Inhalt

4 C-Anweisungen 4-2
4.1 Ausdrucksanweisungen 4-3
4.2 Zusammengesetzte Anweisungen (Anweisungsblöcke) 4-3
4.3 Schleifenanweisungen 4-4
4.3.1 while - Schleife 4-4
4.3.2 do - Schleife 4-5
4.3.3 for - Schleife 4-6
4.4 Strukturbezogene Sprunganweisung 4-7
4.5 Auswahlanweisungen 4-9
4.5.1 if - Anweisung 4-9
4.5.2 switch - Anweisung 4-10
4.6 Beispiel Euklidischer Algorithmus 4-13

4 C-Anweisungen

Ein Programm besteht aus einer Folge von Funktionen. Jede Funktion hat einen Deklarationsteil, eine Folge von lokalen Deklarationen, und einen Anweisungsteil, einer Folge hintereinandergeschriebener Anweisungen.

Ausdrucksanweisungen
zusammengesetzte Anweisungen
Schleifenanweisungen
Auswahlanweisungen
Sprunganweisungen

4.1 Ausdrucksanweisungen

Syntax: *[Ausdruck]* **;**

Semantik: Berechnung des Ausdrucks**.**

Das Semikolon **;** ist hier ein **Abschlusssymbo**l, kein Trennzeichen.

	i + 1;	keine Wirkung nach außen.
	++ i;	**i** wird inkrementiert.
Leeranweisung:	**;**	

```
double r = 2, PI = 3.141593, U, A;
U = 2 * PI * r; A = PI * r * r;
```

4.2 Zusammengesetzte Anweisungen (Anweisungsblöcke)

Syntax: **{** *[lokale Deklaration ...] [Anweisung ...]* **}**
Leeranweisung: {}
Blöcke können geschachtelt werden.

Semantik:
Die angegebenen Anweisungen werden hintereinander entsprechend ihrer Reihenfolge abgearbeitet.

Verfügbarkeitsbereich lokaler Deklarationen.

Gültigkeit:	**Sichtbarkeit einer Größe**	⇒	**Zugriff**
Verfügbarkeit:	**Lebensdauer einer Größe**	⇒	**Speicher**

bloecke.c

```
/*
 * Demonstration von Lebensdauer und Sichtbarkeit
 */
# include <stdio.h>

int main()
{
  int x = 0; printf( "%d\n", x);                /* 0 */
  {
    int x = 1; printf( "%d\n", x);              /* 1 */
    {
      int x = 2; printf( "%d\n", x);            /* 2 */
    }
    printf( "%d\n", x);                         /* 1 */
  }
  printf( "%d\n", x);                           /* 0 */
  return 0;
}
```

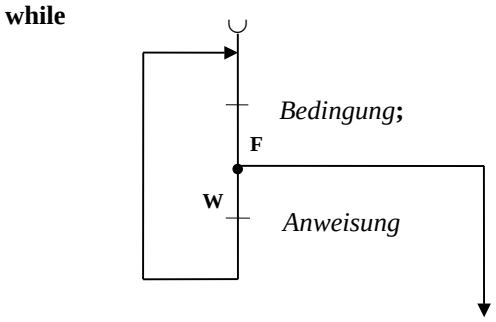
4.3 Schleifenanweisungen

4.3.1 while - Schleife

Die ***while*** Schleife ist eine **vorgeprüfte (prechecked)**, **abweisende Schleife**: Die *Bedingung* wird überprüft *bevor* die Anweisung ausgeführt wird. Solange die *Bedingung* wahr ist wird die *Anweisung* wiederholt.

Syntax: **while (** *Bedingung* **)** *Anweisung*

Semantik:
1. Die *Bedingung* wird ausgewertet.
2. Ist ihr Wert wahr (logisch interpretiert), so wird die *Anweisung* ausgeführt und mit 1. fortgesetzt.




```
int x = 5;

while( x > 0 )
{
  printf("%d \n", --x);
}
```

Ausgaben in den einzelnen Schleifendurchläufen:

Durchlauf	x
0	5
1	**4**
2	**3**
3	**2**
4	**1**
5	**0**
	0

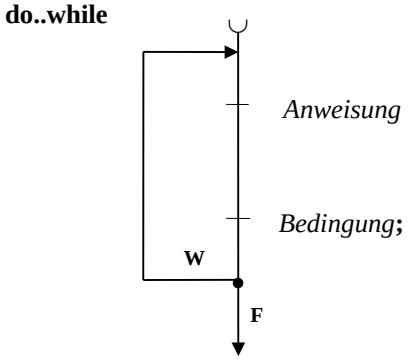
4.3.2 do - Schleife

Die ***do*** Schleife ist eine **nachgeprüfte (postchecked)**, **nicht abweisende Schleife**: Die *Anweisung* wird ausgeführt *bevor* die Bedingung überprüft wird. Solange die *Bedingung* wahr ist wird die *Anweisung* wiederholt.

Syntax: **do** *Anweisung* **while (** *Bedingung* **);**

Semantik:
1. Die *Anweisung* wird ausgeführt.
2. Ist der Wert der *Bedingung* wahr, so wird mit 1. fortgesetzt.

Das Semikolon am Ende der ***do*** - Schleife ist für die Syntaxanalyse notwendig!




```
int x, y; x = y = 0;

do
{
  x += ++y;
  printf( "%d %d\n", x, y);
} while( y < 5);
```

Zwischenergebnisse in den einzelnen Schleifendurchläufen:

Durchlauf	x	y
	0	0
1	**1**	**1**
2	**3**	**2**
3	**6**	**3**
4	**10**	**4**
5	**15**	**5**
	15	5

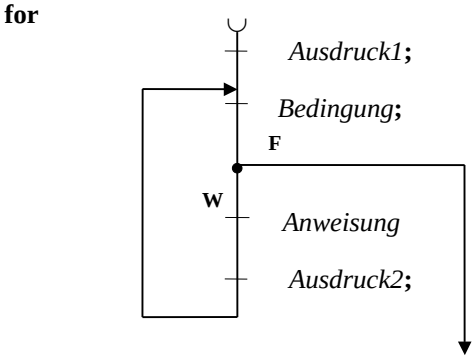
Unterschied zur *while* – Schleife:
Während ***while*** - Schleifen zuerst die Schleifenbedingungen testen und dann die Anweisung ausführen, führen die ***do*** - Schleifen die Anweisung einmal aus, bevor sie die Bedingung überprüfen:

In der ***while*** - Schleife wird der Schleifenkörper **nicht unbedingt** durchlaufen.
In der ***do*** - Schleife wird der Schleifenkörper **mindestens** einmal durchlaufen.

4.3.3 for - Schleife

Syntax: **for (** *[Ausdruck1]* **;** *[Bedingung]* **;** *[Ausdruck2]* **)** *Anweisung*
Das Semikolon fungiert hier als Trennzeichen!

Semantik:
1. Der *Ausdruck1* wird ausgeführt.
2. Ist der Wert der *Bedingung* falsch, so wird die Schleife abgebrochen.
3. Die *Anweisung* wird ausgeführt.
4. Der *Ausdruck2* wird ausgeführt, anschließend bei 2. fortgesetzt.



Ausdruck1, *Bedingung* und *Ausdruck2* sind optional. Fehlt die *Bedingung*, so muss man explizit für den Abbruch sorgen!

Die Fakultät natürlicher Zahlen wird iterativ definiert:

$$0! = 1$$

$$n! = \prod_{k=1}^{n} k$$

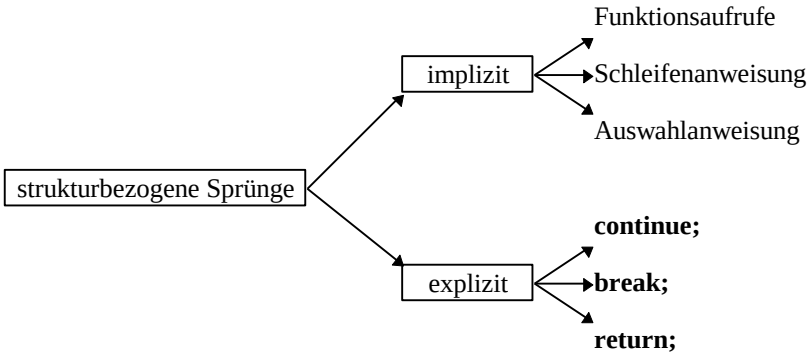
itFak.c

```
/*
 * Berechnen der Fakultaet fuer n = 12, iterativ
 */
# include <stdio.h>

int main()
{
  int i, n = 12;  /* 13! = 6'227'020'800 > ULONG_MAX */
  unsigned long fak = 1;

  for( i = 2; i <= n; i++) fak *= i;

/*
 * for( i = 2; i <= n; fak *= i++);
 * for( ; n > 1; fak *= n--);
 */

  printf( "%lu\n", fak);              /* 479'001'600 */

  return 0;
}
```

`ULONG_MAX` Macro in **<*limits.h*>** für die größte darstellbare ***unsigned long*** - Zahl
`%lu` Formatbeschreiber für ***unsigned long*** - Ausgabe

4.4 Strukturbezogene Sprunganweisung

Durch die Strukturierungsmöglichkeiten einer Programmiersprache gibt es eine Vielzahl von Sprüngen, die beim Programmablauf ausgeführt werden. Das Verlassen einer Schleifen erfolgt durch eine Bedingung. Eine Auswahlanweisung steuert das Durchlaufen eines Programmteils in Abhängigkeit einer Bedingung.



Es ist aber auch möglich, explizit ein Verlassen einer Anweisung zu erzwingen.

Beenden einer Anweisung
Syntax: **break;**

Semantik:
Die Abarbeitung der **Anweisung** wird abgebrochen.

Speziell für Schleifen kann ein Schleifendurchlauf abgebrochen werden.

Beenden eines Schleifendurchlaufs
Syntax: **continue;**

Semantik:
Die Abarbeitung des **Schleifendurchlaufs** wird abgebrochen, danach die Schleifenbedingung überprüft und entsprechend ihrem Wert die Schleife weiterbehandelt.

Beispiel ***formale Endlosschleife***.

sum.c

```
/*
 * sum.c: Summenbildung
 */
# include <stdio.h>

int main()
{
  int zahl, summe = 0, add = 0;

  do
  {                                  /* Schleifenabbruch */
    if( scanf( "%d", &zahl) == EOF) break;
```

```
      if( zahl == 0) continue;     /* Durchlaufabbruch */

      summe += zahl; ++add;
    } while( 1);             /* Formale Endlosschleife */

    printf( "Summe: %d\n", summe);
    printf( "Additionen: %d\n", add);

    return 0;
  }
```

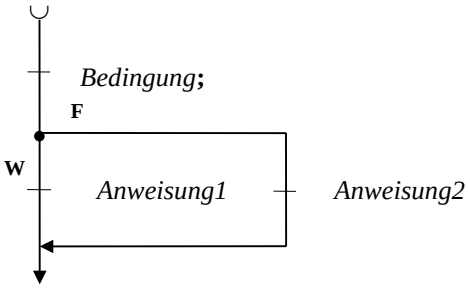
4.5 Auswahlanweisungen

4.5.1 if - Anweisung

Syntax: **if (** *Bedingung* **)** *Anweisung1* *[* **else** *Anweisung2]*

Semantik:
1. Die *Bedingung* wird ausgewertet.
2. Ist ihr Wert wahr, so wird die *Anweisung1* ausgeführt.
3. Ist ihr Wert falsch, so wird, falls vorhanden, die *Anweisung2* ausgeführt.

if..else



Lösung der Gleichung $ax+b=0$.

```
if( a == 0)
  if( b == 0) printf("L = R\n");
  else printf("L = {}\n");
else printf("L = { %f}\n", -b/a);
```

Bei geschachtelten ***if*** - Anweisungen wird einem ***else*** stets das letzte vorhergehende ***if*** zugeordnet. Evtl. sind Leeranweisungen erforderlich.

Auswahlanweisung oder bedingte Anweisung ?

Die bedingte Ausdrucksanweisung

x = *Bedingung* **?** *Ausdruck1* **:** *Ausdruck2* **;**

ist folglich äquivalent mit der Auswahlanweisung

if (*Bedingung* **) x =** *Ausdruck1* **; else x =** *Ausdruck2* **;**.

Das Beispiel (s.o.)

```
quadrant = x >= 0? y >= 0? 1: 4: y >= 0? 2: 3;
```

als ***if*** - Anweisung:

```
if( x >= 0)
  if( y >= 0) quadrant = 1; else quadrant = 4;
else
  if( y >= 0) quadrant = 2; else quadrant = 3;
```

4.5.2 switch - Anweisung

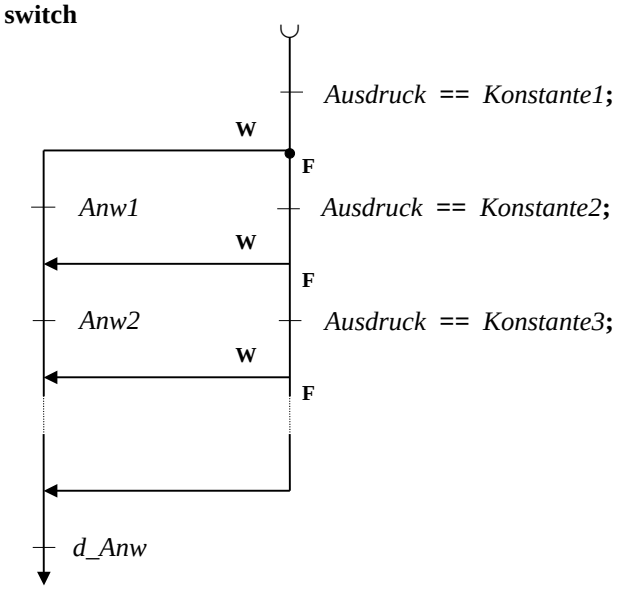
Syntax:

```
switch ( int_Ausdruck )
{
   case int_Konstante1 : Anweisung11 Anweisung12 ...
   case int_Konstante2 : Anweisung21 Anweisung22 ...
   . . .
[..default : d_Anweisung1 d_Anweisung2 ... ]
}
```

Der ***default*** - Fall kann an jeder Stelle in der Auswahlliste stehen.

Semantik:
1. Der *int_Ausdruck* wird berechnet.
2. Sein Wert wird mit den *int_Konstanten* verglichen.
3. Bei Gleichheit wird die Arbeit dort **fortgesetzt**.
4. Bei Ungleichheit mit **allen** *int_Konstanten* wird, falls vorhanden, bei ***default*** fortgesetzt.

Ablaufdiagramm einer ***switch*** - Anweisung für den Sonderfall, das ***default*** am Ende der Auswahlliste steht:



Semantische Falle:
Entgegengesetzt zu anderen Programmiersprachen haben wir es hier nicht mit einer üblichen Fallunterscheidung zu tun, denn nach dem Einsetzen der Abarbeitung von Anweisungen werden alle darauffolgenden Anweisungen auch abgearbeitet. Um das zu verhindern, muss man mit ***break;*** die ***switch*** - Anweisung explizit abbrechen.
Natürlich kann man sich die hier verwendete Implementierung auch zunutze machen:

```
. . .    case Sub : op2 = - op2;
         case Add : . . .
```

zahl1.c

```
/*
 *Erzeugen des zu einer Zahl gehoerigen Zahlwortes
 */

# include <stdio.h>

int main()
{
  int n;

/* Eingabe */
  printf( "Eingabe Zahl zwischen 100 und 999 :\n");
  scanf( "%d", &n);

/* Verarbeitung und Ausgabe */
  switch( n / 100)
  {
    case 1: printf( "Ein"); break;      /* Hunderter */
    case 2: printf( "Zwei"); break;
    case 3: printf( "Drei"); break;
    case 4: printf( "Vier"); break;
    case 5: printf( "Fuenf"); break;
    case 6: printf( "Sechs"); break;
    case 7: printf( "Sieben"); break;
    case 8: printf( "Acht"); break;
    case 9: printf( "Neun"); break;
    default: printf( "Falsche Zahl\n"); return 1;
  }

  printf( "hundert");

  switch( n % 100)
  {
    case  1: printf( "eins"); break; /* Sonderfaelle */
    case 11: printf( "elf"); break;
    case 12: printf( "zwoelf"); break;
    case 16: printf( "sechzehn"); break;
    case 17: printf( "siebzehn"); break;
    default: switch( n % 10)   /* 13, 14, 15, 18, 19 */
             {
               case 0:  /* nichts */ break; /* Einer */
               case 1:  printf( "ein"); break;
               case 2:  printf( "zwei"); break;
               case 3:  printf( "drei"); break;
               case 4:  printf( "vier"); break;
               case 5:  printf( "fuenf"); break;
               case 6:  printf( "sechs"); break;
               case 7:  printf( "sieben"); break;
               case 8:  printf( "acht"); break;
               case 9:  printf( "neun");
             }

             if( n % 100 / 10 > 1) printf( "und");
```

```
                switch( n % 100 / 10)
                {
                  case 0:  /* nichts */ break;/* Zehner */
                  case 1:  printf( "zehn"); break;
                  case 2: printf( "zwanzig"); break;
                  case 3: printf( "dreiszig"); break;
                  case 4: printf( "vierzig"); break;
                  case 5: printf( "fuenfzig"); break;
                  case 6: printf( "sechzig"); break;
                  case 7: printf( "siebzig"); break;
                  case 8: printf( "achtzig"); break;
                  case 9: printf( "neunzig");
                }
    }

    printf( "\n");

    return 0;
  }
```

4.6 Beispiel Euklidischer Algorithmus

Berechnung des größten gemeinsamen Teilers ***ggT(m, n)*** und des kleinsten gemeinsamen Vielfachen ***kgV(m, n)*** zweier natürlicher Zahlen ***n*** und ***m*** mittel Euklidischen Algorithmus.

"*Eingabe, Verarbeitung, Ausgabe*" . . . Das **EVA – Prinzi**p der Datenverarbeitung.

ggTkgV.c

```
/*
 * ggTkgV.c:
 * Berechnen des groessten gemeinsamen Teilers und
 * des kleinsten gemeinsamen Vielfachen
 * zweier natuerlicher Zahlen.
 */

# include <stdio.h>

int main()
{
  unsigned int m, n, a, b, c;

/* Eingabe */
  printf( "m = "); scanf("%d", &m);
  printf( "n = "); scanf("%d", &n);

/* Verarbeitung */
  a = m; b = n;
  do
  {
    c = a % b;        /* Rest der ganzzahligen Division */
    a = b; b = c;             /* Vertauschen der Werte */
  } while( c != 0);

/* Ausgabe */
  printf( "ggT( %d, %d) = %d\n", m, n, a);
  printf( "kgV( %d, %d) = %d\n", m, n, m * n /a);

  return 0;
}
```