Bezeichner

- Bezeichner (Namen) für
  - Variablen
  - Funtktionen 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`

    $\rightsquigarrow$ stets positiv; ggf. Minus-Operator anwenden
  - Suffix `u` oder `U` $\rightsquigarrow$ `unsigned`
  - Suffix `l` oder `L` $\rightsquigarrow$ `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. `'\x`*Hexadezimalziffern*`'` (`'\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

$$\texttt{int} \rightarrow \texttt{unsigned int} \rightarrow \texttt{long} \rightarrow \texttt{unsigned long} \rightarrow \texttt{long long}$$
$$\rightarrow \texttt{unsigned long long} \rightarrow \texttt{float} \rightarrow \texttt{double} \rightarrow \texttt{long double}$$

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

Verhalten der Werte bei Typumwandlungen

- großer $\longrightarrow$ kleinerer Ganzzahltyp $\rightsquigarrow$ $Abschneiden\ der\ oberen\ Bits$
- Ganzzahltyp $\longrightarrow$ Gleitpunkttyp $\rightsquigarrow$ (meist) nächster darstellbarer Wert
- Gleitpunkttyp $\longrightarrow$ Ganzzahltyp $\rightsquigarrow$ Abschneiden der Nachkommastellen
- Gleitpunkttyp $\longrightarrow$ 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**: $\{Anweisung\_1$
   $Anweisung\_2$
   $\vdots$
   $Anweisung\_n$
   $\}$

   - 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`
       $\rightsquigarrow$ Vergleichsoperatoren: `==, !=, <=, >=, <, >`
       und logische Operatoren `&&, ||, !`
   - kann auch gar nicht ausgeführt werden $\rightsquigarrow$ „$abweisende\ Schleife$“

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

   - wird mindestens einmal durchlaufen $\rightsquigarrow$ „$annehmende\ Schleife$“

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

$init$:	Anweisung	$\rightsquigarrow$	Initialisierung des Schleifenzählers
$test$:	Ausdruck	$\rightsquigarrow$	„`while`-Bedingung“
$update$:	Anweisung	$\rightsquigarrow$	Überschreiben des Schleifenzählers

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

- Schleifenzähler muss als Ganzahltyp definiert sein
- $init$, $test$ oder $update$ können leer sein, z.B. `for(;;)`
  $\rightsquigarrow$ 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                          /* optional ...
  Anweisung_else                        ... 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.

  $\rightsquigarrow$ 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,
     ⋮
     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

   $\rightsquigarrow$ *typ_i formaler_Parameter_i*;

2. Initialisierung mit aktuellen Parametern

   $\rightsquigarrow$ *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;
```