdrängen damit globale)

Bsp.: Widerstandsberechnung

Hochschule Fulda
University of Applied Sciences

- Gleicher Ansatz wie eben:

```
hole Stromstaerke
hole Spannung
berechne Widerstand
zeige Ergebnis
```

```
int main() {
    float u, r, i;
    i = holeWert('I');
    u = holeWert('U');
    r = berechneWiderstand(u, i);
    zeigeErgebnis('R', r);
}
```

Durch einfache Modifikation auf
ähnliche Probleme erweiterbar ☺

$$R = \frac{U}{I} \Rightarrow U = R \cdot I$$

- Funktionen erlauben Strukturierung und Modularisierung
- Fehlervermeidung und Arbeitserleichterung durch Wiederverwendung

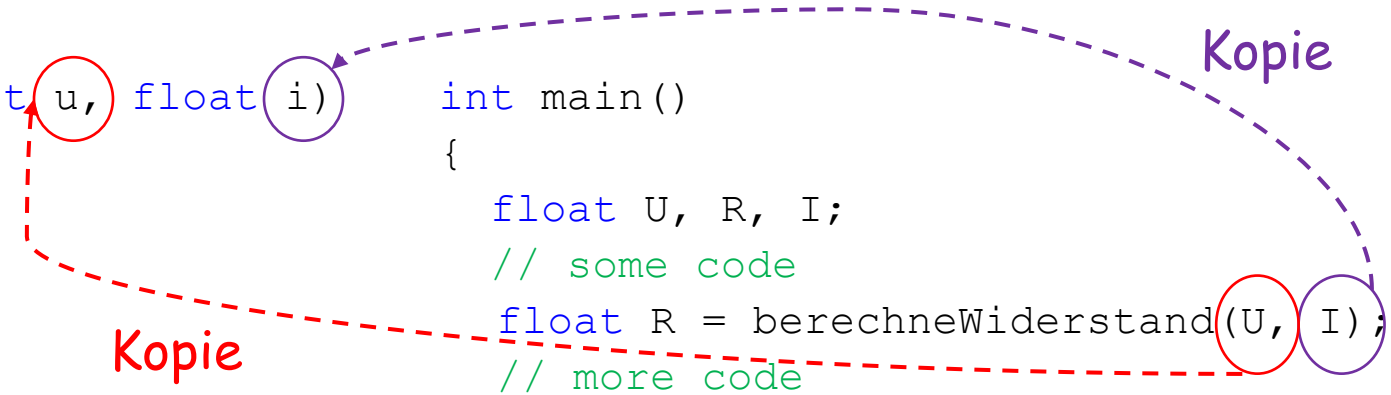
Funktionsaufruf

Hochschule Fulda
University of Applied Sciences

- Funktionen werden aufgerufen mit Namen u. Liste von Parametern (Argumente)
- *Argumente* müssen in *Datentyp*, *Reihenfolge* u. *Anzahl* der Signatur entsprechen
- Alle Argumente (aktuelle Parameter) werden von links nach rechts ausgewertet und an die formalen Parameter gebunden
  - Die Variablenwerte werden dabei je kopiert!

```
float berechneWiderstand(float u, float i)
{
  float r = 0;
  if (fabs(i) >= 1E-6f)
    r = u / i;
  return r;
}
```

```
int main()
{
  float U, R, I;
  // some code
  float R = berechneWiderstand(U, I);
  // more code
}
```



Beenden durch `return`

Hochschule Fulda
University of Applied Sciences

- Wird in aufgerufener Funktion (*Callee*) eine `return` Anweisung ausgeführt, wird der entsprechende Ausdruck zurückgegeben
  - In aufrufender Funktion (*Caller*) wird der Aufruf durch den Rückgabewert ersetzt
  - Die Kontrolle geht dabei an den Caller zurück
- In jeder *nicht*-`void` Funktion muss es mindestens eine (ggfs. auch mehrere) `return` Anweisung(en) geben

```
float berechneWiderstand(float u, float i)
{
  float r = 0;
  if (fabs(i) >= 1E-6f)
    r = u / i;
  return r;
}
```

Callee

```
int main()
{
  float U, R, I;
  // some code
  float R = berechneWiderstand(U, I);
  // more code
}
```

Caller

Kopie

Definition vs. Deklaration

Hochschule Fulda
University of Applied Sciences

- Quellcode wird beim Kompilieren "von oben nach unten" durchgegangen
  - Alles was verwendet werden soll, muss vorher durch Definition oder zumindest Deklaration dem Compiler bekannt gemacht werden
- Zuerst deklariert:

```
void hilfe();

int main() {
  hilfe();
  return 0;
}

void hilfe() {
  printf("Test\n");
}
```

- Zuerst definiert:

```
void hilfe() {
  printf("Test\n");
}

int main() {
  hilfe();
  return 0;
}
```

Definition vs. Deklaration

Hochschule Fulda
University of Applied Sciences

- Beispiel Definition:

```
int func(int a, int b) {
    return (a + b) * (a + b);
}
```

Eigentliche Funktion,
Implementierung meist nach main() bzw. in eigener Quelldatei

- Beispiel Deklaration:

```
int func(int a, int b);
```

≠

Funktionsprototyp, Angabe vor main() bzw. in Header-File

- Reihenfolge von Deklaration, Definition und Funktionsaufruf wichtig
  - Zusammengehörige Deklarationen meist in Header-File zusammengefasst
- Funktion wird vor erstem Aufruf definiert
  - Oder als *Funktionsprototyp* vorher deklariert („forward declaration“)
  - …und später, also nach *main()* oder in anderer Quelldatei, definiert

Vordefinierte Funktionen

Hochschule Fulda
University of Applied Sciences

- Mathematische Funktionen z.B. sind deklariert in `<math.h>`
  - Hinweis: Winkel werden alle in Bogenmaß (Radiant) übergeben

```
double sin(double x);
double cos(double x);
double tan(double x);
double asin(double x);
double acos(double x);
double atan(double x);
double atan2(double y, double x);
double pow(double x, double y);
```

```
double exp(double x);
double log(double x);
double log10(double x);
double log2(double x);
double sqrt(double x);
double ceil(double x);
double floor(double x);
double fabs(double x);
```

- Weitere wichtige Funktionen finden sich u.a. in `<stdlib.h>` (→ später…)
  - Z.B. Absolutbetrag für Ganzzahlen: `int abs(int n);`

Unterprogramme

Hochschule Fulda
University of Applied Sciences

- Funktionen können beim Aufruf mit Werten versorgt werden
  - *Aktuelle Parameter* (Variablen oder Werte, die beim Funktionsaufruf übergeben werden) müssen in *Anzahl, Typ, Reihenfolge* mit *formalen Parametern* übereinstimmen
- Die formalen Parameter verhalten sich innerhalb des Funktionsrumpfes wie Variablen
  - Außerhalb des Rumpfes sind Parameter nicht sichtbar (können nicht verwendet werden)
- Funktionen werden manchmal auch Unterprogramme genannt
  - Zudem unterscheidet man manchmal zwischen den Begriffen Funktion sowie Prozedur
  - *Funktionen* liefern Rückgabewert bestimmten Typs, können daher in Ausdrücken stehen
  - *Prozeduren* haben keinen Rückgabewert (Rückgabetyp `void`), werden bei Aufruf behandelt wie Anweisungen

Vielen Dank!

Noch Fragen?