1. Übergabe einer variablen Anzahl von Parametern an C-Funktionen

1.1 Vorbemerkung:

In vielen Programmiersprachen kann man eine einmal definierte Funktion (oder Prozedur) nur mit einer ganz bestimmten, festen Anzahl von Parametern aufrufen.

Hat man zum Beispiel die Funktion wie folgt definiert:

```
int f1(int i, int j, int k)
{  ... /* Code der etwas tut ... */
}
```

dann kann man die Funktion **`f1()`** in vielen anderen Programmiersprachen als C nicht mit z. B. nur zwei Parametern aufrufen:

```
int main()
{  ...
   f1(2,3);    /* Wäre fehlerhaft, aber in C erlaubt! */
   f1(2,3,4);  /* So ist's natürlich immer ok.        */
   ...
}
```

In C sind dagegen beide obigen Aufrufe der Funktion **`f1()`** zulässig.

Der Grund dafür und die Grundvoraussetzung, warum so etwas in C funktioniert, ist, wie in C Parameter an eine Funktion auf dem Stack übergeben werden:

Bei einem Aufruf **`f1(2,3)`**; in der **`main()`** Funktion erzeugt der Compiler folgenden Code:

```
8:    f1(2,3);
00401038  push     3
0040103A  push     2
0040103C  call     @ILT+5(_f1) (0040100a)
00401041  add      esp,8
```

- Die aufrufende Funktion (hier im Bsp. die **`main()`** Funktion) präpariert den Stack.
- Dann wird die Funktion **`f1()`** aufgerufen und erhält die Kontrolle.
- Wenn die Funktion **`f1()`** fertig ist (**`return`**), dann erhält die aufrufende Funktion wieder die Kontrolle und entfernt die Aufrufparameter wieder vom Stack.

Letzteres ist der entscheidende Punkt: In C bereinigt nicht die aufgerufene Funktion den Stack. Wäre das der Fall, würde **`f1()`** immer genau 3 **`int`**-Werte vom Stack entfernen, entsprechend der Definition der Funktion **`f1()`**. Dadurch käme der Stack völlig durcheinander, wenn die **`main()`** Funktion statt 3 Werten nur 2 **`int`**-Werte auf dem Stack ablegen würde.

Daß in C immer die aufrufende Funktion den Stack bereinigt, kann man salopp auch so als griffigen Merksatz formulieren: "In C putzt jeder seinen Dreck selber weg!" Dies ist die Grundvoraussetzung dafür, daß in C Funktionsaufrufe mit einer variablen Zahl von Parametern überhaupt möglich sind.

Bemerkung:

- Nach ANSI-Standard legen die meisten C-Compiler die Werte von rechts nach links auf dem Stack ab. Im obigen Beispiel-Aufruf **`f1(2,3)`**; wird mit dem ersten **`push`** Befehl also die **`3`** auf dem Stack abgelegt und erst danach die **`2`**.
- Wie Funktionsparameter in C an die Funktion übergeben werden, und wie nach Beenden der aufgerufenen Funktion die aufrufende Funktion die Werte wieder vom Stack entfernt, ist eine wichtige Kenntnis, wenn C-Code mit schnellen Assembler-Routinen kombiniert werden soll, wie dies heute z.B. häufig bei zeitkritischen Routinen in Microcontollern gemacht wird.

1.2 Wie könnte der Code der aufgerufenen Funktion aussehen?

Das folgende Beispiel dient nur dem Verständnis des nach ANSI genormten Mechanismus <stdarg.h> zur Implementierung von Funktionen, die mit unterschiedlich vielen Parametern

aufgerufen werden. Dazu wird zunächst ein Beispiel gegeben, das den Mechanismus nach ANSI noch nicht verwendet.

Da der Programmier der Funktion **f1()** ja nicht weiß, wieviele Parameter später bei einem Aufruf an seine Funktion übergeben werden, kann man genauso, wie es später der ANSI-Mechanismus vom Prinzip her auch macht, mit einem Pointer **p** arbeiten, der der Reihe nach auf die Parameter auf dem Stack zeigt:

```
#include <stdio.h>
void f1();    /* Das ist anders als: void f1(void); */

int main()
{   f1( 3, 123, 456, 789);  /* Erster Parameter ist */
    f1( 0);                 /* bei allen Aufrufen   */
    f1( 1, 97);             /* ein "Zähler".        */
    return 0;
}

void f1(int v)
{   int *p;
    if (v)
    {   p = &v; /* Der Pointer zeigt auf den Zähler */
        p++;    /* p zeigt auf den ersten unbekannten
                                          Parameter */
        /* Schleife: */
        /* p zeigt immer auf den nächsten Parameter */
        while(v--) printf("%d ",*p++);
        printf("\n");
    }
}
```

1.3 Die Verwendunf des ANSI-genormten Mechanismus mittels <stdarg.h>

Die ANSI-Norm für C stellt dem Programmierer einen rezeptartig anzuwendenden Mechanismus zur Verfügung, mit dem er im Falle einer variablen Zahl von Funktions-Aufruf-Parametern einen nach dem anderen vom Stack holen kann, ohne sich um den Aufbau des Stacks zu kümmern und ohne spezielles Wissen zu benötigen.

Unter Verwendung des genormten Mechanismus, hinter dem sich letztlich drei Makros verbergen, sieht obiges Programm wie folgt aus. Dabei besteht das "Kochrezept" aus sechs Schritten:

```
#include <stdio.h>
#include <stdarg.h>     /* Step 1: Makros einbinden */
void f1(int zaehler, ...);  /* Step 2: Die Ellipse  */

int main()
{   f1( 3, 123, 456, 789);  /* Erster Parameter ist */
    f1( 0);                 /* bei allen Aufrufen   */
    f1( 1, 97);             /* ein "Zähler".        */
    return 0;
}
```

```
void f1(int zaehler, ...)        /* Step 2: Die Ellipse  */
{   va_list ap;  /* Step 3: Eine Variable vom Typ va_list */
    int arg ;

    if (zaehler)
    {   /* Step 4: va_start: va_list-Zeiger initialisier. */
        va_start(ap, zaehler);

        while(zaehler--)
        {   /* Step 5: va_arg-Schleife holt nächstes Argu-
                       ment vom Stack */
            arg = va_arg(ap,int);
            printf("%d ",arg);
        }

        printf("\n");

        /* Step 6: Aufräumungsarbeiten: va_end ausführen  */
        va_end(ap);
    }
}
```

Im Schritt 1 werden mittels **#include <stdarg.h>** die erforderlichen Typdefinitionen und Makros eingebunden. Übrigens: Die Makros und Typen selbst sind nicht in der ANSI-Norm festgelegt. Das heißt, daß sich bei jedem Compilerhersteller etwas anderes dahinter verbergen kann. Z.B. ist der Typ **va_list** bei manchen Compilerherstellern als **(char *)** implementiert, bei anderen als **(void *)**.

Im Schritt 2 wird bei der Deklaration bzw. Definition der aufzurufenden Funktion **f1()** mit den drei Punkten **...** angedeutet, daß nach einem ersten Parameter (von beliebigem Typ) weitere Parameter (von beliebigem Typ) folgen können. Die drei Punkte **...** tragen den Namen Ellipse.

Im Schritt 3 wird eine Variable vom Typ **va_list** vereinbart. Dieser Typ ist in der Header-Datei **<stdarg.h>** deklariert. Der Name der Variable ist frei wählbar. Meist wird sie **ap** genannt (argument pointer). Diese Variable ist letztlich nichts anderes als unser Zeiger **p** aus dem vorhergehenden Beispiel. Dieser Zeiger zeigt später der Reihe nach auf alle Argumente auf dem Stack.

Im Schritt 4 wird dieser Zeiger **ap** so initialisiert, daß er *hinter* das allererste Argument auf dem Stack zeigt. Damit zeigt er auf das erste unbekannte Argument des Funktionsaufrufes. Das verwendete Makro trägt den von der ANSI-Norm festgelegten Namen **va_start()**.

Im Schritt 5 werden vom Makro **va_arg()** gleich zwei Dinge auf einmal erledigt: Das Makro holt sich den nächsten Wert mit dem angegebenen, frei wählbaren Typ vom Stack und erhöht aber auch gleichzeitig den Zeiger **ap** um so viele Bytes, wie der angegebene Typ tatsächlich ausmacht. Damit zeigt der Zeiger **ap** bereits auf das nächste unbekannte Argument auf dem Stack. (Bemerkung: Selbstverständlich berücksichtigt das Makro va_arg(), daß häufig der nächste Wert auf dem Stack erst bei einer z.B. 4-Byte-Grenze zu finden ist.)

Obwohl der Schritt 6 bei vielen Implementierungen ein leeres Makro **va_end()** aufruft, das gar nichts erledigt, soll man es nach ANSI-Norm aus Portabilitätsgründen aufrufen. Es gibt dem Compilerhersteller die Möglichkeit, bestimmte Aufräumungsarbeiten nach abgelaufenem Mechanismus durchzuführen, falls erforderlich. Häufig setzt das Makro den Zeiger **ap** sicherhaitshalber nur auf **NULL**.