C-Kurs
Kontrollstrukturen & Funktionen

Vorbemerkung: Syntax vs. Semantik

Syntax & Semantik

- C-Syntax:
  - Legt fest, welche Zeichenketten teil der Sprache C sind
- C-Semantik:
  - Legt fest, was sie bedeuten
- Beispiel:
  - Syntax: a + b
  - Bedeutung: Addition von a und b

Syntax-Fehler
Beispiel Absolutwert

- Beispiel 1:

```
if ( x < 0 ) {
     x = -x
}
```

- Beispiel 2:

```
if x < 0 {
     x = -x;
}
```

- Beispiel 3:

```
if x < 0 x = -x;
```

```
Korrekt u.a.:
if ( x < 0 ) {
       x = -x;
}

Oder
if ( x < 0 )
       x = -x;
```

- Konsequenz: Programm nicht kompilierbar/übersetzbar

Semantik-Fehler
Beispiel Absolutwert

- Beispiel 1:

```
if ( x > 0 ) {
     x = -x;
}
```

- Beispiel 2:

```
if (x < 0) {
     x = -2 * x;
}
```

```
Korrekt u.a.:
if ( x < 0 ) {
      x = -x;
}

Oder
if ( x < 0 )
      x = -x;
```

- Konsequenz: Programm kompiliert aber tut nicht was es soll…
- Meistens sehr viel schwieriger zu debuggen….

Fehlertypen

- Syntaxfehler
  - Konsequenz: Programm lässt sich nicht kompilieren/übersetzen
- Semantikfehler (auch häufig Programmlogikfehler)
  - Konsequenz: Programm kompiliert aber tut nicht das erhoffte

Bedingte Anweisungen

Blöcke

- Ein Block ist eine Zusammenfassung einer Folge von Anweisungen.

```
{ // begin of block
    int z = x;
    x = y;
    y = z;
} // end of block
```

- Eine Zusammenfassung von Ausdrücken wird in C durch geschweifte Klammern { ... } realisiert.

Blöcke

- Hilfreich für die Lesbarkeit des Programms und für die Fehlersuche ist eine an der Blockstruktur orientierte Einrückung (indentation).
- Blöcke können geschachtelt sein.

```
<block> ::= { <statements> }
<statements> ::=   <statement> |
                   <statements> <statement>
```

Syntax Beschreibung: Backus-Naur-Form (BNF)

- <block> ::= { <statements> }
- <statements> ::= <statement> | <statements> <statement>

- Diese Schreibweise heißt Backus-Naur-Form (BNF) und wird oft zur Syntax-Definition von Programmiersprachen benutzt.
- Es beschreibt die Syntax in rekursiver Weise
  - <komplexes Konstrukt> ::= <einfachere Konstrukte>

Blöcke

- Hilfreich für die Lesbarkeit des Programms und für die Fehlersuche ist eine an der Blockstruktur orientierte Einrückung (indentation).
- Blöcke können geschachtelt sein.

- <block> ::= { <statements> }
- <statements> ::= <statement> |
  <statements> <statement>

- <statement> ::= <assignment> ; | ….
- <assignment> ::= <variable> = <expression>
- <expression> ::= …

Bedingte Anweisung

- Manche Anweisungen sollen nur unter bestimmten Bedingungen ausgeführt werden.
  - Berechne den Absolutwert einer Variable:

```
if ( x < 0 ) {
  x = -x;
}
```

- Syntaktische Form:

```
if ( <condition> ) <block>
```

Bedingte Anweisung mit Alternative

- Verallgemeinerte Form der If-Anweisung lautet

```
if( <condition> ) <block> else <block>
```

- Abhängig von der Bedingung wird eine der Alternativen ausgeführt.
- Beispiel: Maximumsbildung

```
// set z to maximum of x and y
if( x > y ) {       // condition
  z = x;      // then-part
} else {
  z = y;      // else-part
}
```

Bedingte Anweisung mit Alternative

- Das `else if` wird verwendet, um abhängig von einer Bedingung zwischen verschiedenen Blöcken zu wählen:

```
if( n == 1 ) {              // execute block #1
}
else if( n == 2 ) {         // execute block #2
}
else if( n == 3 ) {         // execute block #3
}
else {                      // if all fails, execute block #4
}
```

Logische Ausdrücke (boolean expressions)

- Für logische Ausdrücke gibt es in C keinen speziellen Typ.
  - Wert == 0 => false / falsch
  - Wert != 0 => true / wahr
- Vergleichsoperatoren liefern Integer Werte 0 oder 1:

  == gleich, **!=** ungleich, **<** kleiner, **>** größer,
  <= kleiner gleich, **>=** größer gleich
- Verknüpfung von logischen Ausdrücken:
  - **&&** und (beides wahr)
  - **||** oder (eines von beiden wahr)

```
if( n == 1 || m == 2) {}
```

Schleifen und Iteration

Schleifen und Iterationen

- Es gibt häufig Situationen, in denen ein Programmstück mehrmals mit jeweils sich ändernden Werten durchlaufen werden soll: Schleifen.

- `for`–Schleife: Anzahl der Iterationen ist bekannt.
- `while`–Schleife: Anzahl der Iterationen wird durch eine Bedingung bestimmt.

Inkrementieren und Dekrementieren

- Nützlich sind die post-increment /-decrement Operatoren:
  - `i++` ⇔ `i=i+1`
  - `i--` ⇔ `i=i-1`
- Es existieren äquivalente pre-increment /-decrement Operatoren:
  - `++i` ⇔ `i=i+1`
  - `--i` ⇔ `i=i-1`
- Es gibt nur einen Unterschied, wenn der Operator in einem Ausdruck verwendet wird.

for-Schleife

- Beispiel: Zählt von 0 bis 10.

```
int i;
for( i = 0; i <= 10; i++) {
  printf("i: %d\n", i);
}
```

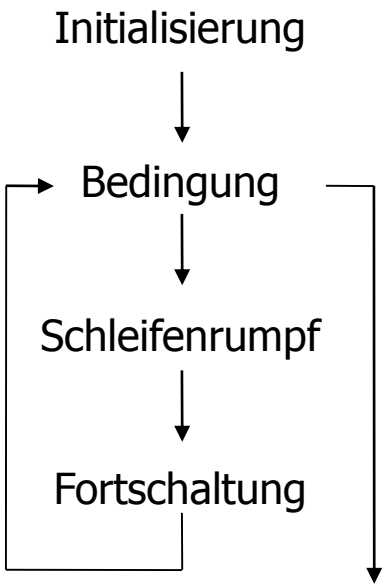
- Allgemein lautet die Syntax:

```
<for-statement> ::=
    for( <for-init>; <condition>; <for-update> )
    <block>

<for-init>      ::= <statement>
<for-update>    ::= <statement>
```

`for`-Schleife

1. **Initialisierung**: Deklaration und Zuweisung für die Schleifenvariable
2. **Bedingung**: Logischer Ausdruck. Wird vor jeder Ausführung des Schleifenrumpfs getestet.
3. **Schleifenrumpf**: Anweisung(en), die wiederholt ausgeführt werden, solange die Bedingung den Wert "true" ergibt.
4. **Fortschaltung**: Anweisungen, die den Wert der Schleifenvariablen ändern.



for-Schleife

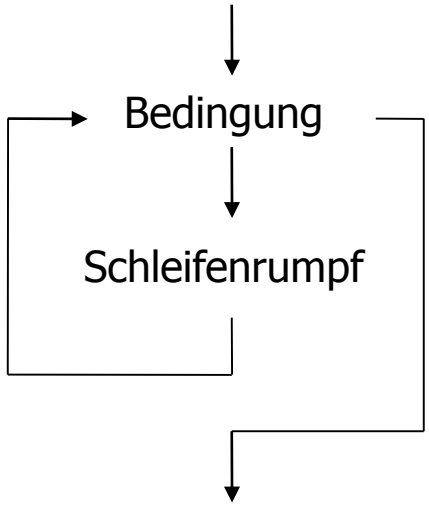
- Schleifen können ineinander geschachtelt werden:

```
int a, b;
for( a = 1; a < 10; a++) {
  for( b = 1; b < 10; b++) {
     printf("a * b: %d ", a * b);
  }
  printf("\n");
}
```

while-Schleifen

- Allgemeine Form:

```
<while-statement> :=
        while( <condition> ) <block>;
```



  - Unter Umständen wird der Schleifenrumpf nie ausgeführt! (Kann auch bei einer for-Schleife passieren)
  - while- und for-Schleifen sind semantisch äquivalent

Vergleich: `while`-/`for`-Schleife Zählen von 0 bis 10

while-Schleife

```
int i = 0;
while( i <= 10) {
  printf("i: %d\n", i);
  i++;
}
```

for-Schleife

```
int i;
for(i=0; i <= 10; i++) {
  printf("i: %d\n", i);
}
```

Funktionen

Funktionen: Motivation

- Strukturierte Programmierung:
  - Modularisierung
  - Vermeidung von komplexen Kontrollstrukturen
  - Kapselung
  - Dokumentation
- Vorteile
  - Übersichtlicher
  - Lesbarer
  - Testbarkeit
  - Wiederverwendbarkeit
  - Wartbarkeit

Funktionen – Beispiel

Berechnung des Maximums

```
int max (int a, int b) {

  if( a > b ) {
    return a;
  } else {
    return b;
  }
}
```

Funktionen – Beispiel

Berechnung des Maximums

```
// function to calculate the maximum of a and b
int max (int a, int b) {

  if( a > b ) { // condition
    return a;   // a is max => return its value
  } else {
    return b;   // b is max => return its value
  }
}
```

Funktionen (vereinfacht)

- Vereinfachte Form der Funktion ist

  `type name ( parameters ) <block>`
- Der Typ der Funktion ist der Typ des Rückgabewertes
- Rückgabe eines Wertes mittels Schlüsselwort **`return`**

- Beispiel: Maximum

```
// function to calculate the maximum of a and b
int max (int a, int b) {
  if( a > b ) {    // condition
    return a;      // a is max => return its value
  } else {
    return b;      // b is max => return its value
  }
}
```

Funktionsaufruf

```
…                                   // Definition of max
int main() {
  int r1, r2;
  int n = 10;
  int m = 11;

  r1 = max(10, 11);                 // Aufruf mit festen Werten
  r2 = max(n, m);                   // Aufruf mit Variablen

  printf("max of 10,11: %d\n", r1);
  printf("1: max of n, m: %d\n", r2);
  printf("2: max of n, m: %d\n", max(n, m));
            // Aufruf innerhalb einer anderen Funktion
}
```

Funktionen

- Funktionen bilden das Grundgerüst jedes Programms:
  - Modularisierung
  - Vermeidung von komplexen Kontrollstrukturen
  - Kapselung
  - Dokumentation
- Gültigkeit von Variablen / Scoping
  - Immer innerhalb des Blockes, in dem sie definiert sind
  - Gilt insbesondere für die Variablen einer Funktion
  - Wertübergaben von einer Funktion an eine andere mittels Parameter und Rückgabewert

Funktionsaufrufe

- Jede Funktion kann von jeder Funktion aufgerufen werden
- Beispiele:
  - max() von main() aus
  - max() von printf() aus

- Insbesondere kann eine Funktion auch sich selber aufrufen! => Rekursion

Funktionen (vereinfachte Syntax)

- <function> ::= <declarator-specifier><declarator> block

- <declarator-specifier> ::= <type-specifier> | …
- <type-specifier> ::= void | int | char | …
- <declarator> ::= <identifier> () |
  <identifier> (<parameter-list>) |
  …
- <parameter-list> ::= <type-specifier> <identifier> |
  <type-specifier> <identifier>, <parameter-list>
- <identifier> ::= …