6. Funktionen

Keine Programmiersprache kommt ohne Funktionen aus

- Wiederverwertbare Programmfragmente
- Unterteilung in Teilprobleme
- Beliebig oft aus anderen Funktionen aufrufbar
- Keine Unterscheidung von Prozeduren, Funktionen etc.

6.1. Definition

```
typ funktions_name (parameterliste)
{
   vereinbarungen
   anweisungen
}
```

wobei

- funktions_name ist der Name der Funktion (Bezeichner)
- typ ist der Datentyp der Funktion. Bei fehlender Datentypangabe wird int angenommen.
- parameterliste ist die kommagetrennte Aufzählung der Parameter der Funktion mit Datentyp und Name

Bemerkung

Dies ist die neue Form der Funktionsdefinition nach ANSI C. Die ältere K&R Form wird hier nicht besprochen. Compiler, die die neue Form nicht unterstützen, sind veraltet.

Beispiel

```
1:      int fakultaet ( int n )
2:      {
3:          int i, resultat=1;
4:
5:          for ( i=2; i<=n; i++ ) resultat *= i;
6:
7:          return resultat;
8:      }
```

Zeile 1 ist der Kopf (head) der Funktion. Der Name der Funktion ist fakultaet. Ihr Datentyp (Datentyp des Rückgabewertes) ist int. Die Funktion benötigt einen Parameter vom Typ int.

Der Teil zwischen **{** (Zeile 2) bis **}** (Zeile 8) ist der Rumpf (body) der Funktion. Der Rumpf ist ein Block und enthält Vereinbarungen und Anweisungen. Im Rumpf wird die eigentliche Aufgabe der Funktion ausgeführt.

Die return-Anweisung beendet die Funktion und sorgt für die Rückgabe des Resultates.

An anderen Stellen des Programms wird die Funktion über ihren Namen aufgerufen

```
k = 3 * fakultaet(10) - 18;
```

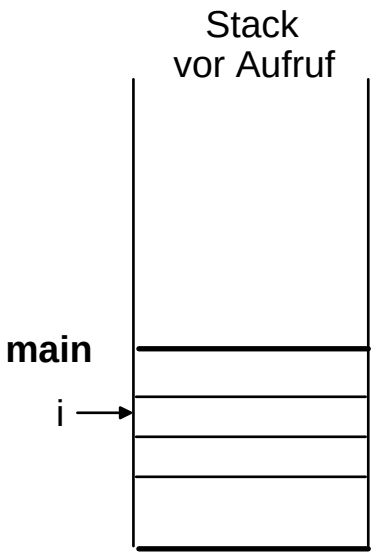
Die Funktion kommuniziert über ihre Parameter und den Rückgabewert mit anderen Programmteilen.

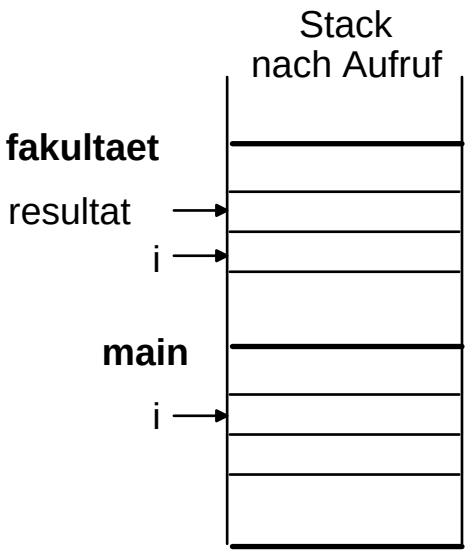
6.2. Lokale Variable

- Innerhalb der Funktion definiert
- nicht sichtbar in anderen Funktionen
- werden beim Aufruf der Funktion angelegt und am Ende wieder gelöscht

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

Lokale Variable werden in einem besonderen Bereich des Memory, dem **Stack**, verwaltet (zusammen mit anderen privaten Daten von Funktionen)





In beiden Funktionen main und fakultaet tritt eine Variable i auf. i in main und i in fakultaet sind völlig verschiedene Variable.

Die Parameter gehören auch zu den lokalen Variablen. Sie werden zusätzlich vom aufrufenden Programmcode mit Werten belegt.

Es gibt auch globale Variable: Definition ausserhalb einer Funktion (siehe weiter unten).

Im Gegensatz zu Programmiersprachen wie Pascal können Funktionen nicht innerhalb einer anderen Funktion definiert werden: keine lokalen Funktionen.

6.3. Aufruf und Parameterübergabe

Die Funktion wird aufgerufen über ihren Namen. In Klammern dahinter folgen die Argumente der Funktion.

```
k = fakultaet(7);
```

Die Klammern müssen auch bei einer leeren Parameterliste angegeben werden (Unterschied Variable ⇔ Funktionsaufruf).

Die Reihenfolge, die Anzahl und der Datentyp der Argumente muss mit der Parameterliste der Funktionsdefinition übereinstimmen (Beachte jedoch: Funktionen mit variabler Parameterliste).

Der Funktionsaufruf ist ein Ausdruck => kann wie Variable oder Konstante in anderen Ausdrücken verwendet werden (solange mit Datentyp vereinbar).

Übergabemechanismus: call by value

Beispiel

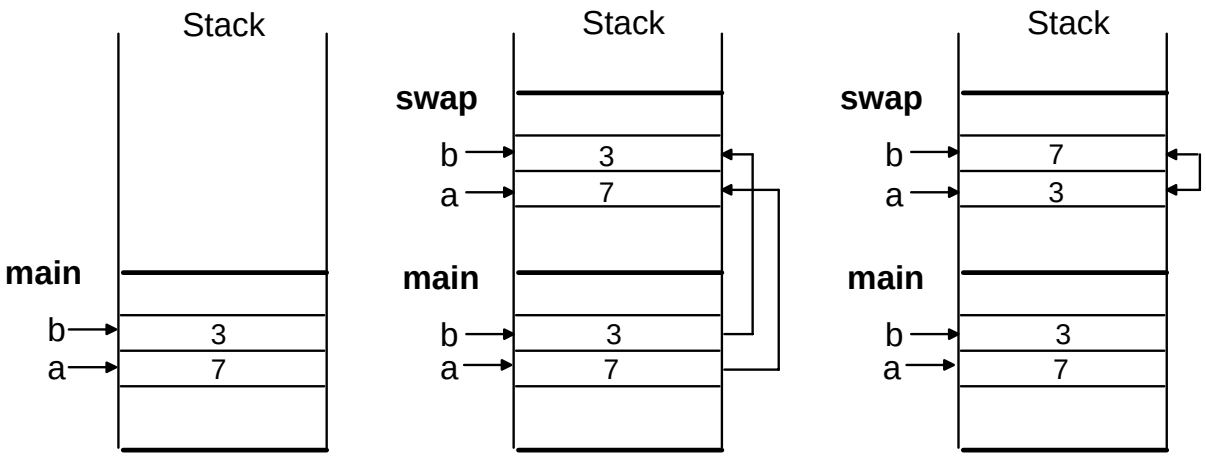
```
void swap ( int a, int b )
{
      int tmp = a;
      a = b;
      b = tmp;
}

a = 7;   b = 3;
printf("VOR     a=%d, b=%d\n", a, b);
swap(a,b);
printf("NACH    a=%d, b=%d\n", a, b);

/* Ausgabe:      VOR          a=7, b=3
 *               NACH         a=7, b=3  */
```

Was passiert?

1. Vor Funktionsaufruf werden die Argumente ausgewertet.
2. Die Werte werden der Funktion in der richtigen Reihenfolge übergeben
3. Funktion übernimmt die Werte und legt sie in **lokale Variable** ab
4. Anweisungen der Funktion werden ausgeführt
5. Rückgabewert wird in die aufrufende Funktion kopiert



Parametervariable können durch Funktionsaufruf nicht geändert werden.

Lösung: Übergabe der Adresse statt des Wertes:

```
swap(&a,&b);
```

Übergabe von Arrays

Beispiel

```
void arinc ( int n, int B[] )
{
      int i;
      for ( i=0; i<n; i++ ) B[i]++;
}

main ()
{
      int A[3] = { 1, 2, 3 };
      printf("%d %d %d\n", A[0], A[1], A[2]);
      arinc(3,A);
      printf("%d %d %d\n", A[0], A[1], A[2]);
}

/* Ausgabe:     1 2 3
 *              2 3 4    */
```

Nach dem Aufruf von arinc sind die Werte von A verändert. Funktion hat Zugriff auf ursprüngliche Arrayelemente!

Arrays werden nach **call by reference** übergeben.

Erklärung folgt später: Arraynamen sind Zeiger auf Speicherbereich des Arrays.

6.4. Rückgabewerte

- Funktion liefert Rückgabewert => Verwendung in einem Ausdruck im aufrufenden Programmteil.
- Funktion hat Datentyp => Datentyp des Rückgabewertes.
- Funktion ohne Datentyp definiert => implizit int als Datentyp angenommen.

Bemerkung: Die Funktion main hatte immer den Datentyp int.

Beachte: Der Datentyp des Rückgabewertes darf sein: ein arithmetischer Typ (char, int, float, double), eine Struktur, eine Union, ein Zeiger (siehe später) oder void. Nicht erlaubt ist ein Array oder eine Funktion.

Die return-Anweisung

return *ausdruck*;

beendet eine Funktion und gibt den Wert von ausdruck an den aufrufenden Programmteil zurück. Der Ausdruck ausdruck wird in den Datentyp der Funktion umgewandelt.

Eine Funktion kann beliebig viele return-Anweisungen enthalten, oder auch keine. Fehlt eine return-Anweisung, so wird am Ende der Funktion ein implizites return ausgeführt; der Rückgabewert ist dann nicht definiert, d.h. kann ein beliebiges Bitmuster sein.

void Funktionen

C kennt nur ein Unterprogrammkonstrukt. Prozeduren, die keinen Rückgabewert liefern (siehe Pascal oder FORTRAN), entsprechen Funktionen mit Datentyp **void**.

```
#include <stdio.h>

void stars ( void )
{
        printf("**********\n);
}

main ()
{
        stars();
        printf("Hello, world\n");
        stars();
}
```

- Datentyp der Funktion void => kein Rückgabewert
- Parameterliste void => keine Parameter

6.5. Funktionsprototypen

typ funktions_name (parameterliste);

- Prototyp => Deklaration einer Funktion.
- Fehlerüberprüfung durch Compiler.
- Legt Datentyp und Parameterliste fest.
- In der Parameterliste können die Namen der Parameter fehlen und nur ihre Datentypen aufgeführt sein.

Eine Funktion ohne Parameter sollte im Prototyp das Schlüsselwort void (leer, nichtig) an Stelle der Parameterliste haben. Leeres Klammernpaar bedeutet unbekannte Parameterliste => keine Überprüfung durch Compiler.

Ein Prototyp steht

- **ausserhalb** einer Funktion => sichtbar bis Ende File
- bei den Vereinbarungen **innerhalb** einer anderen Funktion => sichtbar innerhalb dieser Funktion
- **kann entfallen**, wenn Funktion vor ihrer ersten Verwendung definiert wird.

Ohne Prototyp und vorangehende Definition wird Datentyp der Funktion int und unbekannte Parameterliste angenommen => Keine Fehlerüberprüfung!

	Variable	Funktion
Deklaration	legt Eigenschaften wie Datentyp fest	legt Eigenschaften von Parametern und Rückgabewert fest
Definition	Legt Eigenschaften fest und alloziert Speicherplatz	legt Parameter und Rückgabewert fest und definiert Programmcode

6.6. Rekursion

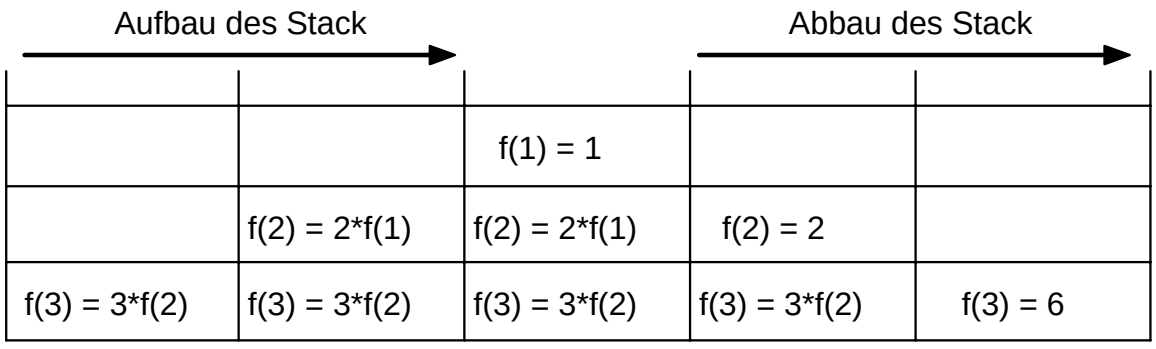
Beispiel aus der Mathematik:

$$n! = \begin{cases} n * (n-1)! & \text{für } n > 0 \\ 1 & \text{für } n == 0 \end{cases}$$

Definition der Fakultät durch sich selbst und einem Anfangswert

Rekursive Funktion ruft sich selbst auf.

```
int fakrek ( int n )
{
      if ( n > 1 ) {
          return n * fakrek(n-1);
      } else {
          return 1;
      }
}
```



Beispiel: Quicksort

```
void qcksort ( int a[], int left, int right)
{
    int l=left; r=right;
    int tmp, x=a[(l+r)/2];

    do {
     while ( a[l] < x ) l++;
     while ( x < a[r] ) r--;
     if ( l <= r ) {
         tmp = a[l];   a[l] = a[r];   a[r] = tmp;
         l++;   r--;
     }
    } while ( l <= r );

    if ( left < r ) qcksort(a, left, r);
    if ( l < right ) qcksort(a, l, right);

}

void QuickSort( int a[], int n )
{
    qcksort(a,0,n);
}
```