

## Bezeichner

- Bezeichner (Namen) für
  - Variablen
  - Funktionen u. ä.
- sind Zeichenfolgen, die mit einem Buchstaben oder `_` beginnen
- Unterscheidung von Groß- und Kleinschreibung!!
- mindestens 31 Zeichen werden vom Compiler ausgewertet

## Literale

- bezeichnen eine Konstante, die durch ihren Wert dargestellt wird

- **ganzzahlige Literale:**

- int:    dezimal         $[1-9][0-9]^*$    oder 0                    z.B.: 26
- oktal          $0[0-7]^*$                     z.B.: 032
- hexadezimal    $0x[0-9a-f]^+$    oder  $0X[0-9A-F]^+$    z.B.: 0x1a

↔ stets positiv; ggf. Minus-Operator anwenden

- Suffix u oder U ↔ unsigned
- Suffix l oder L ↔ long

- **Gleitpunkt-Literale:**

- `double`: Dezimalbruch mit Dezimalpunkt z.B.: `300.0`  
Exponentialdarstellungen *Mantisse**e**Exponent* z.B.: `3e2`  
(zur Basis 10) *Mantisse**E**Exponent* z.B.: `.3E3`
- Suffix `f` oder `F`  $\rightsquigarrow$  `float`
- Suffix `l` oder `L`  $\rightsquigarrow$  `long double`

- **Character-Literale:**

- Zeichen in einfachen Hochkommata, z.B. `'0'`
- Ersatzdarstellungen nicht druckbarer Zeichen in einfachen Hochkommata, z.B. `'\n'`, `'\t'`, aber auch `'\'`
- Oktal- oder Hexadezimaldarstellung des Zeichens:  
`'\Oktalziffern'` (`'\060'`) bzw. `'\xHexadezimalziffern'` (`'\x30'`)

## Implizite Typumwandlung bei einfachen Typen

- Wenn Operatoren verschiedene Typen verknüpfen:
  - kleiner Ganzzahltypen werden immer nach `int` umgewandelt.  
(`unsigned short` in `unsigned int`, falls `short` und `int` äquivalent sind)
  - Umwandlung aller Operanden in den höchsten Typ des Ausdrucks gemäß

`int` → `unsigned int` → `long` → `unsigned long` → `long long`  
→ `unsigned long long` → `float` → `double` → `long double`

- Bei Wertzuweisungen:
  - Umwandlung des rechten Ausdrucks in Typ der linken Variablen

## Verhalten der Werte bei Typumwandlungen

- großer  $\rightarrow$  kleinerer Ganzzahltyp  $\rightsquigarrow$  *Abschneiden der oberen Bits*
- Ganzzahltyp  $\rightarrow$  Gleitpunkttyp  $\rightsquigarrow$  (meist) nächster darstellbarer Wert
- Gleitpunkttyp  $\rightarrow$  Ganzzahltyp  $\rightsquigarrow$  Abschneiden der Nachkommastellen
- Gleitpunkttyp  $\rightarrow$  Typ mit zu kleinem Wertebereich  $\rightsquigarrow$  *unbestimmt*

## Explizite Typumwandlung einfacher Typen

- erzwingen von Typumwandlungen
- (*Zieltyp*) *arithmetischer Ausdruck*
  - arithmetischer Ausdruck* zusammengesetzt aus
    - \* Variablen einfacher Typen
    - \* Literalen einfacher Typen
    - \* Operatoren und Klammern
- Beispiel: `(double) ganzzahl;`
- *Vorsicht*: gleiche Regeln wie bei impliziter Typumwandlung

## Kontrollstrukturen

1. **Block:**  $\{$  *Anweisung\_1*  
    *Anweisung\_2*  
    :  
    *Anweisung\_n*  
     $\}$

- Block fasst eine Folge von Anweisungen zusammen  $\rightsquigarrow$  „*Sequenz*“
  - Block kann überall auftreten, wo Anweisungen stehen dürfen  
 $\rightsquigarrow$  Blöcke können geschachtelt werden
  - kein Semikolon nach einem Block
- $\rightsquigarrow$  Anweisungen der `main()`-Funktion bilden einen Block

## 2. **while**-Schleife: `while` (*Ausdruck*) *Anweisung*

- Wiederholung der *Anweisung* solange, bis der *Ausdruck* den Wert 0 hat
  - **arithmetischer** *Ausdruck*
  - **boolescher** *Ausdruck* mit Rückgabewerten 1 für true oder 0 für false
    - ↔ Vergleichsoperatoren: ==, !=, <=, >=, <, >
    - und logische Operatoren &&, ||, !
- kann auch gar nicht ausgeführt werden ↔ „*abweisende Schleife*“

## 3. **do while**-Schleife: `do` { *Anweisung* } `while` (*Ausdruck*)

- wird mindestens einmal durchlaufen ↔ „*annehmende Schleife*“



#### 4. **for**-Schleife: `for (init; test; update)` *Anweisung*

<i>init</i> :	Anweisung	↔	Initialisierung des Schleifenzählers
<i>test</i> :	Ausdruck	↔	„while-Bedingung“
<i>update</i> :	Anweisung	↔	Überschreiben des Schleifenzählers

Beispiel: `for (i = 0; i < 10; i++)`

- Schleifenzähler muss als Ganzzahltyp definiert sein
- *init*, *test* oder *update* können leer sein, z.B. `for(;;)`  
↔ ggf. Anweisung(en) nutzen
- dynamische, abweisende Schleife

5. **Selektion** `if` (*Ausdruck*)  
    *Anweisung\_1*  
    else  
        *Anweisung\_2*

- Der else-Zweig ist optional.
- Bei Schachtelung von `if`-Anweisungen bezieht sich ein else-Zweig immer auf die letzte `if`-Anweisung ohne else.

```
if (x >= 0)
    if (x > 0)
        printf("groesser als Null\n");
    else;
else
    printf("kleiner als Null\n");
```

## 6. Mehrfachselektion – else if

- Auswahl unter mehreren Alternativen
- if-Anweisung als Anweisung im else-Zweig

```
if (Ausdruck_1)
    Anweisung_1
else if (Ausdruck_2)
    Anweisung_2
    :
else if (Ausdruck_n)
    Anweisung_n
else
    Anweisung_else           /* optional ...
                               ... optional */
```

## 7. Mehrfachselektion mit **switch** s. Literatur

## Sprunganweisungen

1. `break`;  $\rightsquigarrow$  Abbruch der Schleife  
 $\rightsquigarrow$  zur ersten Anweisung nach der Schleife
2. `continue`;  $\rightsquigarrow$  Abbruch des Schleifendurchlaufs  
 $\rightsquigarrow$  zum *Ausdruck* der Schleife
3. `goto Marke`; Sprung in Anweisung hinter *Marke*:  
nur zum Abfangen von Laufzeitfehlern einsetzen

## Fehlerarten

- **Compilerfehler:** Fehler, die der Compiler erkennt, z.B.
  - Semikolon vergessen
  - Variable benutzt aber nicht definiert etc.↔ meist kein ausführbares Programm
- **Laufzeifehler:** Fehler, die bei der Abarbeitung des Programms auftreten, z.B.
  - Division durch 0
  - fehlende Werte
  - Zugriff auf Datei, die nicht geöffnet werden kann
  - kein Speicherplatz mehr vorhanden etc.
- **logische Fehler:** Fehler im Programmentwurf

## Definition von Funktionen

*Funktionskopf Funktionsrumpf*

### 1. Funktionskopf

- *Rückgabetyp Bezeichner (Typ\_1 formaler\_Parameter\_1, Typ\_2 formaler\_Parameter\_2,  $\vdots$  Typ\_n formaler\_Parameter\_n)*
- legt Namen der Funktion fest: *Bezeichner*
- legt Definitionsbereich der Funktion fest: Liste *formaler Parameter*
- legt Wertebereich der Funktion fest: *Rückgabetyp*
- legt fest, wie die Funktion aufgerufen wird

## 2. Funktionsrumpf

- ist ein Block (Sequenz von Anweisungen)
- legt fest, wie die Eingabewerte verarbeitet werden
- return-Anweisung  $\rightsquigarrow$  Rückkehr zu aufrufender Funktion  
 $\rightsquigarrow$  Rückgabewert übergeben

Beispiel:

```
int quadrat (int n) {  
    return n * n;  
}
```

$\rightsquigarrow$  nach return kann Ausdruck stehen

## Funktionsaufruf

*Funktionsname (aktueller\_Parameter\_1, ..., aktueller\_Parameter\_n);*

1. automatisches Anlegen von lokalen Variablen für die Parameter

↪ *typ\_i formaler\_Parameter\_i;*

2. Initialisierung mit aktuellen Parametern

↪ *formaler\_Parameter\_i = aktueller\_Parameter\_i;*

Aktuelle Parameter können **Ausdrücke** sein!

3. Abarbeitung der Anweisungen im Funktionsrumpf

### Funktionsaufruf ist Ausdruck

```
int qud = quadrat(12);    // liefert qud = 144;
```