7 Funktionen

Allgemein

Benennung:

- Prozedur: Funktion ohne Rückgabewert (in C/Java void)
- Funktion: nicht Klassen zugeordnet mit Rückgabewert
- Methode: Funktion die einer Klasse zugeordnet wird. In JAVA also immer, in C (keine OOP) also nie.

Warum Funktionen:

- Funktionen steigern die Lesbarkeit eines Programmes (Information Hiding).
- durch die Wiederverwendung (Reuseability) von Code steigt die Fehlererkennung und damit die Fehlervermeidung.
- Zugriff auf Variable nur über Funktionen (Datenabstraktion) --> OOP.
- Funktionen erleichtern die Wartbarkeit einer Software.

→ Software soll kurz gehalten werden. Es sollen möglichst viele Funktionen verwendet werden.

Definition/Deklaration

```
[Spezifizierer] Rückgabetyp Funktionsname(Parameter) {
   /* Anweisungsblock mit Anweisungen */
}
```

Wenn eine Funktion keinen Rückgabewert haben soll, dann muss als Rückgabetyp *void* angegeben werden. Wenn an eine Funktion keine Parameter übergeben werden sollen, dann wird die Klammer leer gelassen.

Wird in der Definition der Funktion als Parameter *void* angegeben, dann darf die Funktion nur ohne Parameter aufgerufen werden. Ansonsten können beliebig Parameter mitgegeben werden (wenn auch nichts damit gemacht werden kann).

Wo werden Funktionen eingebaut:

- In der c-Datei auf gleicher Ebene wie die Main-Funktion. Dadurch kann die Funktion von innerhalb der Main-Funktion verwendet werden.
- Innerhalb einer anderen Funktion. Diese Funktionen können nur in dieser Funktion oder innerhalb weiterer Funktionen in dieser Funktion aufgerufen werden.

Damit die Funktionen aufgerufen werden können müssen sie allerdings vor/über dem Aufruf bekannt sein. Entweder werden sie direkt vor ihrem Aufruf implementiert oder sie werden vor ihrem Aufruf nur deklariert (=Prototyp) und an anderer Stelle implementiert.

Korrekte und saubere Lösung: sämtliche Funktionen werden am Beginn deklariert (und nicht implementiert). Die Deklaration ist der Kopf der Funktion gefolgt von einem Semikolon (Prototyp):

```
int add(int a, int b);      // Prototyp / Deklaration im Kopf

int main(void) { ... }

int add(int a, int b) { .... }  // Implementierung der Funktion
```

Diese separate Deklaration spielt eine ganz wichtige Rolle für die weitere Verwendung von Funktionen in anderen c-Modulen/Programmen und wird daher oft auch in die h-Datei ausgelagert. In den Prototypen ist es erlaubt den Variablennamen wegzulassen (also gilt für obiges Beispiel auch `int add(int, int)`.

Sichtbarkeiten / Geltungsbereich von Variablen

Lokale Variable

In Blöcken (geschwungene Klammern) definierte Variable sind bis zum Ende des Blocks sichtbar, dann wird der Speicherplatz freigegeben. Solche Variable sind nur "lokal" in diesem Bereich verfügbar. Wird dieser Code erneut ausgeführt (z.B. weil in einer Schleife), dann ist daher der Wert aus der letzten Ausführung nicht mehr verfügbar.

```
int i = 10;
printf("%d\n", i);          // 10
{
    int i = 11;
    printf("%d\n", i);      // 11
}
printf("%d\n", i);          // 10
```

Globale Variable

Alles was direkt in einer Datei ohne Schlüsselwort definiert wird, kann global verwendet werden. Das trifft auf Variablen genauso wie auf Funktionen zu. In einer zweiten Datei kann auf Variable nur nicht zugegriffen werden,

Grundsätzlich gilt auch für C: **so lokal wie möglich, so global wie nötig**

Lokal STATIC

Lokal Statische Variable behalten ihren Wert auch bei einer Wiederkehr in den lokalen Bereich. Also wie andere lokale Variable auch, kann auf diese nicht ausserhalb ihres lokalen Bereichs zugegriffen werden. Bei Wiedereintritt in diesen lokalen Bereich, ist der Wert der statischen Variable der Selbe, wie nach der letzten Ausführung.

```
void sum(int summand) {
  static int i = 0;     // Initialisierung, wenn statisch, dann nur beim ersten Aufruf relevant
  i += summand;
  printf("Wert der Summe : %d\n",i);
}

int main(void) {
   sum(3);
   sum(4);
   sum(5);
   return 0;
}
```

Erzeugt eine Ausgabe 3-7-12. OBWOHL hier in der Methode mit i = 1 intialisiert wird! Diese Initialisierung der statischen Variable wird nur einmal aufgerufen - beim Ersten Mal.

Global STATIC

Wenn Variable ausserhalb von Funktionen (auch von main) als static definiert, dann sind diese Variable nicht mehr im gesamten Programm sichtbar sondern nur mehr in der Datei in der sie sich befinden.

Call-by-Value <-> Call-by-Reference

In C gibt's keine Objekte zum Kapseln von Daten in einen einzelnen Datentypen. Um mehrere Daten verändern zu können kann eine Funktion einen Wert als Referenz (Adresse) übernehmen:

```
void inc(int* val) {        // Variable die in der Funktion verändert werden können werden mit einem * angeführt
   (*val)++;                // Zugriffe auf den Wert der Variable ebenfalls über einen *
}

int main(void) {
   int num = 0;

   inc(&num);               // Im Aufruf muss eine solche Variable mit einem & übergeben werden
   inc(&num);
   inc(&num);

   inc(&num);
```

```
    inc(&num);
    inc(&num);

    printf("%d", num);
    return 0;
}
```

Übergabe der Adresse an die Funktion. Die Funktion manipuliert die Daten am Speicherort der Variable und verändert damit den Wert der Variable selbst (siehe scanf()). Weitergehende Erklärung der Operatoren * und & im Kapitel **Zeiger**.

Felder

Das Zurückgeben von Feldern aus Funktionen ist problematisch. In C klappt nicht: `int[] sort(int[] arr);` Die *Feldvariable* ist ja eigentlich ein Zeiger auf ein Feld. Wird dieser zurückgegeben klappt das zwar, das Feld selber wird, da's lokal ist, gelöscht. Es gibt folgende Möglichkeiten:

a) statische Variable

```
const char* func() {
  static char* str = "Hallo ich bin's";
  return str;
}
```

Durch die static-Definition wird die Variable behalten und nicht am Ende der Funktion wieder gelöscht. Sie wird ja für den evtl. nächsten Aufruf der Funktion behalten.

Vorteil: sehr einfache schnelle Lösung. Nachteil: Maximale Größe des Felds wird in der Funktion festgelegt. Die Funktion kann nur für ein Feld verwendet werden, Sollen weitere Felder erstellt werden, dann wird immer der gleiche Speicher benutzt (Daten werden überschrieben).

b) Zielfeld mitgeben

```
char* func(char* out, int len) {
  // out ist ein "Ausgabeparameter"

  // out verändern...

  if (<Fehler>) return NULL;
  else return out;
}

// Verwendung:

char x[50];

if (func(x, 50) != NULL) {
  puts(x);
}
```

so machen's viele Standard-Funktionen. Vorteil: Größe des Felds wird außerhalb der Funktion festgelegt.

c) Speicher reservieren

Siehe dazu Kapitel Zeiger.

```
char* func() {
  char* str = malloc(50);

  // ...

  return str;

}
```

```
// Verwendung:
char* x = func();

if (x != NULL) {
  puts(x);

  free(x);
}
```

Vorteil: Größe des Felds muss vorher nicht bekannt sein. Nachteil: Speicher muss wieder freigegeben werden. Ansonsten Gefahr von Speicherlecks.

Verteilung auf mehrere Dateien

Häufig ist es sinnvoll Code auf mehrere Dateien (nennt man Module) zu verteilen:

- Übersicht, Größe, Strukturierung
- Verteilte Programmierung im Team

Beispiel:

main-Module **main.c**:

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

int main()
{
    otherModule();
    printf("... Here Main Speaking ... \n");
    return 0;
}
```

zweites Modul **modul.c**:

```
#include <stdio.h>

void otherModule() {
    printf("... Here modul.c speaking ...\n");
}
```

Bemerkenswert: in keinem der beiden Module ist eine Referenz (Dateiname, externe Deklaration zu finden). In der Kommandozeile werden die beiden Module nun zu Objekt-Dateien kompliert:

```
...>gcc -c main.c
...>gcc -c modul.c
```

damit werden die Dateien main.o und modul.o erstellt. Im Folgenden werden die Objekt-Dateien miteinander verlinkt:

```
...>gcc -o multimod.exe main.o modul.o
```

damit wird das Programm multimod.exe erstellt. Der Aufruf zeigt:

```
...>multimod.exe
... Here modul.c speaking ...
... Here Main Speaking ...
```

Schnittstellenbeschreibung - Header-Dateien

Um Module miteinander verlinken zu können beschreibt man die Schnittstellen in h-Dateien (Header-Dateien). Dazu ein Beispiel:

main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "funct.h"      // include kopiert Inhalt von funct.h an diese Stelle

int main()
{
    funct();            // Aufruf der Funktion funct(), deklariert in funct.h
    funct_var = 3;      // Verwenden der funct-Variable funct_var
    return 0;
}
```

funct.h:

```
#ifndef FUNCT_H_INCLUDED        // wenn FUNCT_H_INCLUDED schon definiert, dann mach das hier nicht
#define FUNCT_H_INCLUDED        // Definieren von FUNCT_H_INCLUDED --> Datei kann nur einmal eingebunden werden.

void funct(void);               // Prototyp
extern int funct_var;           // Deklaration/keine Definition

#endif // FUNCT_H_INCLUDED
```

Die Präprozessordirektiven sorgen dafür das eventuelle Definitionen in den Header-Dateien nur einmal eingebaut werden. Ansonsten gibt's Fehler.

funct.c:

```
#include <stdio.h>
#include "funct.h"              // sinnvoll aber nicht notwendig

int funct_var;                  // Definition der Variable

void funct(void) {              // Definition der Funktion
    printf("Hello from Funct");
}
```

Vorgangsweise für die Erstellung solcher Projekte:

- Implementierung der Funktionen in der c-Datei schreiben
- Überlegen, was von dieser Implementierung von aussen sichtbar sein soll.
- Das kommt als Deklaration (Prototyp) in die h-Datei.
- Für alles in der c-Datei, das woanders herkommt, gibt es einen #include, der die notwendige h-Datei einbindet.
- Werden in der h-Datei Dinge von woanders benutzt, dann enthält die h-Datei einen entsprechenden #include.
- Jede Datei, sowohl h-Datei als auch c-Datei ist in sich vollständig. Werden dort Dinge benutzt, dann müssen diese vor der Verwendung deklariert worden sein. Wie und wo diese Deklaration herkommt, ist dabei zweitrangig. Es kann sein, dass die Deklaration vor der Verwendung steht, es kann aber auch sein, dass die Deklaration über einen weiteren Include mit aufgenommen wird.

Sind von Modulen nur Objekt-Dateien verfügbar (siehe Standard-Bibliotheken), dann werden h-Dateien benötigt um die Schnittstellen in diesen Modulen offenzulegen.

Spezifizierer

extern

Wenn Sie bei der Deklaration einer Funktion die Speicherklasse nicht angeben, ist diese automatisch mit extern gesetzt. Solche Funktionen können sich auch in einer anderen Quelldatei befinden. Dann speziell empfiehlt es sich, dieses Schlüsselwort zu verwenden (auch wenn dies nicht nötig wäre). Dieser Hinweis kann hilfreich für den Programmierer sein, weil er sofort weiß, worum es sich handelt. extern zu setzen dient also nicht der Verbesserung bzw. Optimierung des Quellcodes, sondern ist ein Hinweis für dessen Leser.

static

Wenn Sie einer Funktion das Schlüsselwort static zuweisen, können Sie diese Funktion nur innerhalb der Datei nutzen, in der sie definiert wurde. Es ist somit das Gegenteil des Schlüsselworts extern.

volatile

volatile ist zwar keine Speicherklasse, sollte aber hier trotzdem erwähnt werden. Mit volatile verhindern Sie (analog zu Variablen), dass der Compiler den Quellcode optimiert und die Funktion immer wieder neu aus dem Hauptspeicher gelesen werden muss.