Wulf Alex • Gerhard Bernör

UNIX, C und Internet

Moderne Datenverarbeitung in Wissenschaft und Technik

Stand: 9. Januar 2005 (Test)

Springer
Berlin Heidelberg New York
Hong Kong London
Milan Paris Tokyo

Korrekturen und Ergänzungen:
`http://www.alex-weingarten.de/debian/`

0.1 Funktionen

0.1.1 Aufbau und Deklaration

In C ist eine **Funktion** eine abgeschlossene Programmeinheit, die mit der Außenwelt über einen Eingang und wenige Ausgänge – gegebenenfalls noch Notausgänge – verbunden ist. Hauptprogramm, Unterprogramme, Subroutinen, Prozeduren usw. sind in C allesamt Funktionen. Eine Funktion ist die kleinste kompilierbare Einheit (nicht: ausführbare Einheit, das ist ein Programm), nämlich dann, wenn sie zugleich allein in einem File steht. Mit weniger als einer Funktion kann der Compiler nichts anfangen.

Da die **Definitionen von Funktionen** nicht geschachtelt werden dürfen (wohl aber ihre Aufrufe), gelten Funktionen grundsätzlich global. In einem C-Programm stehen alle Funktionen einschließlich `main()` auf gleicher Stufe. Das ist ein wesentlicher Unterschied zu PASCAL, wo Funktionen innerhalb von Unterprogrammen definiert werden dürfen. In C gibt es zu einer Funktion keine übergeordnete Funktion, deren Variable in der untergeordneten Funktion gültig sind.

Eine Funktion übernimmt von der aufrufenden Anweisung einen festgelegten Satz von Argumenten oder Parametern, tut etwas und gibt keinen oder genau einen Wert an die aufrufende Anweisung zurück.

Vor dem ersten Aufruf einer Funktion muß ihr Typ (d. h. der Typ ihres Rückgabewertes) bekannt sein. Andernfalls nimmt der Compiler den Standardtyp `int` an. Entsprechend dem ANSI-Vorschlag bürgert es sich zunehmend ein, Funktionen durch ausführliche **Prototypen** vor Beginn der Funktion `main()` zu deklarieren:

```
/* Beispiel fuer Funktionsprototyp */

float multipl(float x, float y);     /* Prototyp */

/* es reicht auch: float multipl(float, float); */
/* frueher nach K+R: float multipl(); */

int main()
{
float a, b, z;
.
.
z = multipl(a, b);                /* Funktionsaufruf */
.
.
}

float multipl(float x, float y)   /* F'definition */
{
return (x * y);
}
```

Quelle 0.1 . C-Programm mit Funktionsprototyp

Durch die Angabe der Typen der Funktion und ihrer Argumente zu Beginn des Programms herrscht sofort Klarheit. Die Namen der Parameter sind unerheblich; Anzahl, Typ und Reihenfolge sind wesentlich. Noch nicht alle Compiler unterstützen die Angabe der Argumenttypen. Auch den Standardtyp `int` sollte man deklarieren, um zu verdeutlichen, daß man ihn nicht vergessen hat. Änderungen werden erleichtert.

0.1.2 Pointer auf Funktionen

Der **Name** einer Funktion ohne die beiden runden Klammern ist der Pointer auf ihren Eingang (entry point). Damit kann ein Funktionsname überall verwendet werden, wo Pointer zulässig sind. Insbesondere kann er als Argument einer weiteren Funktion dienen. In funktionalen Programmiersprachen ist die Möglichkeit, Funktionen als Argumente höherer Funktionen zu verwenden, noch weiter entwickelt. Arrays von Funktionen sind nicht zulässig, wohl aber Arrays von Pointern auf Funktionen, siehe Programm **??** auf Seite **??**.

Makros (`#define ...`) sind keine Funktionen, infolgedessen gibt es auch keine Pointer auf Makros. Zu Makros siehe Abschnitt **??** *Präprozessor* auf Seite **??**.

0.1.3 Parameterübergabe

Um einer Funktion die Argumente oder Parameter zu übermitteln, gibt es mehrere Wege. Grundsätzlich müssen in der Funktion die entsprechenden Variablen als Platzhalter oder **formale Parameter** vorkommen und deklariert sein. Im Aufruf der Funktion kommt der gleiche Satz von Variablen – gegebenenfalls unter anderem Namen – mit jeweils aktuellen Werten vor; sie werden als **aktuelle Parameter** oder Argumente bezeichnet. Die Schnittstelle von Programm und Funktion muß zusammenpassen wie Stecker und Kupplung einer elektrischen Verbindung, d. h. die Liste der aktuellen Parameter muß mit der Liste der formalen Parameter nach Anzahl, Reihenfolge und Typ der Parameter übereinstimmen.

Bei der **Wertübergabe** (call by value) wird der Funktion eine Kopie der aktuellen Parameter des aufrufenden Programmes übergeben. Daraus folgt, daß die Funktion die aktuellen Parameter des aufrufenden Programmes nicht verändern kann.

Bei der **Adressübergabe** (call by reference) werden der Funktion die Speicheradressen der aktuellen Parameter des aufrufenden Programmes übergeben. Die Funktion kann daher die Werte der Parameter mit Wirkung für das aufrufende Programm verändern. Sie arbeitet mit den Originalen der Parameter. Das kann erwünscht sein oder auch nicht. Bei beiden Mechanismen werden die Parameter vollständig ausgerechnet, ehe die Funktion betreten wird.

Wie die Parameterübergabe in C, FORTRAN und PASCAL aussieht, entnimmt man am besten den Beispielen. Die Parameter sind vom Typ integer, um die Beispiele einfach zu halten. Ferner ist noch ein Shellscript angegeben, das eine C-Funktion aufruft, die in diesem Fall ein selbständiges Programm (Funktion `main()`) sein muß.

Der von einer Funktion zurückgegebene Wert (**Rückgabewert**) kann nur ein einfacher Typ oder ein Pointer sein. Zusammengesetzte Typen wie Arrays, Strings oder Strukturen können nur durch Pointer zurückgegeben werden. Es ist zulässig, keinen Wert zurückzugeben. Dann ist die Funktion vom Typ void und macht sich allein durch ihre Nebeneffekte bemerkbar.

Für die Systemaufrufe von UNIX und die Standardfunktionen von C ist im Referenz-Handbuch in den Sektionen (2) und (3) angegeben, von welchem Typ die Argumente und der Funktionswert sind. Da diese Funktionen allesamt C-Funktionen sind, lassen sie sich ohne Probleme in C-Programme einbinden. Bei anderen Sprachen ist es denkbar, daß kein einem C-Typ entsprechender Variablentyp verfügbar ist. Auch bei Strings gibt es wegen der unterschiedlichen Speicherung in den einzelnen Sprachen Reibereien. Falls die Übergabemechanismen unverträglich sind, muß man die C-Funktion in eine Funktion oder Prozedur der anderen Sprache so verpacken, daß das aufrufende Programm eine einheitliche Programmiersprache sieht. Das Vorgehen dabei kann maschinenbezogen sein, was man eigentlich vermeiden will.

In den folgenden Programmbeispielen wird die Summe aus zwei Summanden berechnet, zuerst im Hauptprogramm direkt und dann durch zwei Funktionen, die ihre Argumente – die Summanden – by value beziehungsweise by reference übernehmen. Die Funktionen verändern ihre Summanden, was im ersten Fall keine Auswirkung im Hauptprogramm hat. Hauptprogramme und Funktionen sind in C, FORTRAN und PASCAL geschrieben, was neun Kombinationen ergibt. Wir betreten damit zugleich das an Fallgruben reiche Gebiet der Mischung von Programmiersprachen (mixed language programming). Zunächst die beiden Funktionen im geliebten C:

```
/* C-Funktion (Summe) call by value */
/* Compileraufruf cc -c csv.c, liefert csv.o */

int csv(int x,int y)
{
int z;
puts("Funktion mit Parameteruebernahme by value:");
printf("C-Fkt. hat uebernommen:    %d   %d\n", x, y);
z = x + y;
printf("C-Fkt. gibt folgende Summe zurueck: %d\n", z);
/* Aenderung der Summanden */
x = 77; y = 99;
return(z);
}
```

Quelle 0.2 . C-Funktion, die Parameter by value übernimmt

```
/* C-Funktion (Summe) call by reference */
/* Compileraufruf cc -c csr.c, liefert csr.o */

int csr(int *px,int *py)
{
```

```
int z;
puts("Funktion mit Parameteruebenahme by reference:");
printf("C-Fkt. hat uebernommen: %d   %d\n", *px, *py);
z = *px + *py;
printf("C-Fkt. gibt folgende Summe zurueck:  %d\n", z);
/* Aenderung der Summanden */
*px = 66; *py = 88;
return(z);
}
```

Quelle 0.3 . C-Funktion, die Parameter by reference übernimmt

Im bewährten FORTRAN 77 haben wir leider keinen Weg gefunden, der Funktion beizubringen, ihre Parameter by value zu übernehmen (in FORTRAN 90 ist es möglich). Es bleibt daher bei nur einer Funktion, die – wie in FORTRAN üblich – ihre Parameter by reference übernimmt:

```
C     Fortran-Funktion (Summe) call by reference
C     Compileraufruf f77 -c fsr.f

      integer function fsr(x, y)
      integer x, y, z

      write (6,'("F-Fkt. mit Uebernahme by reference:")')
      write (6,'("F-Fkt. hat uebernommen: ",2I6)') x, y
      z = x + y
      write (6,'("F-Fkt. gibt zurueck: ",I8)') z
C     Aenderung der Summanden
      x = 66
      y = 88
      fsr = z

      end
```

Quelle 0.4 . FORTRAN-Funktion, die Parameter by reference übernimmt

PASCAL-Funktionen kennen wieder beide Möglichkeiten, aber wir werden auf eine andere Schwierigkeit stoßen. Vorläufig sind wir jedoch hoffnungsvoll:

```
{Pascal-Funktion (Summe) call by value}
{Compileraufruf pc -c psv.p}

module b;
import StdOutput;
export
    function psv(x, y: integer): integer;
implement
    function psv;
    var z: integer;
    begin
        writeln('Funktion mit P'uebernahme by value:');
```

```
        writeln('P-Fkt. hat uebernommen: ', x, y);
        z := x + y;
        writeln('P-Fkt. gibt folgenden Wert zurueck: ', z);
        { Aenderung der Summanden }
        x := 77; y := 99;
        psv := z;
    end;
end.
```

Quelle 0.5 . PASCAL-Funktion, die Parameter by value übernimmt

```
{Pascal-Funktion (Summe) call by reference}
{Compileraufruf pc -c psr.p}

module a;
import StdOutput;
export
    function psr(var x, y: integer): integer;
implement
    function psr;
    var z: integer;
    begin
        writeln('Funktion mit P'uebernahme by reference:');
        writeln('P-Fkt. hat uebernommen: ', x, y );
        z := x + y;
        writeln('P-Fkt. gibt folgenden Wert zurueck: ', z);
        { Aenderung der Summanden }
        x := 66; y := 88;
        psr := z;
    end;
end.
```

Quelle 0.6 . PASCAL-Funktion, die Parameter by reference übernimmt

Die Funktionen werden für sich mit der Option `-c` ihres jeweiligen Compilers kompiliert, wodurch Objektfiles mit der Kennung `.o` entstehen, die beim Kompilieren der Hauptprogramme aufgeführt werden. Nun zu den Hauptprogrammen, zuerst wieder in C:

```
/* C-Programm csummec, das C-Funktionen aufruft */
/* Compileraufruf cc -o csummec csummec.c csr.o csv.o */

#include <stdio.h>

extern int csv(int x,int y),
           csr(int *px,int *py);

int main()
{
int a, b;
```

```
puts("Bitte die beiden Summanden eingeben!");
scanf("%d %d", &a, &b);
printf("Die Summanden sind:  %d   %d\n", a, b);
printf("Die Summe (direkt) ist:   %d\n", (a + b));
printf("Die Summe ist: %d\n", csv(a, b));
printf("Die Summanden sind:  %d   %d\n", a, b);
printf("Die Summe ist:  %d\n", csr(&a, &b));
printf("Die Summanden sind:  %d   %d\n", a, b);
return 0;
}
```

Quelle 0.7 . C-Programm, das Parameter by value und by reference an C-Funktionen übergibt

Nun das C-Hauptprogramm, das eine FORTRAN-Funktion aufruft, ein in der Numerik häufiger Fall:

```
/* C-Programm csummef, das eine FORTRAN-Funktion aufruft */
/* Compileraufruf cc -o csummef csummef.c fsr.o -lcl */

#include <stdio.h>

extern int fsr(int *x,int *y);

int main()
{
int a, b;
scanf("%d %d", &a, &b);
printf("Die Summanden sind:  %d   %d\n", a, b);
printf("Die Summe (direkt) ist:   %d\n", (a + b));
printf("Die Summe ist:  %d\n", fsr(&a, &b));
printf("Die Summanden sind:  %d   %d\n", a, b);
return 0;
}
```

Quelle 0.8 . C-Programm, das Parameter by reference an eine FORTRAN-Funktion übergibt

Die Linker-Option `-lcl` ist erforderlich, wenn FORTRAN- oder PASCAL-Module in C-Programme eingebunden werden. Sie bewirkt die Hinzunahme der FORTRAN- und PASCAL-Laufzeitbibliothek `/usr/lib/libcl.a`, ohne die Bezüge (Referenzen) auf FORTRAN- oder PASCAL-Routinen unter Umständen offen bleiben. Anders gesagt, in den FORTRAN- oder PASCAL-Funktionen kommen Namen vor – zum Beispiel `write` – deren Definition in besagter Laufzeitbibliothek zu finden ist. C und PASCAL sind sich im großen ganzen ähnlich, es gibt aber Unterschiede hinsichtlich des Geltungsbereiches von Variablen, die hier nicht deutlich werden:

```
/* C-Programm csummep, das PASCAL-Funktionen aufruft. */
/* Compiler: cc -o csummep csummep.c psv.o psr.o -lcl */

#include <stdio.h>
```

```
extern int psv(int x,int y),psr(int *x,int *y)

int main()
{
int a, b;
puts("Bitte die beiden Summanden eingeben!");
scanf("%d %d", &a, &b);
printf("Die Summanden sind: %d   %d\n", a, b);
printf("Die Summe (direkt) ist: %d\n", (a + b));
printf("Die Summe ist: %d\n", psv(a, b));
printf("Die Summanden sind: %d   %d\n", a, b);
printf("Die Summe ist: %d\n", psr(&a, &b));
printf("Die Summanden sind: %d   %d\n", a, b);
return 0;
}
```

Quelle 0.9 . C-Programm, das Parameter by value und by reference an PASCAL-Funktionen übergibt

Hiernach sollte klar sein, warum die C-Standardfunktion `printf(3)` mit Variablen als Argument arbeitet, während die ähnliche C-Standardfunktion `scanf(3)` Pointer als Argument verlangt. `printf(3)` gibt Werte aus, ohne sie zu ändern. Es ist für das Ergebnis belanglos, ob die Funktion Adressen (Pointer) oder Kopien der Variablen verwendet (die Syntax legt das allerdings fest). Hingegen soll `scanf(3)` Werte mit Wirkung für die aufrufende Funktion einlesen. Falls es sich nur um einen Wert handelte, könnte das noch über den Returnwert bewerkstelligt werden, aber `scanf(3)` soll mehrere Werte – dazu noch verschiedenen Typs – verarbeiten. Das geht nur über von `scanf(3)` und der aufrufenden Funktion gemeinsam verwendete Pointer.

Nun die drei FORTRAN-Hauptprogramme mit Aufruf der Funktionen in C, FORTRAN und PASCAL:

```
C     FORTRAN-Programm, das C-Funktionen aufruft
C     Compileraufruf f77 -o fsummec fummec.f csv.o csr.o

      program fsummec

$ALIAS csv (%val, %val)

      integer a, b , s, csr, csv

      write (6, 100)
      read (5, *) a, b
      write (6, 102) a, b
      s = a + b
      write (6, 103) s
C     call by value
      s = csv(a, b)
      write (6, 104) s
```

```
      write (6, 102) a, b
C     call by reference
      s = csr(a, b)
      write (6, 105) s
      write (6, 102) a, b

  100 format (’Bitte die beiden Summanden eingeben!’)
  102 format (’Die Summanden sind: ’, 2I6)
  103 format (’Die Summe (direkt) ist: ’, I8)
  104 format (’Die Summe ist: ’, I8)
  105 format (’Die Summe ist: ’, I8)

      end
```

Quelle 0.10 . FORTRAN-Programm, das Parameter by value und by reference an C-Funktionen übergibt

```
C     FORTRAN-Programm, das F77-Funktion aufruft
C     Compileraufruf f77 -o fsummef fsummef.f fsr.o

      program fsummef

      integer a, b , s, fsr

      write (6, 100)
      read (5, *) a, b
      write (6, 102) a, b
      s = a + b
      write (6, 103) s
C     call by value nicht moeglich
C     call by reference (default)
      s = fsr(a, b)
      write (6, 105) s
      write (6, 102) a, b

  100 format (’Bitte die beiden Summanden eingeben!’)
  102 format (’Die Summanden sind: ’, 2I6)
  103 format (’Die Summe (direkt) ist: ’, I8)
  105 format (’Die Summe ist: ’, I8)

      end
```

Quelle 0.11 . FORTRAN-Programm, das Parameter by reference an eine FORTRAN-Funktion übergibt

```
C     FORTRAN-Programm, das PASCAL-Funktionen aufruft
C     Compileraufruf f77 -o fsummep fsummep.f psv.o psr.o

      program fsummep
```

```
$ALIAS psv (%val, %val)

      integer a, b, s, psv, psr
      external psv, psr

      write (6, 100)
      read (5, *) a, b
      write (6, 102) a, b
      s = a + b
      write (6, 103) s
C     call by value
      s = psv(a, b)
      write (6, 104) s
      write (6, 102) a, b
C     call by reference
      s = psr(a, b)
      write (6, 105) s
      write (6, 102) a, b

  100 format ('Bitte die beiden Summanden eingeben!')
  102 format ('Die Summanden sind: ', 2I6)
  103 format ('Die Summe (direkt) ist: ', I8)
  104 format ('Die Summe ist: ', I8)
  105 format ('Die Summe ist: ', I8)

      end
```

Quelle 0.12 . FORTRAN-Programm, das Parameter by value und by reference an PASCAL-Funktionen übergibt

Die FORTRAN-Compiler-Anweisung `$ALIAS` veranlaßt den Compiler, der jeweiligen Funktion die Parameter entgegen seiner Gewohnheit by value zu übergeben. Zum guten Schluß die PASCAL-Hauptprogramme:

```
{PASCAL-Programm, das C-Funktionen aufruft}
{Compiler: pc -o psummec psummec.p csv.o csr.o}

program psummec (input, output);

var a, b, s: integer;

function csv(x, y: integer): integer;      {call by value}
         external C;

function csr(var x, y: integer): integer;  {call by ref.}
         external C;

begin
writeln('Bitte die beiden Summanden eingeben!');
readln(a); readln(b);
write('Die Summanden sind: '); write(a); writeln(b);
s := a + b;
```

```
write('Die Summe (direkt) ist: '); writeln(s);
s := csv(a, b);
write('Die Summe ist: '); writeln(s);
write('Die Summanden sind: '); write(a); writeln(b);
s := csr(a, b);
write('Die Summe ist: '); writeln(s);
write('Die Summanden sind: '); write(a); writeln(b);
end.
```

Quelle 0.13 . PASCAL-Programm, das Parameter by value und by reference an C-Funktionen übergibt

```
{PASCAL-Programm, das FORTRAN-Funktion aufruft}
{Compiler: pc -o psummef psummef.p fsr.o}

program psummef (input, output);

var a, b, s: integer;

function fsr(var x, y: integer): integer;  {call by ref.}
         external ftn77;

begin
writeln('Bitte die beiden Summanden eingeben!');
readln(a); readln(b);
write('Die Summanden sind: '); write(a); writeln(b);
s := a + b;
write('Die Summe (direkt) ist: '); writeln(s);
s := fsr(a, b);
write('Die Summe ist: '); writeln(s);
write('Die Summanden sind: '); write(a); writeln(b);
end.
```

Quelle 0.14 . PASCAL-Programm, das Parameter by reference an eine FORTRAN-Funktion übergibt

```
{PASCAL-Programm, das PASCAL-Funktionen aufruft}
{Compileraufruf pc -o psummep psummep.p psv.o psr.o}

program psummep (input, output);

var a, b, s: integer;

function psv(x, y: integer): integer;      {call by value}
         external;

function psr(var x, y: integer): integer;  {call by ref.}
         external;

begin
```

```
writeln('Bitte die beiden Summanden eingeben!');
readln(a); readln(b);
write('Die Summanden sind: '); write(a); writeln(b);
s := a + b;
write('Die Summe (direkt) ist: '); writeln(s);
s := psv(a, b);
write('Die Summe ist: '); writeln(s);
write('Die Summanden sind: '); write(a); writeln(b);
s := psr(a, b);
write('Die Summe ist: '); writeln(s);
write('Die Summanden sind: '); write(a); writeln(b);
end.
```

Quelle 0.15 . PASCAL-Programm, das Parameter by value und by reference an PASCAL-Funktionen übergibt

Sollten Sie die Beispiele nachvollzogen haben, müßte Ihr Linker in zwei Fällen mit einer Fehlermeldung `unsatisfied symbol: output (data)` die Arbeit verweigert haben. Die PASCAL-Funktionen `psv()` und `psr()` geben etwas auf das Terminal aus. Bei getrennt kompilierten Modulen erfordert dies die Zeile:

```
import StdOutput;
```

Das importierte, vorgefertigte PASCAL-Modul `StdOutput` macht von einem Textfile `output` Gebrauch, das letzten Endes der Bildschirm ist. Im PASCAL-Programm öffnet die Zeile

```
program psummep (input, output);
```

dieses Textfile. In C-Programmen wird das File mit dem Filepointer `stdout` ebenso wie in FORTRAN-Programmen die Unit 6 automatisch geöffnet. Hinter dem Filepointer bzw. der Unit steckt der Bildschirm. Leider sehen wir – in Übereinstimmung mit unseren Handbüchern – keinen Weg, das PASCAL-File `output` mit `stdout` von C oder der Unit 6 von FORTRAN zu verbinden. Wollen wir PASCAL-Funktionen in ein C- oder FORTRAN-Programm einbinden, müssen die Funktionen auf Terminalausgabe verzichten (eine Ausgabe in ein File wäre möglich):

```
{Pascal-Funktion (Summe) call by value, ohne Output}
{Compileraufruf pc -c xpsv.p}

module b;
export
    function psv(x, y: integer): integer;
implement
    function psv;
    var z: integer;
    begin
        z := x + y;
        { Aenderung der Summanden }
        x := 77; y := 99;
        psv := z;
```

```
    end;
end.
```

Quelle 0.16 . PASCAL-Funktion, die Parameter by value übernimmt, ohne Ausgabe

```
{Pascal-Funktion (Summe) call by reference}
{ohne Output}
{Compileraufruf pc -c xpsr.p}

module a;
export
    function psr(var x, y: integer): integer;
implement
    function psr;
    var z: integer;
    begin
        z := x + y;
        { Aenderung der Summanden }
        x := 66; y := 88;
        psr := z;
    end;
end.
```

Quelle 0.17 . PASCAL-Funktion, die Parameter by reference übernimmt, ohne Ausgabe

Damit geht es. Der Compilerbauer weiß, wie die einzelnen Programmiersprachen ihre Ausgabe bewerkstelligen und kann Übergänge in Form von Compiler-Anweisungen oder Zwischenfunktionen einrichten. So macht es Microsoft bei seinem großen C-Compiler. Aber wenn nichts vorgesehen ist, muß der gewöhnliche Programmierer solche Unverträglichkeiten hinnehmen.

Auch Shellscripts können Funktionen aufrufen. Diese müssen selbständige Programme wie externe Kommandos sein, der Mechanismus sieht etwas anders aus. Hier das Shellscript:

```
# Shellscript, das eine C-Funktion aufruft. 28.01.1988
# Filename shsumme

print Bitte die beiden Summanden eingeben!
read a; read b
print Die Summanden sind $a $b
print Die Shell-Summe ist ‘expr $a + $b‘
print Die Funktions-Summe ist ‘cssh $a $b‘
print Die Summanden sind $a $b
exit
```

Quelle 0.18 . Shellscript mit Parameterübergabe

Die zugehörige C-Funktion ist ein Hauptprogramm:

```
/* C-Programm zum Aufruf durch Shellskript, 29.01.1988 */
/* Compileraufruf: cc -o cssh cssh.c */

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

{
int x, y;
sscanf(argv[1], "%d", &x);
sscanf(argv[2], "%d", &y);
printf("%d", (x + y));
return 0;
}
```

Quelle 0.19 . C-Programm, das Parameter von einem Shellscript übernimmt

Ferner können Shellscripts **Shellfunktionen** aufrufen, siehe das Shellscript **??** *Türme von Hanoi* auf Seite **??**.

Entschuldigen Sie bitte, daß dieser Abschnitt etwas breit geraten ist. Die Parameterübergabe muß sitzen, wenn man mehr als Trivialprogramme schreibt, und man ist nicht immer in der glücklichen Lage, rein in C programmieren zu können. Verwendet man vorgegebene Bibliotheken, so sind diese manchmal in einer anderen Programmiersprache verfaßt. Dann hat man sich mit einer fremden Syntax und den kleinen, aber bedeutsamen Unverträglichkeiten herumzuschlagen.

0.1.4 Kommandozeilenargumente, main()

Auch das Hauptprogramm `main()` ist eine Funktion, die Parameter oder Argumente übernehmen kann, und zwar aus der **Kommandozeile** beim Aufruf des Programms. Sie kennen das von vielen UNIX-Kommandos, die nichts anderes als C-Programme sind.

Der Mechanismus ist stets derselbe. Die Argumente, getrennt durch Spaces oder Ähnliches, werden in ein Array of Strings mit dem Namen `argv` (**Argumentvektor**) gestellt. Gleichzeitig zählt ein **Argumentzähler** `argc` die Anzahl der übergebenen Argumente, wobei der Funktionsname selbst das erste Argument (Index 0) ist. Bei einem Programmaufruf ohne Argumente steht also der Programmname in `argv[0]`, der Argumentzähler `argc` hat den Wert 1. Das erste nichtbelegte Element des Argumentvektors enthält einen leeren String. Die Umwandlung der Argumente vom Typ String in den gewünschten Typ besorgt die Funktion `sscanf(3)`.

Der Anfang eines Hauptprogrammes mit Kommandozeilenargumenten sieht folgendermaßen aus:

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

{
char a; int x;

if (argc < 3) {
        puts("Zuwenige Parameter");
```

```
        exit(-1);
}
sscanf(argv[1], "%c", &a);
sscanf(argv[2], "%d", &x);
....
```

Quelle 0.20 . C-Programm, das Argumente aus der Kommandozeile übernimmt

Das erste Kommandozeilenargument (nach dem Kommando selbst) wird als Zeichen verarbeitet, das zweite als ganze Zahl. Etwaige weitere Argumente fallen unter den Tisch.

Die Funktion `main()` ist immer vom Typ `extern int`. Da dies der Defaulttyp für Funktionen ist, könnte die Typdeklaration weggelassen werden. Sie kann Argumente übernehmen, braucht es aber nicht. Infolgedessen sind folgende Deklarationen gültig:

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

und alle anderen falsch. Die ersten sechs sind in ihrer Bedeutung gleich, die weiteren gelten bei Argumenten in der Kommandozeile. Die Norm ISO/IEC 9899:1999 sieht für C-Programme, die unter einem Betriebssystem laufen (hosted environment), nur die beiden folgenden Formen vor:

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

und das reicht auch. Unter POSIX-konformen Betriebssystemen kann ein drittes Argument hinzukommen, das die Umgebung (environment pointer) enthält:

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

Den Rückgabewert von `main()` sollte man nicht dem Zufall überlassen, sondern mit einer `return`-Anweisung ausdrücklich festlegen (0 bei Erfolg). Er wird von der Shell übernommen.

0.1.5 Funktionen mit wechselnder Argumentanzahl

Mit `main()` haben wir eine Funktion kennengelernt, die eine wechselnde Anzahl von Argumenten übernimmt. Auch für andere Funktionen als `main()` gibt es einen

Mechanismus zu diesem Zweck, schauen Sie bitte unter `varargs(5)` nach. Der Mechanismus ist nicht übermäßig intelligent, sondern an einige Voraussetzungen gebunden:

- Es muß mindestens ein Argument vorhanden sein,
- der Typ des ersten Arguments muß bekannt sein,
- es muß ein Kriterium für das Ende der Argumentliste bekannt sein.

Die erforderlichen Makros stehen in den include-Files `<varargs.h>` für UNIX System V oder `<stdarg.h>` für ANSI-C. Wir erklären die Vorgehensweise an einem Beispiel, das der Funktion `printf(3)` nachempfunden ist (es ist damit nicht gesagt, daß `printf(3)` tatsächlich so aussieht):

```
/* Funktion printi(), Ersatz fuer printf(), nur fuer
   dezimale Ganzzahlen, Zeichen und Strings. Siehe
   Referenz-Handbuch unter varargs(5), 22.02.91 */
/* Returnwert 0 = ok, -1 = Fehler, sonst wie printf() */
/* Compileraufruf cc -c printi.c */

#include <stdio.h>
#include <varargs.h>

int fputc();
void int_print();

/* Funktion printi(), variable Anzahl von Argumenten */

int printi(va_alist)
va_dcl
{
    va_list pvar;
    unsigned long arg;
    int field, sig;
    char *format, *string;
    long ivar;

/* Uebernahme und Auswertung des Formatstrings */

    va_start(pvar);
    format = va_arg(pvar, char *);

    while (1) {

/* Ausgabe von Literalen */

    while ((*format != '%') && (*format != '\0'))
        fputc(*format++, stdout);

/* Ende des Formatstrings */

    if (*format == '\0') {
        va_end(pvar);
```

```
        return 0;
    }

/* Prozentzeichen, Platzhalter */

    format++;
    field = 0;

/* Auswertung Laengenangabe */

    while (*format >= '0' && *format <='9') {
        field = field * 10 + *format - '0';
        format++;
    }

/* Auswertung Typangabe und Ausgabe des Arguments */

    switch(*format) {
      case 'd':
        sig = ((ivar  = (long)va_arg(pvar, int)) < 0 ? 1 : 0);
        arg = (unsigned long)(ivar < 0 ? -ivar : ivar);
        int_print(arg, sig, field);
        break;
      case 'u':
        arg = (unsigned long)va_arg(pvar, unsigned);
        int_print(arg, 0, field);
        break;
      case 'l':
        switch(*(format + 1)) {
            case 'd':
                sig = ((ivar = va_arg(pvar, long)) < 0 ? 1 : 0);
                arg = (unsigned long)(ivar < 0 ? -ivar : ivar);
                int_print(arg, sig, field);
                break;
            case 'u':
                arg = va_arg(pvar, unsigned long);
                int_print(arg, 0, field);
                break;
            default:
                va_end(pvar);
                return -1;      /* unbekannter Typ */
        }
        format++;
        break;
      case '%':
        fputc(*format, stdout);
        break;
      case 'c':
        fputc(va_arg(pvar, char), stdout);
        break;
      case 's':
        string = va_arg(pvar, char *);
```

```
        while ((fputc(*(string++), stdout)) != '\0') ;
        break;
      default:
        va_end(pvar);
        return -1;              /* unbekannter Typ */
    }
    format++;
}
}

/* Funktion zur Ausgabe der dezimalen Ganzzahl */

void int_print(unsigned long number,int signum,int field)

{
    int i;
    char table[21];
    long radix = 10;

    for (i = 0; i < 21; i++)
        *(table + i) = ' ';

/* Umwandlung Zahl nach ASCII-Zeichen */

    for (i = 0; i < 20; i++) {
        *(table + i) = *("0123456789" + (number % radix));
        number /= radix;
        if (number == 0) break;
    }

/* Vorzeichen */

    if (signum)
        *(table + ++i) = '-';

/* Ausgabe */

    if ((field != 0) && (field < 20))
        i = field -1;

    while (i >= 0)
    fputc(*(table + i-), stdout);
}

/* Ende */
```

Quelle 0.21 . C-Funktion mit wechselnder Anzahl von Argumenten

Nach dem include-File `varargs.h` folgt in gewohnter Weise die Funktion, hier `printi()`. Ihre Argumentenliste heißt `va_alist` und ist vom Typ `va_dcl`, ohne Semikolon! Innerhalb der Funktion brauchen wir einen Pointer `pvar` auf die Argu-

mente, dieser ist vom Typ `va_list`, nicht zu verwechseln mit der Argumentenliste `va_alist`. Die weiteren Variablen sind unverbindlich.

Zu Beginn der Arbeit muß das Makro `va_start(pvar)` aufgerufen werden. Es initialisiert den Pointer `pvar` mit dem Anfang der Argumentenliste. Am Ende der Arbeit muß entsprechend mit dem Makro `va_end(pvar)` aufgeräumt werden.

Das Makro `va_arg(pvar, type)` gibt das Argument zurück, auf das der Pointer `pvar` zeigt, und zwar in der Form des angegebenen Typs, den man also kennen muß. Gleichzeitig wird der Pointer `pvar` eins weiter geschoben. Die Zeile

```
format = va_arg(pvar, char *);
```

weist dem Pointer auf char `format` die Adresse des Formatstrings in der Argumentenliste von `printi()` zu. Damit ist der Formatstring wie jeder andere String zugänglich. Zugleich wird der Pointer `pvar` auf das nächste Argument gestellt, üblicherweise eine Konstante oder Variable. Aus der Auswertung des Formatstrings ergeben sich Anzahl und Typen der weiteren Argumente.

Damit wird auch klar, was geschieht, wenn die Platzhalter (%d, %6u usw.) im **Formatstring** nicht mit der Argumentenliste übereinstimmen. Gibt es mehr Argumente als Platzhalter, werden sie nicht beachtet. Gibt es mehr Platzhalter als Argumente, wird irgendein undefinierter Speicherinhalt gelesen, unter Umständen auch der dem Programm zugewiesene Speicherbereich verlassen. Stimmen Platzhalter und Argumente im Typ nicht überein, wird der Pointer `pvar` falsch inkrementiert, und die Typumwandlung geht vermutlich auch daneben.

Es gibt eine Fallgrube bei der Typangabe. Je nach Compiler werden die Typen `char` und `short` intern als `int` und `float` als `double` verarbeitet. In solchen Fällen muß dem Makro `va_arg(pvar, type)` der interne Typ mitgeteilt werden. Nachlesen oder ausprobieren, am besten beides.

0.1.6 Iterativer Aufruf einer Funktion

Unter einer **Iteration** versteht man die Wiederholung bestimmter Programmschritte, wobei das Ergebnis eines Schrittes als Eingabe für die nächste Wiederholung dient. Viele mathematische Näherungsverfahren machen von Iterationen Gebrauch. Programmtechnisch führen Iterationen auf Schleifen. Entsprechend muß eine Bedingung angegeben werden, die die Iteration beendet. Da auch bei einem richtigen Programm eine Iteration manchmal aus mathematischen Gründen nie zu einem Ende kommt, ist es zweckmäßig, einen Test für solche Fälle einzubauen wie in folgendem Beispiel:

```
/* Quadratwurzel, Halbierungsverfahren, 14.08.92 */
/* Compileraufruf cc -o wurzel wurzel.c */

#define  EPS 0.00001
#define  MAX 100

#include <stdio.h>
```

```
void exit();

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

{
int i;
double a, b, c, m;

if (argc < 2) {
    puts("Radikand fehlt.");
    exit(-1);
}

/* Initialisierung */

i = 0;
sscanf(argv[1], "%lf", &c);
sscanf(argv[1], "%lf", &c);
a = 0;
b = c + 1;

/* Iteration */

while (b - a > EPS) {
    m = (a + b) / 2;
    if (m * m - c <= 0)
        a = m;
    else
        b = m;

/* Begrenzung der Anzahl der Iterationen */

    i++;
    if (i > MAX) {
         puts("Zuviele Iterationen! Ungenau!");
         break;
    }
}

/* Ausgabe und Ende */

printf("Die Wurzel aus %lf ist %lf\n", c, m);
printf("Anzahl der Iterationen: %d\n", i);
exit(0);
}
```

Quelle 0.22 . C-Programm zur iterativen Berechnung der Quadratwurzel

Die Funktion, die iterativ aufgerufen wird, ist die Mittelwertbildung von a und b; es lohnt sich nicht, sie auch programmtechnisch als selbständige Funktion zu definieren, aber das kann in anderen Aufgaben anders sein.

0.1.7 Rekursiver Aufruf einer Funktion

Bei einer **Rekursion** ruft eine Funktion sich selbst auf. Das ist etwas schwierig vorzustellen und nicht in allen Programmiersprachen erlaubt. Die Nähe zum Zirkelschluß ist nicht geheuer. Es gibt aber Probleme, die ursprünglich rekursiv sind und sich durch eine Rekursion elegant programmieren lassen. Eine **Zirkeldefinition** ist eine Definition eines Begriffes, die diesen selbst in der Definition enthält, damit es nicht sofort auffällt, gegebenenfalls um einige Ecken herum. Ein **Zirkelschluß** ist eine Folgerung, die Teile der zu beweisenden Aussage bereits zur Voraussetzung hat. Bei einer Rekursion hingegen

- wiederholt sich die Ausgangslage nie,
- wird eine Abbruchbedingung nach endlich vielen Schritten erfüllt, d. h. die Rekursionstiefe ist begrenzt.

In dem Buch von ROBERT SEDGEWICK findet sich Näheres zu diesem Thema, mit Programmbeispielen. Im ersten Band der *Informatik* von FRIEDRICH L. BAUER und GERHARD GOOS wird die Rekursion allgemeiner abgehandelt.

Zwei Beispiele sollen die Rekursion veranschaulichen. Das erste Programm berechnet den größten gemeinsamen Teiler (ggT) zweier ganzer Zahlen nach dem Algorithmus von EUKLID. Das zweite ermittelt rekursiv die Fakultät einer Zahl, was man anders vielleicht einfacher erledigen könnte.

```
/* Groesster gemeinsamer Teiler, Euklid, rekursiv */
/* Compileraufruf cc -o ggtr ggtr.c */

#include <stdio.h>

int ggt();

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

{
int x, y;

sscanf(argv[1], "%d", &x); sscanf(argv[2], "%d", &y);
printf("Der GGT von %d und %d ist %d.\n", x, y, ggt(x, y));
return 0;
}

/* Funktion ggt() */

int ggt(int a,int b)
{
if (a == b) return a;
else if (a > b) return(ggt(a - b, b));
else return(ggt(a, b - a));
}
```

Quelle 0.23 . C-Programm Größter gemeinsamer Teiler (ggT) nach Euklid, rekursiv

Im folgenden Programm ist außer der Rekursivität die Verwendung der Bedingten Bewertung interessant, die den Code verkürzt.

```
/* Rekursive Berechnung von Fakultaeten */

#include <stdio.h>

int main()
{
    int n;
    puts("\nWert eingeben, Ende mit CTRL-D");
    while (scanf("%d", &n) != EOF)
        printf("\n%d Fakultaet ist %d.\n\n", n, fak(n));
    return 0;
}

/* funktion fak() */

int fak(int n)
{
    return(n <= 1 ? 1 : n * fak(n - 1));
}
```

Quelle 0.24 . C-Programm zur rekursiven Berechnung der Fakultät

Weitere rekursiv lösbare Aufgaben sind die Türme von Hanoi und Quicksort. Rekursive Probleme lassen sich auch iterativ lösen. Das kann sogar schneller gehen, aber die Eleganz bleibt auf der Strecke.

Da in C auch das Hauptprogramm `main()` eine Funktion ist, die auf gleicher Stufe mit allen anderen Funktionen steht, kann es sich selbst aufrufen:

```
/* Experimentelles Programm mit Selbstaufruf von main() */

#include <stdio.h>

int main()
{
puts("Selbstaufruf von main()");
main();
return(13);
}
```

Quelle 0.25 . C-Programm, in dem main() sich selbst aufruft

Das Programm wird von `lint(1)` nicht beanstandet, einwandfrei kompiliert und läuft, bis der Speicher platzt, da die Rekursionstiefe nicht begrenzt ist (Abbruch mit break). Allerdings ist ein Selbstaufruf von `main()` ungebräuchlich.

0.1.8 Assemblerroutinen

Auf die Assemblerprogrammierung wurde in Abschnitt **??** *Programmiersprachen* auf Seite **??** bereits eingegangen. Da das Schreiben von Programmen in Assembler mühsam ist und die Programme nicht portierbar sind, läßt man nach Möglichkeit die Finger davon. Es kann jedoch zweckmäßig sein, einfache, kurze Funktionen auf Assembler umzustellen. Einmal kann man so unmittelbar auf die Hardware zugreifen, beispielsweise in Anwendungen zum Messen und Regeln, zum anderen zur Beschleunigung oft wiederholter Funktionen.

```
/* fakul.c Berechnung von Fakultaeten */

/* Die Grenze fuer END liegt in der Segmentgroesse */
/* bis 260 werden alle Werte in einem Array gespeichert,
   darueber wird Wert fuer Wert berechnet und ausgegeben */
/* Ziffern in Neunergruppen, nutzt long aus */

#define END  260
#define MAX 1023
#define DEF   16
#define GRP   58
#define GMX  245

/* GRP muss in aadd.asm eingetragen werden */
/* GMX muss in laadd.asm eingetragen werden */

#include <stdio.h>

unsigned long f[END + 1][GRP];          /* global */

void add(unsigned long *, unsigned long *);
void exit(int);
long time(long *);

/* Assemblerfunktionen zur Beschleunigung &/

extern void aadd(unsigned long *, unsigned long *);
extern void lshift(unsigned long *);

/* Hauptprogramm */

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

{
int e, i, j, k, r, s, flag, ende, max = DEF;
unsigned long x[GRP];
unsigned long *z;
long z1, z2, z3;

/* Auswertung der Kommandozeile */
```

```
if (argc > 1) {
        sscanf(*(argv + 1), "%d", &max);
        max = (max < 0) ? -max : max;
        if (max > MAX) {
           printf("\nZahl zu gross! Maximal %d\n", MAX);
           exit(1);
        }
}

ende = (max > END) ? END : max;

time(&z1);                      /* Zeit holen */

/* Rechnung */

**f = (unsigned long)1;

for (i = 1; i <= ende; i++) {
        for (j = 0; j < GRP; j++)    /* x nullsetzen */
                *(x + j) = 0;
        k = i/4;
        for (j = 1; j <= k; j++) {   /* addieren */
                aadd(x, *(f + i - 1));
        }
        lshift(x);
        lshift(x);
        for (k = 0; k < (i % 4); k++)
                aadd(x, *(f + i - 1));
        for (j = 0; j < GRP; j++) {  /* zurueckschreiben */
                *(*(f + i) + j) = *(x + j);

/* *(*(f + i) + j) ist dasselbe wie f[i][j] */

        }
}

time(&z2);                      /* Zeit holen */

/* Ausgabe, fuehrende Nullen unterdrueckt */

printf("\n\tFakultaeten von 0 bis %4d\n", max);

for (i = 0; i <= ende; i++) {
        flag = 0;
        printf("\n\t%4d ! =     ", i);
        for (j = GRP - 1; j >= 0; j-) {
            if (!(*(*(f + i) + j)) && !flag);
            else
                if (!flag) {
                    printf("%9lu ", *(*(f + i) + j));
                    flag = 1;
                }
```

```
                else
                    printf("%09lu ", *(*(f + i) + j));
        }
}

/* falls wir weitermachen wollen, muessen wir das Array
   f[261][58] umfunktionieren in f[2][*].
   In f[0] steht vorige Fakultaet, in f[1] wird addiert. */

if (max > END) {

/* f[0] einrichten */

        e = GMX;                /* kleiner 7296 */

        for (j = 0; j < e; j++)
                *(*f + j) = 0;  /* f[0] nullsetzen */

        aadd(f[0], f[END]);     /* vorige Fak. addieren */

/* Rechnung wie gehabt */

        r = 0; s = e;

        for (i = END + 1; i <= max; i++) {
                for (j = 0; j < e; j++)    /* f[1] nullen */
                        *(*(f + s) + j) = 0;

                k = i/4;
                for (j = 1; j <= k; j++) {  /* addieren */
                        laadd(*(f + s), *(f + r));
                }
                lshift(*(f + s));
                lshift(*(f + s));
                for (k = 0; k < (i % 4); k++)
                        laadd(*(f + s), *(f + r));

                flag = 0;               /* f[1] anzeigen */
                printf("\n\n\t%4d ! =     ", i);
                for (j = e - 1; j >= 0; j-) {
                   if (!(*(*(f + s) + j)) && !flag);
                   else
                      if (!flag) {
                          printf("%9lu ", *(*(f + s) + j));
                          flag = 1;
                      }
                      else
                          printf("%09lu ", *(*(f + s) + j));
                }
        r = (r > 0) ? 0 : e; /* f[1] wird das naechste f[0] */
        s = (s > 0) ? 0 : e;
        }
```

```
}

/* Ende Weitermachen */

/* Anzahl der Stellen von max! */

if (max > END) {
        ende = r; j = GMX - 1;
}
else {
        j = GRP - 1;
}

        flag = 0;
        for (; j >= 0; j-) {
                if (!(*(*(f + ende) + j)) && !flag)
                        ;
                else
                    if (!flag) {
                        unsigned long z = 10;
                        flag = 1;
                        for (i = 1; i < 9; i++) {
                            if (*(*(f + ende) + j) / z) {
                                        flag++;
                                        z *= 10;
                            }
                            else
                                break;
                        }
                    }
                    else
                        flag += 9;
        }

time(&z3);                      /* Zeit holen */

printf("\n\n\tZahl %4d ! hat %4d Stellen.\n", max, flag);

if (max > END)
  printf("\tRechnung+Ausgabe brauchten %4ld s.\n", z3 - z1);
else {
  printf("\tDie Rechnung brauchte   %4ld s.\n", z2 - z1);
  printf("\tDie Ausgabe brauchte    %4ld s.\n", z3 - z2);
}

return 0;
}
```

Quelle 0.26 . C-Programm zur Berechnung von Fakultäten

Das vorstehende Beispiel mit Microsoft Quick C und Quick Assembler für den IBM-PC bietet einen einfachen Einstieg in die Assemblerprogrammierung, da das

große Programm nach wie vor in einer höheren Sprache abgefaßt ist. Das Beispiel ist in einer zweiten Hinsicht interessant. Auf einer 32-Bit-Maschine liegt die größte vorzeichenlose Ganzzahl etwas über 4 Milliarden. Damit kommen wir nicht weit, denn es ist bereits:

$$13! = 6227020800 \tag{0.1}$$

Wir stellen unsere Ergebnisse dar durch ein Array von langen Ganzzahlen, und zwar packen wir immer neun Dezimalstellen in ein Array-Element:

```
unsigned long f[END + 1][GRP]
```

Bei asymmetrischen Verschlüsselungsverfahren braucht man große Zahlen. Die Arithmetik zu diesem Datentyp müssen wir selbst schreiben. Dazu ersetzen wir die eigentlich bei der Berechnung von Fakultäten erforderliche Multiplikation durch die Addition. Diese beschleunigen wir durch Einsatz einer Assemblerfunktion `aadd()`:

```
                COMMENT +
                C-Funktion aadd() in MS-Assembler, die ein
                Array of unsigned long in ein zweites Array
                addiert, Parameter Pointer auf die Arrays.
                +

                .MODEL  small,c
                .CODE
grp             EQU     4 * 58          ; siehe C-Programm
milliarde       DD      1000000000

; fuer laadd() obige Zeile austauschen
; grp           EQU     4 * 245         ; siehe C-Programm


aadd            PROC    USES SI, y:PTR DWORD, g:PTR DWORD

                sub     cx,cx
                sub     si,si
                clc
; for-Schleife nachbilden
for1:
; aktuelles Element in den Akku holen, long = 4 Bytes!
                mov     bx,y
                mov     ax,WORD PTR [bx+si][0]
                mov     dx,WORD PTR [bx+si][2]
; Uebertrag zu Akku addieren
                add     ax,cx
                adc     dx,0
; vorige Fakultaet zu Akku addieren, Uebertrag beachten
                mov     bx,g
                add     ax,WORD PTR [bx+si][0]
                adc     dx,WORD PTR [bx+si][2]
; Summe durch 10 hoch 9 dividieren, Quotient ergibt
; Uebertrag ins naechste Element des Arrays, Rest
: ergibt aktuelles Element.
```

```
; zweite for-Schleife:
                sub     cx,cx
for2:
                cmp     dx,WORD PTR milliarde[2]
                jl      SHORT fertig
                sub     ax,WORD PTR milliarde[0]
                sbb     dx,WORD PTR milliarde[2]
                inc     cx
                jmp     SHORT for2
fertig:
; Rest zurueckschreiben in aktuelles Array
                mov     bx,y
                mov     WORD PTR [bx+si][0],ax
                mov     WORD PTR [bx+si][2],dx
; Schleifenzaehler um 4 (long!) erhoehen
                add     si,4
; Ruecksprungbedingung
                cmp     si,grp
                je      SHORT done
                jmp     SHORT for1
; Ende der Funktion
done:
                ret
aadd            ENDP
                END
```

Quelle 0.27 . Assemblerfunktion 1 zur Addition von Feldern

Die Fakultäten werden berechnet, gespeichert und zum Schluß zusammen ausgegeben. So können wir die Rechenzeit von der Ausgabezeit trennen. Es zeigt sich, daß die Rechenzeit bei Ganzzahl-Arithmetik gegenüber der Bildschirmausgabe keine Rolle spielt.

Auf diesem Weg kommen wir bis in die Gegend von $260!$, dann ist ein Speichersegement (64 KByte) unter PC-DOS voll. Wir können nicht mehr alle Ergebnisse speichern, sondern nur die vorangegangene und die laufende Fakultät. Sowie ein Ergebnis vorliegt, wird es ausgegeben. Die Assemblerfunktion `laadd()` zur Addition unterscheidet sich in einer Zeile am Anfang. Die im Programm vorgesehene Grenze $MAX = 1023$ ist noch nicht die durch das Speichersegment bestimmte Grenze, sondern willkürlich. Irgendwann erheben sich Zweifel am Sinn großer Zahlen. Selbst als Tapetenmuster wirken sie etwas eintönig.

0.1.9 Memo Funktionen

- C-Programme sind aus gleichberechtigten Funktionen aufgebaut. Zu diesen gehört auch `main()`.
- Eine Funktion übernimmt bei ihrem Aufruf einen festgelegten Satz von Parametern oder Argumenten. Der Satz beim Aufruf muß mit dem Satz bei der Definition nach Anzahl, Typ und Reihenfolge übereinstimmen wie Stecker und Kupplung einer elektrischen Steckverbindung.

- Bei der Parameterübergabe by value arbeitet die Funktion mit Kopien der übergebenen Parameter, kann also die Originalwerte nicht verändern.
- Bei der Parameterübergabe by reference erfährt die Funktion die Adressen (Pointer) der Originalwerte und kann diese verändern. Das ist gefährlicher, aber manchmal gewollt. Beispiel: `scanf()`.
- Auch die Funktion `main()` kann Argumente übernehmen, und zwar aus der Kommandozeile. Die Argumente stehen in einem Array of Strings (Argumentvektor).
- Es gibt auch Funktionen wie `printf()`, die eine von Aufruf zu Aufruf wechselnde Anzahl von Argumenten übernehmen. Der Mechanismus ist an einige Voraussetzungen gebunden.
- Eine Funktion gibt keinen oder genau einen Wert als Ergebnis an die aufrufende Funktion zurück. Dieser Wert kann ein ein Pointer sein.
- In C darf eine Funktion sich selbst aufrufen (rekursiver Aufruf).
- Assemblerfunktionen innerhalb eines C-Programms können den Ablauf beschleunigen. Einfacher wird das Programm dadurch nicht.

0.1.10 Übung Funktionen

Jetzt verfügen Sie über die Kenntnisse, die zum Schreiben einfacher C-Programme notwendig sind. Schreiben das Programm zur Weganalyse, aufbauend auf der Aufgabenanalyse und der Datenstruktur, die Sie bereits erarbeitet haben. Falls Sie sich an den Vokabeltrainer wagen wollen, reduzieren Sie die Aufgabe zunächst auf ein Minimum, sonst werden Sie nicht fertig damit.