12 Funktionen

12.1. Grundlagen

Es wurde bereits die Aufteilung von Programmen in einzelne Module angesprochen. Diese Aufteilung auch in einem Programm zu vollziehen erscheint daher wünschenswert. In allen höheren Programmiersprachen und auch im Befehlsumfang nahezu aller Prozessoren ist diese Möglichkeit in Form von Unterprogrammen realisiert. Man teilt dabei Unterprogramme in so genannte Funktionen und Prozeduren ein. Es gilt also:

- Unterprogramme sind Hilfsmittel, um Problemstellungen in kleine Teilprobleme zerlegen zu können
- sie dienen damit einer strukturierten Programmierung und erhöhen damit die Lesbarkeit, Erweiterbarkeit und Wartbarkeit der Programme
- typische Anwendung für Unterprogramme ist die Erledigung immer wiederkehrender Aufgaben
- für die wichtigsten Aufgaben gibt es bereits Unterprogramme, sie sind in der C-Programmbibliothek enthalten
- für eigene Zwecke kann man natürlich auch eigene Unterprogramme schreiben
- C Programme bestehen typischerweise aus vielen kleinen und nicht aus wenigen großen Unterprogramme

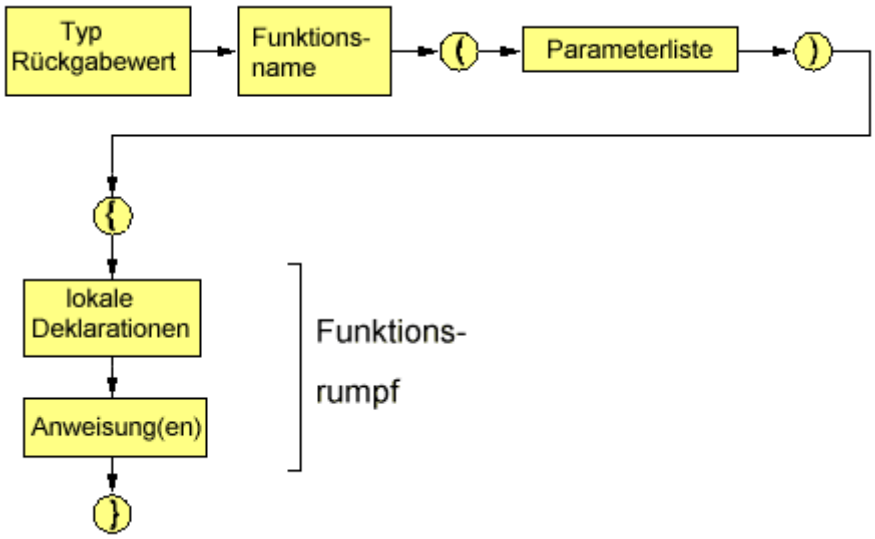
Wobei in C die Begriffe Unterprogramm und Funktion gleichwertig verwendet (synonym) werden. Die Definition eines Unterprogramms erfolgt durch Angabe eines Typs, gefolgt vom Unterprogrammnamen und einer Parameterliste in runden Klammern.

```
int blabla ()
  {
  return (0);
  }
```

Der Aufruf kann dann z. B. als Anweisung `x = blabla();` erfolgen. Unterprogramme sind stets **global**: sie können nur außerhalb jedes anderen Unterprogramms (einschließlich main()) definiert werden, und sind dann aus jeder aufrufbar. Unterprogrammdefinitionen lassen sich demnach in C nicht schachteln, sie werden alle auf einer "Ebene" definiert.

12.2. Aufbau eines Unterprogramms

Die Definition eines Unterprogramms wird syntaktisch beschreiben als:



Eine Unterprogramm hat also einen festgelegten Aufbau (genauere Definition der ersten Zeile siehe unten):

```
Typ Funktionsname(Parameterliste)
  {
    Vereinbarungen
    Anweisungen
    return (Funktionswert)     optional
  }
```

Die runden Klammern bei der Vereinbarung müssen stehen, damit *Funktionsname* zur Funktion wird. Die Parameterliste in den runden Klammern ist optional, d.h. sie muss nur vorhanden sein, wenn der Funktion wirklich Parameter übergeben werden. Andernfalls ist als Platzhalter *void* anzugeben. Die Parameterliste enthält sowohl die Namen als auch die Typdeklarationen der Parameter. Die Gesamtheit der Vereinbarungen und Anweisungen der Funktion selbst nennt man den Funktionskörper ("body"), er muss durch geschweifte Klammern eingeschlossen sein. Die *return*-Anweisung darf an beliebiger Stelle im Funktionskörper stehen, mit ihr erfolgt der Rücksprung in die aufrufende Funktion oder ins Hauptprogramm. Die *return*-Anweisung kann auch ganz fehlen, dann erfolgt der Rücksprung in die aufrufende Funktion beim Erreichen des Funktionsende (der schließenden geschweiften Klammer um den Funktionkörper).
Der Funktionswert hinter der *return*-Anweisung wird meist in runde Klammern eingeschlossen. Dies ist aber nicht notwendig. Fehlt der Funktionswert, so wird an die aufrufende Funktion auch kein Wert zurückgegeben.

Im Prinzip sieht ein C-Programm immer wie das folgende Beispiel aus:

```
void main (void)
  {
  int x, y;
  ...
  blabla ();
  ...
  x = foobar(y);
```

```
  ...
  }

void blabla (void)
  {
  ...
  }

int foo ();
  {
  ...
  }

int foobar (int x)
  {
  ...
  x = 10 * foo();
  ...
  }
```

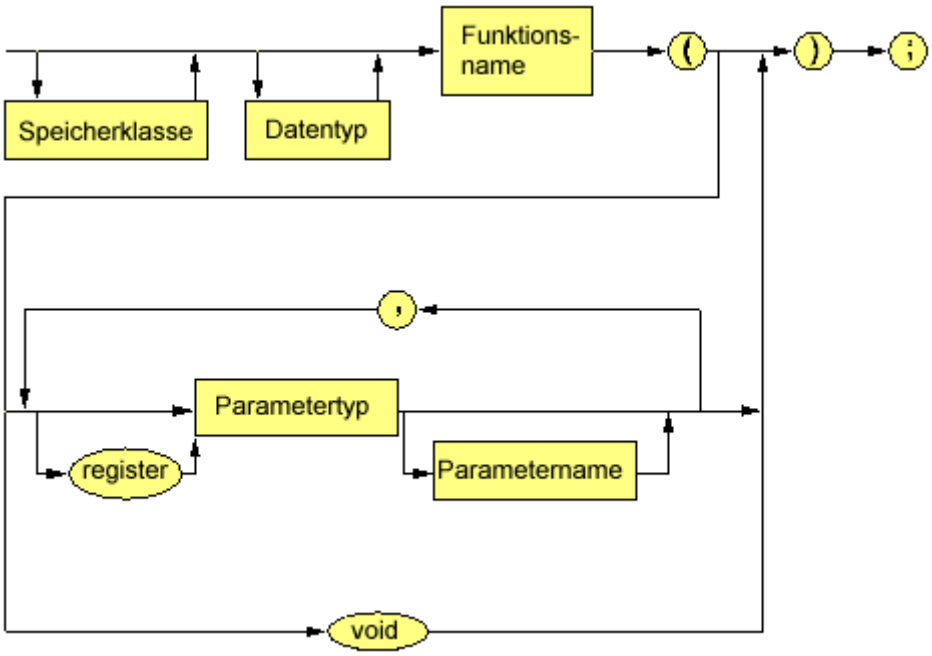
Dabei ergibt sich ein Problem. In main() ist nämlich noch gar nicht bekannt, welche Parameter die Funktionen *blabla(), foo()* und *foobar()* haben, welchen Typ die Parameter besitzen und welche Rückgabewert die Funktion hat. Die Funktionen werden ja erst unterhalb von *main()* definiert. Der C-Compiler nimmt dann *int* als Standardtyp an. Lösen lässt sich das Problem durch eine **Vorwärts-Deklaration** der Funktionen. Dabei werden nur Funktionsname und Parameter angegeben und die Angabe mit Strichpunkt abgeschlossen. Die Funktionsdefinition an anderer Stelle enthält dann auch den Code dazu. Zum Beispiel:

```
void blabla (void);
int foo ();
int foobar(int x);

void main (void)
  ...

  weiter wie oben
```

Man spricht hier von so genannten Funktionsprototypen. Aus den Funktionsprototypen lässt sich auch die Signatur der Funktion bestimmen, also Rückgabetyp, Funktionname und Übergabe- bzw. Transitparameter. Das Syntaxdiagram für die Definition des Funktionsprototypen sieht so aus:



12.3. Funktionsparameter

Werte und Zeiger als Übergabeparameter

Man unterscheidet Übergabe-, Transit und Rückgabeparameter. Funktionen besitzen reine Übergabeparameter und einen Rückgabewert, welcher mit *return* zurückgegeben wird. Prozeduren besitzen Übergabe- und Transitparameter, wobei Übergabeparameter Werte (Values) und Transitparameter Adressen (Referenzen) sind.
Soll eine Funktion mit unterschiedlichen Eingabewerten (= Parametern) aufgerufen werden, kann bei der Vereinbarung eine Liste der Parameter hinter dem Funktions-Namen aufgeführt werden. Bei jedem dieser Parameter wird dessen Typ aufgeführt. Diese formalen Parameter haben einen frei wählbaren Namen, der innerhalb des Anweisungsteils der Funktion gültig ist und wie jede andere Variable verwendet werden kann.

```
int quad (int x)
  {
  return (x * x);
  }
```

Grundsätzlich gilt: **Eine Funktion muss vor ihrer Verwendung deklariert werden.** Es hat sich daher als zweckmäßig erwiesen, zu Beginn eines Programms nicht nur Variablen, sondern auch Funktionen zu deklarieren. Dies geschieht durch den Funktionsnamen mit Parameterliste. Statt des Anweisungsteils in geschweiften Klammern wird nur ein Strichpunkt gesetzt.

```
int quad (int x);
```

Die Funktion mit Anweisungsteil wird dann später im Programm definiert. Solche Prototyp-Deklarationen findet man auch in der zu der Sourcecodedatei (.c) dazugehörenden Headerdatei (.h).

Beim Aufruf der Funktion werden die formalen Parameter durch die aktuellen Parameter (=Argumente) ersetzt (Parameterversorgung der Funktion). Ähnlich wie bei allen anderen Variablen können diese durch ihre expliziten Werte, Konstante oder Variable versorgt werden. Zum Beispiel:

```
y = quad(25);
```

Der Typ des aktuellen Parameters muss natürlich auch den Typ des formalen Parameters entsprechen, dies wird bei C – Programmen geprüft, um sichere Programme zu bekommen.

Beispiel für eine Funktion, Potenzen berechnen. Beachte die richtige Aufteilung des Programms in einzelne Dateien, jede Sourcecodeatei (.c) hat eine dazugehörende Header datei (.h) und die richtige Einbindung der Headerdatei.

```
// potenz.h
// Definition des Funktionsprotyps (Signatur)

#ifndef  POTENZ H
#define  POTENZ H

int power(int,int);

#endif

// main.c

#include <stdio.h>

#ifndef  POTENZ H
#include ”potenz.h”
#endif

int main(void)
  {
  int i;
  for (i = 1; i <= 10; ++i)
      printf("%d %d %d\n", i, power(2,i), power(-3,i));
  exit(0);
  }

// potenz.c

#ifndef  POTENZ H
#include ”potenz.h”
#endif

int power(int base, int n)
  {
  int i, p = 1;
  for (i = 1; i <= n; i++)
    p = p*base;
  return p;
  }
```

Die Unterscheidung zwischen Übergabe- und Transitparameter erfolgt durch die Art der Parameterübergabe.

- Bei Übergabeparametern wird der aktuelle Parameter ausgewertet und das Resultat dem formalen Parameter, der eine lokale Variable darstellt, zugewiesen. Es wird in der Funktion keine Variablendeklaration für den aktuellen Parameter mehr benötigt. Diese Art wird **Werte-Parameter** (call by value) genannt und ist die häufigste Form der Ersetzung. Anstelle des Parameters kann auch ein Ausdruck stehen (z. B.: quad(2*x + 1)). Hier sind nach Rückkehr zum aufrufenden Programmteil alle Änderungen des aktuellen Parameters wieder verloren, es wird nur mit einer Kopie in der Funktion gearbeitet.
- Bei Transitparameter ist der aktuelle Parameter eine Variable. Die Variable ersetzt den formalen Parameter. Diese Art wird **Variablen-Parameter** (call by reference) genannt und muss immer dann verwendet werden, wenn der Parameter ein Resultat der Funktion darstellt. In C wird hier die Adresse der Variablen übergeben. Dies geschieht bei der Funktionsdefinition durch das Voranstellen eines Sternchens (*) vor den Parameternamen, z. B.`int *varpar`.
  Um beim Aufruf der Funktion den Wert herauszutransportieren, wird dem aktuellen Variablenparameter ein &-Zeichen vorangestellt. (z. B. foo(&x)). Zur Wiederholung, es gilt die Regel:
  - Adressoperator:
    & liefert die Adresse einer Variablen zurück. Beispiel: `int a;`: Adresse von a ist `&a`
  - Inhaltsoperator:
    * gibt den Inhalt einer Speicherstelle mit einer bestimmten Adresse an. Beispiel: `int a;`: a = *(&a)

  Beispiel: Funktion, die die Inhalte zweier Variablen vertauscht.

```
#include <stdio.h>
// Funktionsprototypen
void tausche(int iZahl1, int iZahl2);
void swap(int *piZahl1, int *piZahl2);

int main(void)
{
// Lokale Variablen
    int iZahl1,iZahl2;

     iZahl1 = 10;
     iZahl2 = 20;
     printf("Zu vertauschende Zahlen\n");
     printf("Zahl1:  %d  Zahl2:  %d\n",iZahl1,iZahl2);
     tausche(iZahl1,iZahl2);

     printf("Nach Funktion TAUSCHE\n");
     printf("Zahl1: %d  Zahl2: %d\n",iZahl1,iZahl2);

     swap(&iZahl1,&iZahl2);
     printf("Nach Funktion SWAP\n");
     printf("Zahl1: %d  Zahl2: %d\n",iZahl1,iZahl2);
```

```
     return 0;
}

// Funktionen
// tausche vertauscht Werte nur in der Funktion CALL BY VALUE
void tausche(int iZahl1, int iZahl2)
{
     int iAblage;

     iAblage = iZahl1;
     iZahl1 = iZahl2;
     iZahl2 = iAblage;
    printf("Funktion TAUSCHE\n");
     printf("Zahl1:  %d Zahl2:  %d\n",iZahl1,iZahl2);
}

// swap vertausche Werte in der Funktion und auch außerhalb
// CALL BY REFERENCE
void swap(int *piZahl1, int *piZahl2)
{
     int iAblage;
// int* piAblage;

     iAblage = *piZahl1;
// piAblage = piZahl1;
     *piZahl1 = *piZahl2;
// piZahl1 = piZahl2;
     *piZahl2 = iAblage;
// piZahl2 = iAblage;
     printf("Funktion SWAP\n");
     printf("Zahl1:  %d Zahl2:  %d\n",*piZahl1,*piZahl2);
}
```

Argumente vom Zeigertyp eröffnen die Möglichkeit, aus einer Funktion heraus Objekte in der aufrufenden Funktion anzusprechen und zu verändern. Die Funktion `swap()` im obigen Beispiel zeigt dieses.

Felder als Übergabeparameter

Eindimensionale Felder

Da der Name eines Feldes ein Zeiger auf das erste Element des Feldes ist (mit anderen Worten: die Adresse des ersten Elements ist), wird bei Feldern immer eine Adresse übergeben. Das bedeutet, dass die Funktion immer mit dem Originalfeld arbeitet. Es findet keine Feldgrenzenüberprüfung beim Funktionsaufruf statt. Für die Vereinbarung der Aktualparameter hat man zwei Möglichkeiten:

1. Felddeklaration
   Intern wird dann eine Typumwandlung nach Zeiger vorgenommen.
2. Zeiger

Beispiel: Feldelemente aufaddieren.

```
#include <stdio.h>
```

```
// Indexschreibweise
void sum(int iaFeld[5],int); // void sum(int iaFeld[],int);
// Pointerschreibweise
void sum(int*, int);


int main(void)
  {
  int iaFeld[5] ={0,2,3,4,5};
  int n = 5, iSumme;

  printf("summe = %d\n",iSumme);
// Indexschreibweise
  iSumme = sum(&iaFeld[0],n);
// Pointerschreibweise
  iSumme = sum(iaFeld, n);
  printf("summe = %d\n",iSumme);
  exit(0);
  }


// Indexschreibweise
void sum(int iaFeld[5], int n) // void sum(int iaFeld[], int n)
  {
  int iSumme = 0;
  while (n-- >1)
    iSumme = iSumme + a[n];
  return iSumme;
  }
// Pointerschreibweise
void sum(int* iaFeld, int n)
  {
  int iSumme = 0;
  while (n-- >1)
    iSumme = iSumme + *(iaFeld + n*4);
  return iSumme;
  }
```

Es ist nicht nötig, bei der Funktionsdefinition die Arraylänge festzulegen, da der Compiler keine Längenprüfung durchführt. Daher ist es ratsam die Länge des Arrays wie im Beispiel als zweiten Parameter anzugeben. Arrays werden stets per Referenzaufruf übergeben.

Zweidimensionale Felder

Die Übergabe von zweidimensionalen Feldern an eine Funktion erfolgt nicht anders als bei eindimensionalen Feldern. Die aufgerufene Funktion erhält auch hier nicht das gesamte Feld, sondern nur seine Anfangsadresse bzw. die Adresse eines bestimmten Teils des Feldes. Man hat aber allerdings bei der Angabe der Übergabeparameter außer des ersten die Größe der zweiten Dimension explizit anzugeben, da diese für die Berechnung der Speicherposition benötigt wird.

```
void showMatrix(char a[][5]); //auch void showMatrix(char** a);

void showMatrix(char a[][5]) // auch void showMatrix(char** a)

  {
```

```
  while (n-- >1)
    a[0][0] += a[n][n];
  }
```

12.4. Der exit-Status

Wenn ein Programm oder ein Unterprogramm seinen Ablauf beendet hat, sollte es dem System bzw. den aufrufenden Programmteil mitteilen, auf welche Weise ("alles OK", "Fehler aufgetreten", etc.) diese Beendigung eingetreten ist. Dazu dient der exit-Status bzw. die return-Anweisung. Der exit-Status bzw. die return-Anweisung ist eine ganze Zahl, die im System unmittelbar nach Programmbeendigung in irgendeiner Form zurückbleibt bzw. an den aufrufenden Programmteil zurückgegeben wird. Die Beachtung des exit-Status wird spätestens dann wichtig, wenn Programme sich gegenseitig aufrufen und abhängig vom Ausgang Entscheidungen treffen müssen. Mit dem exit-Status wird nur das Hauptprogramm beendet.

Der exit-Status eines Programms ist der Rückgabewert der Funktion `main`, also i.A. der Wert der in `main` mit `return` zurückgeliefert wird. Zur sofortigen Beendigung eines Programms bei einem kritischen Fehler - wenn etwa eine Eingabedatei nicht geöffnet werden konnte - und zum definierten Setzen des exit-Status verwendet man die Bibliotheksfunktion

```
void exit(int status); .
```

Nach Konvention bedeutet der Wert 0, dass alles OK ist, während eine von 0 verschiedene Zahl einen Fehlerindikator darstellen kann. Es soll nochmals darauf hingewiesen werde, dass auch bei Prozeduren ein Fehlercode mit der return-Anweisung zurückgegeben werden muss.

Alle Fehlercodes sollen bzw. müssen im aufrufenden Programmteil abgefragt und darauf entsprechend reagiert werden.

Ein Beispiel:

```
int funktion(int, int);

int  main(void)
  {
  ...
  iFehler = funktion(iWert1, iWert2);
  if(iFehler == 1)
// Programm wird sofort beendet, gravierender Fehler
     exit(1);
  else if (iFehler == 333)
  ...
// Rückgabewert des Hauptprogramms an das System
  return 0;
  }

int funktion(int iWert1, int iWert2)
{
   int iFehler;
   ...
```

```
   return (iFehler);
}
```

12.5 Seiteneffekte

Als Seiteneffekt werden Zuweisungen an globale Variable bezeichnet. Solche Effekte tragen meist zur Verschleierung der Programmstruktur bei und erschweren gewöhnlich die Verifikation und Änderbarkeit des Programms. So sind sie oft ein Quell versteckter und seltsamer Fehler. Globale Variablen dürfen in Unterprogrammen nie verwendet werden, da damit die Wiederverwendbarkeit dieser erschwert bzw. unmöglich gemacht wird.

12.6. Kommandozeilenparameter

Unter Verwendung von Kommandozeilen-Parameter versteht man die Übergabe von Argumenten an die Funktion `main()` beim Aufruf des Programms. Normalerweise sieht die Signatur der Hauptfunktion so aus:

```
int main(int argc, char *argv[])
```

Dabei gibt `argc` (Argument Count) die Zahl der Argumente an, und `argv` ist ein Array, in dem diese Argumente in Form von Strings vorliegen. `argv` (Argument Value) ist ein Array aus Stringpointern Die Bezeichnungen `argc` und `argv` sind Konvention, aber syntaktisch aber nicht vorgeschrieben. Beispiel: ein Programm heißt *foo* und wird aufgerufen mit

```
foo myfile 1 3.8
```

dann sind die Argumente von `main` folgendermaßen belegt:

`argc` : 4 (Zahl der Argumente, und zwar:)

`argv[0]` : `"/...<`*Pfad*`>.../foo"`

`argv[1]` : `"myfile"`

`argv[2]` : `"1"`

`argv[3]` : `"3.8"`

`argv[4]` : `NULL`

Um Strings in Zahlen zu konvertieren, stehen die z.B. die Funktionen `int atoi(char*)` und `double atof(char*)` (deklariert in `<stdlib.h>`) zur Verfügung.
Ein Programmbeispiel:

```
#include <stdio.h>
#include <stdlib.h>

int  main(int argc, char *argv[])
  {
  int  i;
```

```
  double  df;

  if(argc != 3)
    {
    printf("usage: %s <i>  <df>\n", argv[0]);
    return 1;
    }
  i  = atoi(argv[1]);
  df = atof(argv[2]);
  printf("  i:  %d      df:  %f\n", i, df);
  return 0;
  }
```