Programmierung in C und Fortran

Meik Hellmund[1]

Sommersemester 2008

[1] hellmund@math.uni-leipzig.de

Ein Beispielprogramm in C und Fortran:

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

bool test(int);

int main(){

    int n;
    bool res;

    while (true) {
        printf("Bitte ganze Zahl eingeben, \"0\"  für Abbruch:");
        scanf("%i", &n);

        if(n==0) return 0;

        res=test(n);

        if(res==true) {
            printf("Treffer!\n");
        } else {
            printf("Das war leider nicht perfekt. -:(\n");
        }
    }
}


bool test(int x){

    int n=0;





    for(int i=1; i<=x/2; i++) {
        if(x%i == 0) n+=i;
    }

    if (n==x) return true;
    else return false;
}
/* -------------------Dateiende ----------------------------*/
```

```fortran
program bsp1
    implicit none
    integer :: n
    logical :: res,test

    do
         print *, 'Bitte ganze Zahl eingeben, "0"  für Abbruch:'
         read  *, n

         if(n == 0) exit

         res = test(n)

         if(res == .true.) then
            print *, 'Treffer!'
         else
            print *, 'Das war leider nicht perfekt. -:('
         endif
    end do
end program


logical function test(x)
    implicit none
    integer :: i,x
    integer, dimension(x/2) :: m

    m=0
    test=.false.

    forall(i=1:x/2, mod(x,i) == 0) m(i)=i

    if(sum(m) == x) test = .true.



end function
! ------------------ Dateiende ----------------------------
```

Einführung und Literatur

Diese Vorlesung ist der Versuch, eine parallele Einführung in C und Fortran zu geben. Die Hoffnung ist, daß abstrakte Konzepte (Datentypen, Ablaufsteuerung, Parameterübergabe an Unterprogramme etc.) durch eine Gegenüberstellung zwei verschiedener konkreter Realisierungen klarer hervortreten.

Schwerpunkt ist die Anwendung der Sprachen zur Lösung von Problemen der numerischen Mathematik. Deshalb wird z.B. der Umgang mit Zeichenketten und Bitmustern nur kurz gestreift, der Umgang mit Feldern und Matrizen dagegen genauer behandelt.

Wert soll gelegt werden auf einen sauberen Programmierstil: Modularisierung, Trennung von Ein/Ausgabe-Routinen und Algorithmen, standardkonforme und betriebssytemunabhängige Programmierung, wiederverwendbaren Code, Verwendung existierender Bibliotheken u.a.

Die Bedeutung von C bedarf wohl keiner Begründung. Im Gegensatz dazu ist Fortran in letzter Zeit "unpopulär" geworden. Es ist jedoch die Programmiersprache, die sich unmittelbar an den Bedürfnissen von numerischen Problemen orientiert. Auf diesem Gebiet hat sie viele Vorteile gegenüber anderen Sprachen, die sich auch in kürzeren Entwicklungszeiten und schnellerem Code niederschlagen.

Es gibt zahlreiche Bücher über C und Fortran. Herausgestellt sei hier nur der C-Klassiker (es exisitieren auch deutsche Übersetzungen)

- B.W. Kernighan und D. Ritchie, *"The C Programming Language"*

und für Fortran

- M. Metcalf und J. Reid, *"Fortran 95/2003 explained"*

Eine umfassende Einführung in numerische Programmierung ist

- Ch. Überhuber, *"Computernumerik"*, 2 Bände, Springer 1995

Man findet im WWW ebenfalls leicht einführende Texte, Tutorials und Informationen. Hier einige Empfehlungen:

- M. Metcalf, *Fortran 90 Tutorial*,
  `http://wwwasdoc.web.cern.ch/wwwasdoc/f90.html`
- C.G. Page, *Professional Programmer's Guide to Fortran77*,
  `http://www.star.le.ac.uk/~cgp/fortran.html`
  Eine kompakte Darstellung der alten Fortranversion Fortran77. Zahlreiche Bibliotheken und numerische Programme liegen noch in diesem Dialekt vor.
- T. Love, *ANSI C for Programmers on UNIX Systems*,
  `http://www-h.eng.cam.ac.uk/help/documentation/docsource/teaching_C.pdf`
- Die C-FAQ (Frequently Asked Questions List), auf Deutsch
  `http://www.dclc-faq.de/inhalt.htm` und English `http://c-faq.com/`
- Die Fortran-FAQ `http://www.faqs.org/faqs/fortran-faq/`
- (Fast) alles über Fließkommazahlen:
  W. Kahan, *IEEE Standard 754 for Binary Floating-Point Arithmetic*,
  `http://http.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps`
- LAPACK - Das Softwarepaket zur Linearen Algebra `http://www.netlib.org/lapack/`
- Netlib - Die Sammlung numerischer Software `http://www.netlib.org/`

1 Geschichte von C und Fortran

1.1 Fortran

1957 John Backus bei IBM entwickelt eine Sprache zur “**For**mula **tran**slation”

1966 ANSI (American National Standards Institute, vgl. DIN in Deutschland) veröffentlicht den ersten Standard für eine Programmiersprache überhaupt: Fortran 66

1978 ANSI Standard X3.9-1978 definiert Fortran 77

1980 Die International Standards Organization ISO adoptiert diesen Standard im Dokument ISO 1539-1980

80er Jahre Eine Expertengruppe arbeitet an einer durchgehenden Modernisierung und Erweiterung der Sprache unter dem Arbeitstitel Fortran 8X.

1991 Fortran 90 wird von ANSI und ISO als Standard veröffentlicht. Neuheiten gegenüber Fortran 77 umfassen die Einführung von Zeigern und dynamischer Speicherverwaltung, Modulen und Interfaces, intrinsische Funktionen für Felder und vieles mehr.

1995 Eine leichte Erweiterung und Überarbeitung, Fortran 95, wird veröffentlicht und schließlich als Standard ISO/IEC 1539-1:1997 verabschiedet.

2003 Revision des Standards (Fortran 2003) mit Einführung von Objektorientierung, parametrisierten Typen (vgl. “templates” in C++), definierter Schnittstelle zu C-Programmen und mehr.

2008 Fortran 2008 ist in Arbeit

1.2 C

1970–73 Dennis M. Ritchie bei AT&T entwickelt C.

1978 “The C Programming Language” von B.W. Kernighan und D. M. Ritchie erscheint. Die in der 1. Auflage beschriebene Sprache weicht in einigen kleinen Details vom späteren Standard ab. Dieser C-Dialekt wird heute oft “K&R C” genannt.

80er Jahre C findet weite Verbreitung. Bjarne Stroustrup nimmt C als Grundlage für objektorientierte Erweiterungen und entwickelt C++.

1989 Der ANSI C Standard X3.159-1989 wird verabschiedet.

1990 Die International Standards Organization ISO adoptiert diesen Standard im Dokument ISO/IEC 9899:1990

1999 Eine überarbeitete und erweiterte Version, C99 genannt, wird von ISO vorgeschlagen.

1.3 Gegenwärtige Situation

- ANSI C-1989 ist weit verbreiteter Standard. Die Erweiterungen von C99 werden noch wenig genutzt, obwohl sie einige für die numerische Programmierung nützliche Features enthalten (variable length arrays etc.).
- Fortran95/2003 ist verbreitet, viele wichtige Bibliotheken liegen immer noch in Fortran77 vor.
- Diese Vorlesung behandelt C99 und Fortran95.

2 Grundlagen

Ein Programm besteht aus einer Menge von Funktionen, die in einer oder mehreren Quelltextdateien (source code files) untergebracht sind. Zusätzliche Funktionen können durch Programmbibliotheken (libraries) bereitgestellt werden.

Funktionen bestehen aus Definitionen, Deklarationen und Anweisungen.

Es gibt eine ausgezeichnete Funktion, die den Namen `main` hat. Diese wird beim Programmstart vom Betriebssystem aufgerufen.

Es gibt eine ausgezeichnete Funktion, die mit der Anweisung `program <name>` eingeleitet wird. Diese wird beim Programmstart vom Betriebssystem aufgerufen.

Funktionen können in Modulen zusammengefaßt werden.

Außer den Funktionsdefinitionen kann eine Quelldatei noch

- Präprozessoranweisungen
- globale Definitionen und Deklarationen
- Prototypen (Deklarationen von Funktionen)
- `typedef`-Anweisungen

- `module`-Definitionen
- `block data`-Definitionen
- `common block`-Definitionen

enthalten.

Definition und Deklaration

Eine Variable/Funktion ist definiert, wenn sie erzeugt wird und Speicherplatz für sie zur Verfügung gestellt wird. Eine Variable/Funktion wird deklariert, wenn man dem Compiler nur ihren Typ mitteilt und sich ihre Definition an anderer Stelle (z.B. in einer anderen Quelltextdatei oder Bibliothek) befindet.

Namen von Variablen und Funktionen

Namen von Variablen und Funktionen können aus den lateinischen Buchstaben `a-zA-Z`, dem Unterstrich `_` und Ziffern bestehen. Das erste Zeichen darf keine Ziffer sein.

Groß- und Kleinbuchstaben werden unterschieden.

Groß- und Kleinbuchstaben werden nicht unterschieden (`Nmax` und `NMAX` ist dieselbe Variable).

Syntax

Anweisungen enden mit Semikolon. Zeilenenden haben keine syntaktische Bedeutung. Das Semikolon **beendet** Anweisungen.

Jede Zeile enthält eine Anweisung. Wenn eine Zeile mit einem `&` endet, wird die Anweisung auf der nächsten Zeile fortgesetzt. Mehrere Anweisungen pro Zeile sind möglich und müssen mit Semikolon getrennt werden. Das Semikolon **trennt** Anweisungen.

Sprungmarken (labels)

Label sind Namen, die durch Doppelpunkt getrennt vor einer Anweisung stehen:

```
    if( y>0 ) goto M1;
    ...
 M1: x = sin(y);
```

Label sind Nummern aus bis zu 5 Ziffern, die vor der Anweisung stehen:

```
    if( y > 0) goto 23
    ...
 23 x = sin(y)
```

Kommentare

Kommentare beginnen mit `/*` und enden (eventuell nach mehreren Zeilen) mit `*/`. Sie können nicht verschachtelt sein (`/*` und `*/` haben keine "Klammersyntax"), das erste `*/` in einem Kommentar beendet ihn.
Kommentare können sich auch von `//` bis zum Zeilenende erstrecken.

```
    /* das ist ein Kommentar */
    /* das
       /* auch */
     das nicht mehr */
    x=x+1; // Kommentar
```

Kommentare erstrecken sich von einem `!` bis zum Zeilenende.

```
   x = sin(y)  ! Kommentar
```

Funktionen und Blöcke

Geschweifte Klammern {} fassen Definitionen und Anweisungen zu einem Block zusammen. An jeder Stelle, an der eine Anweisung stehen darf, kann auch ein Block stehen.
Alle zu einer Funktion gehörenden Anweisungen bilden einen Block:

<typ> *<name>*`(`*<typ1>* *<arg1>*, *<typ2>* *<arg2>...*`) {`
`...`
`}`
Bsp:

```
      double det(int n, double a[n][n]){
      ...
      }
```

Funktionen beginnen mit

<typ> `function` *<name>* `(`*<arg1>, <arg2>,...)*

oder
`subroutine` *<name> (<arg>, <arg2>,...)*

und werden mit
`end function` oder `end subroutine`

beendet. Definitionen und Deklarationen müssen vor der ersten ausführbaren Anweisung stehen.
Bsp:

```
      real(kind=8) function det(n,a)
         integer :: n
         real(kind=8), dimension (n,n) :: a
         ...
      end function
```

3 Vom Quelltext zum ausführbaren Programm

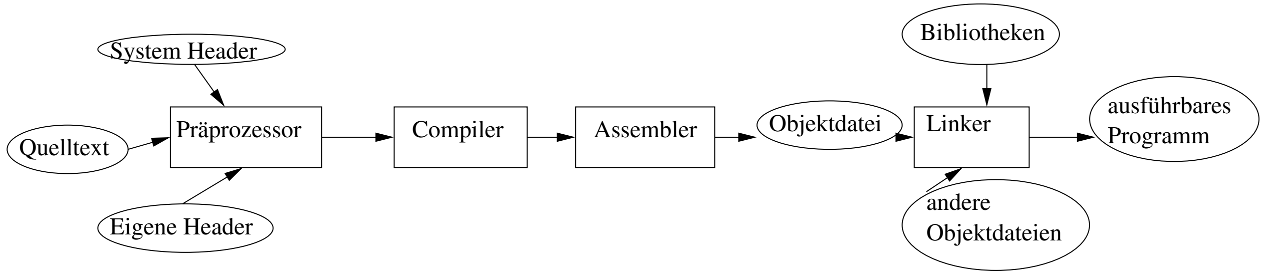

Abbildung 1: Kompilierungsschritte in C

Klassischerweise unterscheidet man zwischen Compiler- und Interpretersprachen. Ein Interpreter liest Anweisung für Anweisung eines Programmtextes ein, übersetzt sie in Maschinensprache und **führt sie sofort aus**. Ein Compiler hingegen transformiert einen Quelltext als Ganzes in eine ausführbare Binärdatei.

Diese Unterscheidung ist heute nicht mehr so scharf möglich: Für viele Sprachen existieren Compiler und Interpreter; die Arbeitsweise vieler Interpreter ist wesentlich komplexer geworden und ähnelt der eines Compilers. Trotzdem kann man wohl sagen, daß C und Fortran typische Compilersprachen sind.

Damit unterscheidet man zwischen Dingen (Fehler, etc.), die während des Compilerlaufs (at compile time) oder erst bei der Programmausführung (at run time = zur Laufzeit) passieren. Triviale Ausdrücke (wie z.B `i=1+1;`) werden – spätestens, wenn man mit eingeschalteter Optimierung kompiliert – bereits vom Compiler ausgewertet.

In C (und in vielen Fortran-Inplementationen) wird als erstes ein Präprozessor-Lauf durchgeführt. Dieser macht reine Textsubstitutionen (Kommentare entfernen, den Inhalt anderer Dateien einfügen etc.), d.h. sein Output ist wieder ein C-Quelltext. Anschließend erzeugt der Compiler daraus eine Objektdatei (Dateiendung .o) und der Linker fügt die Objektdatei mit Bibliotheken (Sammlungen von Objektdateien) und evtl. anderen Objektdateien zu einem ausführbaren Programm zusammen.

Alle diese Schritte können durch ein Kommando erledigt werden:

```
gcc -std=c99 beispiel1.c
```

führt alle diese Schritte aus und erzeugt (falls keine Fehler auftreten) eine Datei mit dem ausführbaren Programm namens `a.out`.

Ein etwas komplexeres Beispiel ist

```
gcc -std=c99 -c -O2 -Wall -I ../includes beispiel2.c
```

-std=c99	aktiviere C99-Sprachumfang
-c	läßt den Linker-Schritt weg (d.h., eine Objektdatei `beispiel2.o` wird erzeugt)
-O2	optimiert den erzeugten Code
-Wall	erzeugt Warnungen bei Anweisungen, die zwar gültiges C sind, aber oft Fehlerquellen darstellen (z.B. `if(a=b)...`)
-I <Pfad>	zusätzlicher Suchpfad für `#include`-Präprozessoranweisungen

Noch ein Beispiel: Hier wird die oben erzeugte Objektdatei `beispiel2.o` weiterverwendet.

```
gcc -std=c99 -o myprog -Wall -O2  beispiel3.c beispiel2.o -lm -L/usr/local/lib -lgl
```

-o <name>	erzeugt ein Programm namens `myprog`
<Dateiliste>	Liste von Quelldateien, die übersetzt, und Objektdateien, die dazugelinkt werden sollen
-l<name>	beim Linken ist die Bibliothek lib<name>, hier also `libm` (die Bibliothek der Mathematik-Funktionen) und `libgl`, zu verwenden.
-L <Pfad>	zusätzlicher Suchpfad für Bibliotheken

Als Fortran95-Compiler steht der Intel Fortran Compiler `ifort` zur Verfügung:

```
ifort -o myprog -CB -O3 beispiel3.f90
```

-CB	überprüfe zur Laufzeit, daß Indexgrenzen von Feldern nicht überschritten werden (check bounds)

Zum Einbinden der Lapack-Bibliotheken ist folgendes Kommando nötig:

```
ifort -o myprog -CB -O3 -llapack -lblas -lg2c beispiel4.f90
```

Statisches vs. dynamisches Linken Ohne die Option `-static` wird dynamisch gelinkt: es wird überprüft, ob alle erforderlichen Bibliotheksfunktionen zur Verfügung stehen, aber sie werden nicht in die Binärdatei kopiert, sondern erst zur Laufzeit geladen – Binärdateien werden wesentlich kleiner und man kann eine Bibliothek durch eine neuere Version ersetzen, ohne Programme neu linken zu müssen.

Der Präprozessor

#include <math.h>	Einfügen der Datei `math.h`. Sie wird in vordefinierten Systemverzeichnissen (oft `/usr/include/`) gesucht.
#include "defs/mydef.h"	Einfügen der Datei `defs/mydef.h`. Sie wird relativ zum aktuellen Verzeichnis gesucht.
#define NMAX 1000	Definition einer Präprozessorkonstanten mit Wert. Die Zeichenfolge `NMAX` wird von jetzt ab im gesamten Quelltext durch die Zeichenfolge `1000` ersetzt. Präprozessorkonstanten können in 3 Zuständen sein: undefiniert, definiert ohne Wert oder definiert mit einem Wert.
#define DEBUG	Definition einer Präprozessorkonstanten ohne Wert.

#ifdef DEBUG
printf(...);
#endif
	Bedingte Compilierung. `#ifdef ...` ist wahr, wenn die Konstante mit oder ohne Wert definiert ist. Es gibt auch `#ifndef NAME`, `#else` und `#undef NAME`.

Präprozessorkonstanten können auch beim Compileraufruf mit der `-D` Option gesetzt werden:

```
gcc -DDEBUG -DNMAX=1000 ...
```

4 Datentypen

Ein Datentyp (kurz Typ) ist eine Menge von Werten und darauf definierten Operationen. Z.B. umfaßt der Typ `int` (in C) bzw. `integer` (Fortran) auf einer 32bit-CPU die Werte $-2147483648 \ldots + 2147483647$ ($-2^{31} \ldots 2^{31} - 1$) und als Operationen die übliche Arithmetik ganzer Zahlen.[2]

Etwas weniger abstrakt gesehen ist der Typ eines Objektes die Vereinbarung, wieviel Speicherplatz es benötigt und wie die Bitmuster, die den Inhalt dieses Speicherbereichs darstellen, zu interpretieren sind.

Neben den fundamentalen Typen kann eine Programmiersprache die Definition eigener, abgeleiteter Typen erlauben.

C und Fortran sind beides Sprachen mit einem strengen und statischen Typensystem: prinzipiell hat jede Variable und jede Funktion einen Typ und dieser kann während des Programmablaufs nicht verändert werden.

Die beiden Sprachen sind allerdings unterschiedlich konsequent, was die "Strenge" der Typüberprüfung (insbesondere von Funktionen) betrifft und haben jede ihre eigenen Hintertürchen zum Austricksen des Typsystems.

Sowohl C als auch Fortran kennen die 5 Grundtypen **Zeichen, Boolsche Variable, Ganze Zahl, Fließkommazahl** und **Komplexe Fließkommazahl**. Die genaue Implementierung kann von der Art der CPU und vom Compiler abhängen. Die folgende Tabelle gibt einen ersten Überblick:

C	Fortran	Bits	Bytes	Bereich
char	character	8	1	(-128 ... 127)
unsigned char		8	1	(0 ... 255)
short int	integer(kind=2)	16	2	-32768 ... 32767
unsigned short int		16	2	0 ... 65535
int, long int	integer	32	4	-2147483648 ... 2147483647
long long int	integer(kind=8)	64	8	$-2^{63} \ldots 2^{63} - 1$
float	real	32	4	$\pm 1.1 \times 10^{-38} \ldots \pm 3.4 \times 10^{38}$
double	real(kind=8)	64	8	$\pm 2.3 \times 10^{-308} \ldots \pm 1.8 \times 10^{308}$

Tabelle 1: Beispiele für Datentypen, Intel 32Bit Architektur (IA32), Gnu C Compiler (GCC) und Intel Fortran Compiler (IFort)

Ganze Zahlen

C kennt die 4 Typen `short int, int, long int, long long int` sowie jeweils noch eine `unsigned`-Variante, die nur positive Werte annimmt: `unsigned short int, unsigned int`, usw.
GCC auf IA32 implementiert sie als

`short int`	2 Byte
`int`, `long int`	4 Byte
`long long int`	8 Byte

[2]Genauer: Arithmetik modulo 2^{32}: $2147483640 + 10 = -2147483646$

In Fortran wird die Art der Implementierung durch den `kind`-Parameter festgelegt. Seine Bedeutung ist implementierungsabhängig.[a] IFort auf IA32 kennt

`integer(kind=1)`	1 Byte
`integer(kind=2)`	2 Byte
`integer, integer(kind=4)`	4 Byte
`integer(kind=8)`	8 Byte

[a]Eine implementierungsunabhängige Anforderung eines minimalen Wertebereichs ist mit der Funktion `selected_int_kind` möglich. s.S. 11

Fließkommazahlen

`float`	4 Byte
`double`	8 Byte
`long double`	10/12 Byte[a]

[a] s. Anhang 1

`real, real(kind=4)`	4 Byte
`real(kind=8), double precision`	8 Byte
`real(kind=16)`	16 Byte

Komplexe Fließkommazahlen

sind Paare (Re, Im) von Fließkommazahlen.

```
#include <complex.h>
```

`complex float`	8 Byte
`complex double`	16 Byte
`complex long double`	20/24 Byte

`complex, complex(kind=4)`	8 Byte
`complex(kind=8), double complex`	16 Byte
`complex(kind=16)`	32 Byte

Boolesche Variablen

```
#include <stdbool.h>
```

`bool` mit den 2 Werten `true` und `false`

`logical` mit den 2 Werten `.true.` und `.false.`

Zeichen und Strings (Zeichenketten)

```
char x='a'
char name[5]="Hans"
char Name[]="Hans"
```

(Compiler ermittelt Länge selbst)

```
character :: x
character(len=4) :: name
character(4) :: name2
name='Hans'
name2="Hans"
x='a'
```

Implizite Deklarationen in Fortran

Wenn eine Variable nicht deklariert ist, wird ihr vom Fortran-Compiler ein Standardtyp zugeordnet: Sie wird als `integer` angesehen, wenn ihr Name mit `i,j,k,l,m,n` anfängt, sonst als `real`. Dieses Feature führt z.B. dazu, daß der Compiler keine Fehlermeldung ausgibt, wenn man sich bei einem Variablennamen verschreibt. Man hat dann implizit einen neue Variable eingeführt. Es läßt sich durch die Anweisung

`implicit none`

am Anfang jeder Funktion/Subroutine abschalten.

Maschinenunabhängige `kind`-Parameter in Fortran
Intel Fortran verwendet als `kind`-Parameter einfach die Zahl der Bytes. Das ist aber vom Standard nicht vorgeschrieben, andere Compiler können das anders handhaben. Compilerunabhängig wird man durch die Verwendung zweier spezieller Funktionen:
`integer, parameter :: k10 = selected_int_kind(10)`
`k10` ist der `kind`, der mindestens 10-stellige ganze Zahlen aufnehmen kann (für IFort also gleich 8). Wenn eine Implementierung den geforderten Typ nicht bereitstellen kann, gibt die Funktion -1 zurück.
Analog gibt liefert `selected_real_kind(n)` den `kind`-Wert mit mindestens n gültigen Ziffern:
`integer, parameter :: dbl = selected_real_kind(15)`

Konstanten

Ganzzahlige Konstanten

Integerkonstanten sind vom Typ `int`. Wenn ein `L` oder `l` nachgestellt wird, sind sie vom Typ `long int`. Mit Präfix `0` (Null) werden sie als Oktalzahlen interpretiert, mit einem Präfix `0x` als Hexadezimalzahlen:
`100, 100L, 020 (=16), 0xFF (=255)`

Binärzahl	`b'101'`	(=5)
Oktalzahl	`o'20'`	(=16)
Hexadezimalzahl	`z'FF'`	(=255)

Ein `kind`-Parameter kann mit einem Unterstrich nachgestellt werden:
`123_8`, `123_k10`: Konstante vom Typ `integer(kind=8)` bzw `integer(kind=k10)`

Gleitkommakonstanten

Beispiele in C und Fortran: `1673.0333e-7, 1e+10, 1., -1.e12, .78`

Die Konstanten sind vom Typ `double`. Nachgestelltes F oder L macht sie zum Typ `float` oder `long double`.
Komplexe Konstanten werden mittels `I` geschrieben:
`3.14 - 1.23e-3 * I`

Die Konstanten sind vom Typ `real`. Andere `kinds` bekommt man, indem die geforderte `kind`-Konstante mit Unterstrich nachgestellt wird: `1.34e+17_8` oder `1.34e+17_long`.
Komplexe Konstanten werden als Paare angegeben: `( 3.14, -1.23e-3)`

Zeichenkonstanten

Einzelne Zeichen werden in '...' eingeschlossen und sind vom Typ `int`[a]: `'0', 'A', '"', '\0'` (NUL), `'\n'` (Newline-Zeichen), `'\\'` (Backslash), `'\nnn'` (Beliebiges ASCII-Zeichen durch 3 Ziffern im Oktalsystem dargestellt)
Zeichenketten (strings) werden in "..." eingeschlossen: `"bla bla"`. Intern wird ein Feld von Zeichen erzeugt, das ein Byte länger ist und mit `\0` als Endezeichen abgeschlossen wird.

[a] Ja: `int`, nicht `char`. Näheres später unter "impliziter Typumwandlung".

Zeichen und Zeichenketten werden in '...' oder "..." eingeschlossen. Es gibt keine spezielle Kennung am Ende.

5 Zusammengesetzte Typen

5.1 Homogene Typen (Felder = arrays)

Homogen: Alle Komponenten sind vom selben Typ.
Ein Feld von 10 ganzen Zahlen:

```
int a[10];
```

Die zehn Elemente sind als
`a[0], a[1], ... , a[9]` ansprechbar.

```
integer, dimension(10) :: a
```

Die Elemente sind `a(1), a(2), ... a(10)`.

Andere Indexbereiche können definiert werden:

```
integer, dimension(-2:5) :: a
```

hat 8 Elemente `a(-2), a(-1), a(0),...,a(5)`

Mehrdimensionale Felder:

```
int a[2][3];
```

Die 6 Elemente sind zeilenweise gespeichert, d.h. in der Reihenfolge `a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2]`

```
integer, dimension(2,3) :: a
```

Die Elemente sind spaltenweise gespeichert, der erste Index variiert am schnellsten: `a(1,1), a(2,1), a(1,2), a(2,2), a(1,3), a(2,3)`.
Andere Indexbereiche können definiert werden:

```
integer, dimension(-2:5, 2:4) :: a
```

5.2 Inhomogene Typen

Komponenten haben Namen statt Indizes

```
struct Datum {
      short int Jahr;
      char Monat[4];
      short int Tag;
};                        // Definition eines Typs "Datum"


struct Datum Geburtstag; // Definition von Variablen diesen Typs

struct Datum heute = { 2004, "Feb", 29 };
Geburtstag.Monat = "Jan";
if(heute.Jahr > 1990 ) {...
```

```
type Datum
     integer :: Jahr
     character(len=3) :: Monat
     integer :: Tag
end type

type(Datum) :: Geburtstag, heute

heute = Datum( 2004, 'Feb', 29)
Geburtstag%Monat = 'Jan'
if(heute%Jahr > 1990) ...
```

Verschiedene Daten am selben Speicherplatz

```c
union {
      char ByteFeld[4];
      int GanzZahl;
} word;
```

Verwendung analog zu struct: word.GanzZahl=1 etc.

```fortran
! Feld aus 4 Zeichenketten der Länge 1
   character(len=1), dimension(4) ::  ByteFeld
   integer :: GanzZahl
   equivalence (ByteFeld, GanzZahl)
```

Die 4 Bytes `ByteFeld[0]...[3]` und die 4 Bytes von GanzZahl belegen denselben Speicherplatz.

6 Ablaufsteuerung

6.1 Verzweigungen

```c
if ( x > 3 || y != 0  ) z=3;

if (x > 0) {
   y = sqrt(x);
   z = 0;
}

if (i > 5 && j == 0)
   y = 0;
else {
   y = 3.3;
   z = 0;
}
```

```c
res = <Ausdruck1> ? <Ausdruck2> : <Ausdruck3>
```

Wenn Ausdruck1 wahr ist, wird Ausdruck2 zugewiesen, sonst Ausdruck3. Beispiel:

```c
maxXY = x > y ? x : y;
```

```fortran
if( x>3 .or. y /= 0 ) z=3

if (x > 0) then
    y = sqrt(x)
    z = 0
end if

if (i > 5 .and. j == 0) then
    y = 0
else
    y = 3.3
    z = 0
end if

if (i > 5 .and. j == 0) then
    y = 0
else if ( i == 0 ) then
    y = 5
else if ( i == 3 ) then
    y = 7
else                 ! catch rest
    y = 3.3
    z = 0
end if
```

6.2 Mehrfachverzweigungen

```
switch (i) {
      case -3:
      case -2:
      case -1:
                x=0;
                break;
      case 0:
                x=1;
                break;
      case 1:
      case 2:
                x=3;
                break;
      default:
                x=4;
}
```

```
select case (i)


     case (-3:-1)
                  x = 0

     case (0)
                  x = 1


     case (1:2)


                  x = 3
     case default
                  x = 4
end select
```

Der zu testende Ausdruck muß ganzzahlig oder ein Zeichen (Byte) sein. Das Argument von `case` muß eine Konstante desselben Type sein.

6.3 Schleifen

`while (` *<Ausdruck>* `)` *<Anweisung oder Block>*

`do` *<Anweisung oder Block>* `while (`*<Ausdruck>*`)`

`for (`*<Initialisierung>*`;` *<Test>*`;` *<Update>* `)` *<Anweisung oder Block>*

In C kann es bei den Varianten `while` und `for` sein, daß der Schleifenkörper nie abgearbeitet wird, bei der `do...while`-Version wird der Schleifenkörper auf jeden Fall einmal durchlaufen.
Eine `for`-Schleife `for(i=0;i<10;i++){...}` wird abgearbeitet als:

- Initialisierung i=0
- Test i<10? Wenn ja, Abarbeitung des Schleifenkörpers `{...}`, sonst Ende der `for`-Schleife.
- Update i=i+1
- Test, Schleifenkörper, Update, Test, Schleifenkörper,. . . ,Test

Natürlich kann man das Update auch als letzten Befehl in den Schleifenkörper schreiben: `for(i=0;i<10;){...; i++;}`.
Man kann die Initialisierung vorher machen: `i=0; for(;i<10;){...; i++;}`
und das ist äquivalent zu `i=0; while(i<10){...; i++;}`

`do` *<var>* `=` *<Start>*`,` *<Ende>*
    <Anweisungen>
`end do`

Es gibt auch die Möglichkeit, eine Schrittweite ungleich 1 anzugeben:
`do` *<var>* `=` *<Start>*`,` *<Ende>*`,` *<Schrittweite>*
Diese kann auch negativ sein.
Im Unterschied zu C wird die Anzahl der Schleifendurchläufe am Anfang des Loops aus den Angaben `<Start>,<Ende>` und evtl. `<Schrittweite>` berechnet. Zuweisungen an den Schleifenzähler innerhalb des `do`-Blocks haben keine Auswirkung auf die Anzahl der Durchläufe. Nach dem Durchlaufen der Schleife hat der Zähler den Wert *<Ende>*+1 (bzw. *<Ende>* + *<Schrittweite>*).

6.4 Sprunganweisungen

Sowohl C als auch Fortran kennen die unbedingte Verzweigung `goto <label>`. Wie bei allen Verzweigungen muß das Ziel innerhalb derselben Funktion liegen.

Es gibt Anweisungen, mit denen eine Schleifeniteration oder die ganze Schleife vorzeitig verlassen wird:

```
for(i=0; i<100; i++) {
    y = func(2*i);
    if(y > 2) break;     // Schleife wird vorzeitig abgebrochen
    if(a[i]<0) continue; // Start der nächsten Schleifeniteration
    a[i]=sqrt(a[i]);
}
```

```
do i=0,99
    y = func(2*1)
    if( y > 2 ) exit    ! Schleife wird vorzeitig abgebrochen
    if(a(i) < 0) cycle  ! Start der nächsten Schleifeniteration
    a(i) = sqrt(a(i))
end do
```

7 Arithmetische Operatoren

Wenn die beiden Operanden der binären arithmetischen Operatoren (`+,-,*,/`) nicht vom gleichen Typ sind, wird ein Operand umgewandelt, wobei die Umwandlung immer in Richtung zum längeren Datentyp erfolgt (z.B. `char -> int, int -> float, float -> double`). Das Ergebnis hat denselben Typ wie die beiden Operanden. Bei einer Zuweisung erfolgt anschließend wenn nötig die Anpassung an den Datentyp der linken Seite. Dies kann also auch eine Abwärtskonversion sein.

Multiplikation/Division bindet natürlich stärker als Addition/Subtraktion, ansonsten sind die Operationen linksassoziativ: `a / b / c = (a / b) / c`. Ganzzahlige Division liefert eine ganze Zahl als Ergebnis, der gebrochene Anteil wird abgeschnitten. Damit ist
`3 / 4 * 5.0 = 0 * 5.0 = 0.0` und `5.0 * 3 / 4 = 15.0 / 4 = 3.75`.

Rest der ganzzahligen Division:
`a % b`

Potenz: `pow(a,b)`
ist äquivalent zu `exp(b*log(a))`

Die mathematischen Funktionen `sqrt, log, log10, exp, sin, cos, tan, asin, sinh, ...` sind alle vom Typ `double f(double)`. Ihre Deklarationen sind in `math.h` zu finden.

Die Betragsfunktion `abs` ist vom Typ
`int abs(int)`! Für Fließkommazahlen muß man `fabs` verwenden.

Rest der ganzzahligen Division:
`mod(a, b)`

Potenz: `a**b`
für kleine ganzzahlige b wird dies vom Compiler in Multiplikationen umgewandelt, ansonsten ist es equivalent zu `exp(b*log(a))`.

Die mathematischen Funktionen `abs, sqrt, log, log10, exp, sin, cos, tan, asin, sinh, ...` sind sogenannte generische Funktionen: der Typ des Ergebnisses ist gleich dem Typ des Arguments.

7.1 Komplexe Zahlen

```
#include <complex.h>

complex double a, b, c;
double xabs;

a = 10 - 17*I;
b = csqrt(a + 2*conj(a) + csin(a) + ccos(a));
c = cexp(3.14*I) + clog(a);
xabs = cabs( 3 + 10.2*I) + creal(a) + cimag(a);
```

```
complex(kind=8) :: a,b,c
real(kind=8) :: xabs

a = (10, -17)
b = sqrt(a + 2*conjg(a) + sin(a) + cos(a))      ! Wurzel, Konjugierte, sin, cos
c = exp(cmplx(0, 3.14)) + log(a)                ! exp, log
xabs = abs(cmplx(3, 10.2)) + real(a) + aimag(a) ! Betrag, Realteil, Imaginärteil
```

8 Logische Operatoren und Vergleichsoperatoren

In C kann der Wert jedes Ausdrucks logisch interpretiert werden: `0 (int)`, `0.0 (float)` oder `\0 (char)` bedeuten `false`, alles andere wird als `true` interpretiert.
Umgekehrt: jeder Vergleichsausdruck hat den numerischen Wert 0 oder 1, damit sind Konstruktionen wie `x + (y >= 1 )` möglich.
Vergleiche: `>, <, >=, <=, ==, !=`
Logische Operatoren: `||, &&, !`

Vergleiche erzeugen Werte vom Typ `logical`, Argumente von `if()` müssen von diesem Typ sein:

```
logical isodd
isodd = mod(x,2) .eq. 1
if(isodd) then ...
```

Vergleiche: `>, <, >=, <=, ==, /=`
Logische Operatoren: `.or., .and., .not., .eqv.` (logische Gleichheit), `.neqv.` (logisch ungleich, exklusives oder)

Logische Operatoren werden von links nach rechts ausgewertet, allerdings nur solange, bis ihr Wahrheitswert feststeht. Das spart Rechenzeit – und ermöglicht in C Konstrukte wie `(x >= 0) && (y = sqrt(x))`

9 Zeiger und dynamische Speicherverwaltung

Zeiger enthalten die Adresse eines Objekts (sie ”zeigen” darauf).

```
char * cx;        // Zeiger auf ein Objekt vom Typ char
int  * ip;        // Zeiger auf eine ganze Zahl
double * fx[10];  // Feld von 10 Zeigern auf Double-Zahlen
char * p="Hans";  // Zeiger auf char mit Initialisierung
double ** fz;     // Zeiger auf einen Zeiger auf eine Double-Zahl
void * p;         // generischer Zeiger
struct Datum * d; // Zeiger auf eine Struktur
int (*fp)(float); // Zeiger auf Funktion mit float-Argument, die
                  // int-Wert zurückgibt
```

```
character, pointer :: cx
integer, pointer :: ip
real(kind=8), pointer, dimension(:) :: fxx ! Zeiger auf ein 1-dim Feld
                                           !  von Double-Zahlen
real, pointer, dimension(:,:) :: ax        ! Zeiger auf ein 2-dim Feld

type(Datum), pointer :: dat ! Zeiger auf einen zusammengesetzten Typ
```

- Der Typ `void` dient a) zur Deklaration generischer Zeiger und b) zur Deklaration von Funktionen ohne Rückgabewert:
  `void swap(int a, int b){ ...}`
- Zeiger auf Felder sind in C unüblich. Man verwendet eine Zeiger auf das erste Element.
- Zeiger können miteinander verglichen oder voneinander subtrahiert werden. Weiterhin können ganze Zahlen addiert oder subtrahiert werden. Zeigerarithmetik geschieht dabei immer in Speichereinheiten des Types, auf den der Zeiger zeigt: `ipn = ip + 1` zeigt auf das nächste Feldelement, d.h. es wird z.B. um 4 Bytes weitergezählt.

- Es gibt keine generischen Zeiger.
- Zeiger auf Felder und Zeiger auf Elemente der Felder sind verschiedene Typen.
- Es gibt keine Zeiger auf Funktionen. (Das ändert sich in Fortran 2003.)
- Es gibt keine Zeigerarithmetik.

Der Wert Null hat eine besondere Bedeutung für Zeiger: Nullzeiger sind initialisiert, zeigen aber auf nichts. Definitionen von Zeiger reservieren nur Speicherplatz für den Zeiger selbst, nicht für das Objekt, auf das er zeigen soll.
Entweder man weist einem Zeiger die Adresse eines bereits anderweitig deklarierten Objekts zu:

```
int i=10;
ip = &i;  // Jetzt hat ip einen Wert:
          // die Adresse von i
ip = 0;   // Nullzeiger
```

Der `&`-Operator liefert die Adresse eines Objekts.

```
integer, target  :: i   ! Zeiger können nur auf Objekte zeigen,
                        ! die das target-Attribut haben.
ip => i                 ! ip zeigt auf i
ip => null()            ! Nullzeiger
```

Der `=>`-Operator weist einem Zeiger die Adresse eines Objekts zu. Das ist nur zulässig bei Objekten, die mit dem `target`-Attribut deklariert sind.

oder der Speicherplatz muß dynamisch zur Laufzeit des Programms angefordert werden:

```
#include <stdlib.h>  // Prototypen für malloc, free
  int  n = 100;
  double * ff;  // Zeiger auf Double-Zahlen

      // Speicherplatz für n Double-Zahlen wird
      // reserviert: n * (Länge von double) Bytes
  ff = malloc(n*sizeof(double));
  for(int i=0; i<n; i++) {
      ff[i]=...           // Zeiger kann wie Feld
                          // genutzt werden
  }
  ...
  free(ff);    // dynamischer Speicher wird
               // wieder freigegeben
```

(`malloc` hat den Typ `void * malloc(int)`: es liefert einen generischen Zeiger.)

```
integer :: n,i
real(kind=8), pointer, dimension(:) :: ff  ! Zeiger auf Double-Feld
n = 100


allocate(ff(n))                            ! Speicherplatz für
                                           ! n Zahlen
do i=1, n
   ff(i) = ...
   ...
end do
deallocate(ff)       ! Speicherfreigabe
```

Der *-Operator, angewandt auf einen Zeiger, liefert das Objekt, auf das der Zeiger zeigt:

```
int i, *ip, vec[9]; /* ip ist ein Zeiger auf eine ganze Zahl */

ip   = &i;          /* Jetzt hat ip einen Wert: er zeigt auf i */
*ip += 3;           /* Damit wird i (das, worauf ip gerade zeigt) um 3 erhöht. */
j = 2*(*ip);        /* äquivalent zu j=2*i */
ip   = &vec[3];     /* Jetzt zeigt ip auf das 4. Element von vec. */
ip--;               /* Jetzt zeigt ip auf das 3. Element von vec. */
*ip  = 10;          /* vec[2] hat nun den Wert 10. */
```

Es gibt keinen expliziten Dereferenzierungsoperator vergleichbar mit * in C. In allen arithmetischen und ähnlichen Anweisungen kann ein Zeiger anstelle des Objekts stehen, auf das er zeigt:

```
integer, target  :: i,j
integer, pointer :: ip
ip => i                    ! ip zeigt auf i
ip = 3                     ! nun ist i gleich 3
j  = ip+2                  ! und j gleich 5
```

10 Einige Besonderheiten in C

- Eine Zuweisung <*Variable*> = <*Ausdruck*> ist in C ebenfalls ein Ausdruck, sein Wert und Typ ist der Wert/Typ der linken Seite. Damit sind Mehrfachzuweisungen möglich, wobei = rechtsassoziativ ist: `a = b = c = d` wird ausgewertet als `a = ( b = ( c = d ) )`
(daher liefert `int a; float b,c; b=a=c=3.14;` für b den Wert3.0!)

- Ausdrücke der Form `x = x • y` können als `x •= y` geschrieben werden.

- Die Zuweisungen `i=i+1` und `i=i-1` können auch als `++i` oder `--i` geschrieben werden. Die Postfix-Form `i++` oder `i--` hat die gleiche Wirkung auf `i`, ihr **Wert** als Ausdruck ist jedoch `i` und nicht `i+/-1`. Anders gesagt, `y=++x;` ist äquivalent zu `x=x+1; y=x;` Die Variante `y=x++;` jedoch ist äquivalent zu `y=x; x=x+1;`.

- Zwei durch Komma getrennte Ausdrücke werden von links nach rechts ausgewertet. Typ und Wert des **rechten** Ausdrucks bestimmen Typ und Wert des Gesamtausdrucks.

- Diese Regeln erlauben in C sehr kompakte Schreibweisen. Hier ein Beispiel aus dem Code des PostScript-Interpreters `ghostscript`:

```
ccase = (skew > 0 ? copy_right : ((bptr += chunk_bytes), copy_left)) + function;
```

  "Auseinandergenommen" kann man das auch so schreiben:

```
if(skew > 0) {
      ccase = copy_right + function;
} else {
      bptr = bptr + chunk_bytes;
      ccase = copy_left + function;
}
```

- **Das Hauptprogramm** `main` ist vom Typ `int`. Daher muß es mit einer Anweisung enden, die eine ganze Zahl zurückgibt. Dieser Rückgabewert ist der "exit code" für das Betriebssystems und normalerweise wird 0 verwendet, um ein fehlerfreies Ende des Programms zu signalisieren:

  ```
  int main(){
      ...
      return 0;
  }
  ```

  Das Hauptprogramm bekommmt auch Argumente übergeben, die bei Bedarf ausgewertet werden können. Das erste Argument von `main` ist eine ganze Zahl, die Anzahl der Zeichenketten in der Kommandozeile. Das zweite Argument ist ein Feld von Zeigern auf Zeichenketten: die einzelnen "Worte" der Kommandozeile. Der letzte Zeiger des Feldes ist der Nullzeiger. Üblicherweise wird `main` so deklariert:

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

  Wenn z.B. das Programm "my_solve" heißt und mit der Kommandozeile

  ```
   my_solve datei1 -n 100
  ```

  aufgerufen wird, dann ist

  ```
  argc = 4
  argv[0] = "my_solve"
  argv[1] = "datei1"
  argv[2] = "-n"
  argv[3] = "100"
  argv[4] = 0
  ```

11 Felder in Fortran

Rang und Gestalt Die Anzahl der Indizes eines Feldes heißt **Rang** des Feldes: Vektoren haben den Rang 1, Matrizen den Rang 2 etc.[3]
Die Folge $(n_1, n_2, \ldots, b_{rang})$ der Anzahl der Elemente in jeder Dimension heißt **Gestalt** (shape) des Feldes: Ein Feld aus zehn Elementen hat die Gestalt (10), eine 8×6-Matrix die Gestalt $(8, 6)$. Hierbei geht es nur um die Zahl der Elemente, der Laufbereich der Indizes spielt keine Rolle.
Beispiel:

```
real, dimension(-10:5, -20:-1, 0:1, -1:0, 2) :: grid
```

definiert ein Feld `"grid"` mit 5 Indizes, also vom Rang 5, mit $16 \times 20 \times 2 \times 2 \times 2 = 2560$ Elementen und der Gestalt $(16, 20, 2, 2, 2)$.
Ein Datentyp ohne Index heißt auch **Skalar**.

[3]Diese Definition hat nichts mit dem Begriff des Rangs in der linearen Algebra zu tun.

Elementale Operationen und Zuweisungen Viele Funktionen, die wir bisher nur für Skalare betrachtet haben, können auch auf Felder angewandt werden. Dazu müssen Operanden und Ergebnis die gleiche Gestalt haben oder Skalare sein:

```
float a[20], b[20], c[20];

for(int i=0; i<20; i++) a[i] = b[i] + c[i];
for(int i=0; i<20; i++) c[i] = sqrt(a[i]) + 3 * b[i];
for(int i=0; i<20; i++) b[i] = 3.0;
for(int i=0; i<20; i++) c[i] = 2.0/b[i];
```

```
real, dimension(20) :: a, b, c
real, dimension(-10:9) :: d     ! auch 20 Elemente

a = b + c
c = sqrt(a) + 3 * d  !        c(i)=sqrt(a(i))+3*d(i-11)
b = 3.0
c = 2.0 / b
```

Wenn ein Argument ein Skalar ist, wird es so behandelt wie ein Feld der benötigten Gestalt, in dem alle Einträge den gleichen Wert haben. Wichtig ist nur die Gestalt des Feldes und nicht sein Indexlaufbereich, wie das Beispiel mit dem Vektor `d` demonstrieren soll.
Das komponentweise Produkt von Vektoren oder Matrizen, das man auf diese Weise erhält, ist natürlich nicht das mathematische Matrix- oder Skalarprodukt!

Auch eine selbstgeschriebene Funktion kann elemental sein. Dann läßt sie sich ebenfalls (komponentenweise) auf beliebige Felder anwenden.

```
elemental real(kind=8) function hypo(a,b)
real(kind=8), intent(in) :: a,b
hypo = sqrt( a**2 + b**2 )
end function
...
real(kind=8), dimension(100) :: a,b,c
c = hypo(a,b)
```

Eine Funktion muß eine Reihe von Bedingungen erfüllen, damit sie als "elemental" deklariert werden kann. Sie darf zum Beispiel keine Ein-/Ausgabeoperationen durchführen, keine statischen Variablen (Kap. 13) enthalten und natürlich müssen alle von ihr verwendeten Funktionen auch elemental sein. Sie darf ihre Argumente nicht modifizieren und muß dies auch durch `intent(in)` deklarieren (Kap. 17). Bei ihrer Anwendung muß eine `interface`-Definition (z.B. aus einem Modul) verfügbar sein (vgl. Kap. 14).

where-Anweisung Manchmal soll eine komponentenweise Feldoperation nur auf Komponenten angewandt werden, die einer Bedingung genügen:

```
for(int i=0; i<20; i++)
    if(a[i] > 0) a[i] = 1./a[i];
```

```
where( a > 0 )   a = 1./a
```

Es gibt allgemeinere Formen, die analog zu den Formen der `if`-Anweisung aufgebaut sind:

```
for(int i=0; i<20; i++)
    if(a[i] > 0) {
         a[i] = 1./a[i];
         b[i] = 2 * a[i];
    } else {
         a[i] *= 2;
    }
```

```
where (a > 0)
     a = 1./a
     b = 2 * a
elsewhere
     a = 2 * a
end where
```

```
where (a > 0)
     a = 1./a
     b = 2 * a
end where
```

Teilfelder Teilfelder aus einem größeren Feld können spezifiziert werden, indem man den gewünschten Indexbereich angibt:

```
real, dimension (10,20) :: a,b
real, dimension (5)     :: v,w
real, dimension (8, 6)  :: c

w = 5/v + a(1:5, 7)               !  w(i) = 5/v(i) + a(i, 7),           i = 1...5
c = a(2:9, 5:10) + b(1:8, 15:20)  !  c(i,j) = a(i+1, j+4) + b(i, j+14),  i = 1...8, j = 1...6
a(2, 11:15) = v                   !  a(2, i+10) = v(i),                 i = 1...5
v(2:5) = v(1:4)                   !  v(2)=v(1); v(3)=v(2); v(4)=v(3); v(5)=v(4) Hier sind die Werte auf der re. Seite die Werte vor der Zuweisung
a(2,:) = b(3,:)                   !  Ganze Zeile 2 von a wird gleich Zeile 3 von b gesetzt
a(2, 1:20:4) = v                  !  a(2,1)=v(1); a(2,5)=v(2); a(2,9)=v(3); a(2,13)=v(4); a(2,17)=v(5)
```

Eine nützliche Spezialform sind Vektorindizes: An Stelle eines Indexes oder Indexbereiches darf auch ein ganzzahliges Feld vom Rank 1 stehen: Sei `ivec` ein ganzzahliger Vektor der Länge 4 mit den Elementen `ivec(1)=2; ivec(2)=1; ivec(3)=7; ivec(4)=2` . Dann ist `dd(ivec)` ein Feld bestehend aus den 4 Elementen `dd(2),dd(1),dd(7),dd(2)`. Dies kann auch als `dd( (/ 2,1,7,2 /) )` geschrieben werden.

Feldkonstruktoren Die Werte von kleineren Feldern vom Rang 1 kann man direkt auflisten:

```
integer, dimension(5) :: iy, ix = (/ 12,13,14,15,16 /)
iy = (/ 3,4,5,6,7 /)
```

Hier kann man auch eine implizite Schleife verwenden: `(/ (i, i=1,100) /)` ist äquivalent zu `(/ 1,2,...,100 /)` und `(/ (i**2, i=1,10,2) /)` ist äquivalent zu `(/ 1, 9, 25, ..., 81/)`.

forall-Anweisung Die bisher eingeführten Anweisungen erlauben bereits die weitgehende Einsparung traditioneller Do-Schleifen und eine weitgehend "indexfreie" Verwendung von Vektoren, Matrizen usw. Für einige Aufgaben reicht sie aber noch nicht aus, z.B. wenn man einem Vektor die Diagonalelemente einer Matrix zuweisen will (oder umgekehrt). Hier hilft die `forall`-Anweisung:

```
forall( i = 1:n )                        x(i) = a(i,i)
forall( i = 1:n, j = 1:n, y(i,j) /=0.)   x(j,i) = 1.0/y(i,j)
             ! wird nur für die (i,j)-Paare ausgeführt, für die Bedingung erfüllt ist.
forall( i = 1:n )
     a(i) = 2*i
     where (b(i) > 0) c(i) = sqrt(b(i)   ! where kann verwendet werden
end forall                               ! mehrere Anweisungen sind möglich durch end-block
```

`forall`-Anweisungen können auf Mehrprozessormaschinen vom Compiler besser parallelisiert werden als traditionelle Do-Schleifen.

Spezielle Funktionen (Auswahl)

Skalarprodukt `dot_product(a, b)`: Die Argumente müssen Felder vom Rang 1 und gleicher Größe sein. Das Ergebnis ist das Skalarprodukt $\sum_i a_i b_i$ (oder $\sum_i \bar{a}_i b_i$ falls `a` komplex ist).

Matrixprodukt `matmul(a, b)`:

- a hat Gestalt (n, m) und b Gestalt (m, p). Das Ergebnis $\sum_j a_{ij} b_{jk}$ hat die Gestalt (n, p).
- a hat Gestalt (n, m) und b Gestalt (m). Das Ergebnis $\sum_j a_{ij} b_j$ hat die Gestalt (n).
- a hat Gestalt (m) und b Gestalt (m, p). Das Ergebnis $\sum_j a_j b_{jk}$ hat die Gestalt (p).

Transponierte `transpose(a)`: a muß vom Rang 2 (Gestalt (n, m)) sein. Das Ergebnis hat die Gestalt (m, n).

Gestalt `shape(a)`: liefert ein ganzzahliges Rang-1-Feld mit den Gestaltsparametern $(n, m, \ldots)$ des Feldes `a`.

Reduktionsfunktionen: `sum, product, maxval, minval`
Diese Funktionen liefern ein Ergebnis mit kleinerem Rang als das Argument, daher der Name.

```
real, dimension(100) :: x
a = sum(x)              !  a ist die Summe aller 100 Elemente von x
b = maxval(x)           !  b ist das größte Element aus x
c = sum(a, mask = a > 0) !  c ist die Summe aller Elemente a(i) aus x, die die Bedingung a(i) > 0 erfüllen
```

Alle 4 Funktionen `sum, product, maxval, minval` können zwei optionale Argumente haben, `mask` und `dim`. Wird `dim=n` angegeben, wird die Reduktion (Aufsummation etc) nur bezüglich des n-ten Index durchgeführt, Ergebnis ist ein Feld mit um eins kleinerem Rang. Beispiel: Sei `m` eine Matrix mit 4 Zeilen und 7 Spalten (Gestalt $(4, 7)$). Dann ist `sum(m, dim=1)` ein Feld aus 7 Elementen, den Spaltensummen und `sum(m, dim=2)` ist ein Feld aus 4 Elementen, den Zeilensummen.

12 Ein- und Ausgabe

Dateien werden geöffnet, gelesen oder/und geschrieben und wieder geschlossen. Nach dem Öffnen einer Datei nimmt man in C durch einen FILE-Zeiger und in Fortran durch eine Kanalnummer in allen Lese/Schreib-Operationen darauf Bezug.

Öffnen einer Datei:

```c
#include <stdio.h> // Deklarationen der Standard-
                   // Input/Output Funktionen, des Types FILE usw.
...
FILE * fp1, * fp2, * fp3;  // fp1,... sind Zeiger auf
                           // spezielle FILE-Objekte

fp1 = fopen("beispiel.dat", "r");
     // Die Datei beispiel.dat wird zum Lesen geöffnet

fp2 = fopen("beispiel2.dat", "w");
     // Die Datei wird zum Schreiben (durch Neuanlegen oder
     // Überschreiben) geöffnet

fp3 = fopen("beispiel3.dat", "a");
       // Die Datei wird zum anhängendem Schreiben geöffnet
```

```fortran
 open(unit=4, file='bsp.dat', status='old')
     !  Kanal 4 wird mit der Datei bsp.dat verbunden,
     !  welche bereits existieren muss.

open(unit=17, file='bsp2.dat', status='new', err=12)
     !  Kanal 17 wird mit bsp2.dat verbunden, sie darf  nicht
     !  existieren. Wenn dabei ein Fehler auftritt,
     !  wird "goto 12" ausgeführt.

open(unit=19, file='b3', status='unknown', position='append')
     !  Wenn b3 schon existiert, wird beim Schreiben angehängt
```

Der Dateiname ist eine Zeichenkette, kann also auch eine Variable vom Typ Zeichenkette sein. In C gibt `fopen()` einen Nullzeiger zurück, falls ein Fehler beim Öffnen auftritt.

Mit Hilfe des FILE-Zeigers/der Kanalnummer kann man nun aus einer geöffneten Datei lesen:

```c
fscanf(fp1, <Format>, <Zeiger>...);
fclose(fp1);
```

```fortran
read(4, *) <var1>,<var2>,...
close(4)
```

Mit `fclose()/close()` wird sie wieder geschlossen. Danach ist der FILE-Zeiger/die Kanalnummer nicht mehr benutzbar.

Ein Spezialfall sind die Standardeingabe-, Standardausgabe- und Fehlerausgabe-Kanäle (`stdin, stdout, stderr` im C-Jargon): diese stellt das Betriebssystem als geöffnete Kanäle zur Verfügung und sie dienen zur interaktiven Ein- und Ausgabe[4].

Verwendung von `scanf()` und `printf()` (analog zu `fscanf() und fprintf()`, nur ohne das erste Argument) liest von der Standardeingabe und schreibt auf die Standardausgabe.
`fprintf(stderr,...)` schreibt auf die Fehlerausgabe.

`print` schreibt auf den Standardausgabe-Kanal
`read` ohne Angabe einer Kanalnummer liest von der Standardeingabe

[4] Unter der Unix-Shell sind alle drei Kanäle mit dem Terminal verbunden, in dem man das Programm startet. Die Umleitungsoperatoren können zur Trennung von `stdout` und `stderr` genutzt werden: `meinprogramm < ein.dat > aus.dat 2> err.dat` liest als Standardeingabe Daten aus `ein.dat`, Ausgaben auf den Standardausgabe-Kanal werden nach `aus.dat` geschrieben und Ausgaben auf den Fehlerausgabe-Kanal in die Datei `err.dat`.

Formatierung von Ein- und Ausgabe

- Die Funktionen `(f)scanf`, `(f)printf` haben als 1.(2.) Argument eine Zeichenkette, die Formatspezifikationen enthält.
- Weitere Argumente von `(f)printf` sind die auszugebenden Variablen, bei `(f)scanf` Zeiger auf die einzulesenden Variablen:

```
double a;  float z[10][10];  int n;

fscanf(fp1,"%lf %i %f", &a, &n, &z[2][3]);
printf("a= %12.5g n= %8i\n", a, n);
printf(" z[4,5] = %g\n", z[4][5]);
```

- Die Funktionen `read`, `write` oder `print` haben als zweites Argument (`print`: 1. Argument) einen Format-String.
- Nach den beiden Argumenten in Klammern folgt eine Liste der zu lesenden/zu schreibenden Variablen.
- Als 2. Argument (`print`: 1. Argument) kann auch ein `*` verwendet werden. Damit werden beim Einlesen die Variablen entsprechend ihres Typs gelesen, zum Ausgeben wird ein Standardformat verwendet. Insbesondere zum Einlesen dürfte das für die meisten Zwecke ausreichen.

```
write(3,*) 'a=', a,' b_10= ', b(10)
read (4, *) x,y
write(3, "(' i= ', i8,/, f12.5, g16.6)") i,x,y
print "(' i= ', i8,/, f12.5, g16.6)" i,x,y
```

Im Folgenden werden nur die einfachsten Formatierungsregeln erwähnt. Für die umfangreichen Details sei auf die Dokumentation der C-Funktionen (`"man 3 fprintf"`, `"man 3 fscanf"`) und z.B. auf Kapitel 10.6–10.11 im *Professional Programmer's Guide to Fortran77* verwiesen.

- Der Formatstring kann Zeichen enthalten, die direkt ausgegeben werden.
- Wichtige Formatangaben sind `%i,%s,%e,%f,%g,%le,%lf,%lg` für ganze Zahlen, Zeichenketten, `float`s mit oder ohne abgetrenntem Exponenten (das `%g`-Format wählt je nach Größe der Zahl das `%e` oder `%f`-Format) und `double`s.
- Die Formate können auch Längenangaben enthalten: `%10f, %10.4f` Die erste Zahl ist die Formatbreite, die zweite die Anzahl der Nachkommastellen.
- Beim Lesen unterdrückt ein `*` die Zuweisung: `scanf("%i %*s %i",&m,&n)` kann Zeilen der Form `"23 blabla 45"` einlesen.

- Formatangaben haben die Form `Iw, Ew.d, Fw.d, Gw.d, A, Aw` oder `Lw`. Dabei ist `w` die Formatbreite, `d` die Anzahl der Nachkommastellen. `I` gibt eine ganze Zahl aus, `F` eine Fließkommazahl, `E` eine Fließkommazahl mit abgetrenntem Exponenten und `G` wählt, je nach Größe der Zahl, `E` oder `F`. Das `A`-Format ist für Zeichen und Zeichenketten, das `L`-Format für boolesche Variablen.
- Ein Bruchstrich `/` erzeugt einen Zeilenvorschub analog zu `\n` in C.

Fehlerbehandlung:

`(f)scanf` meldet die Zahl der gelesenen Variablen zurück. Dies können weniger als erwartet sein, wenn beim Lesen/Konvertieren ein Fehler aufgetreten ist.

```
n = scanf("%i %i %i",&i,&j,&k);
if(n<3) {
  /* Beim Lesen ist ein
    Fehler aufgetreten */
...
}
```

`read` kann als weitere Argumente ein Label haben, zu dem im Falle eines Lesefehlers verzweigt wird oder auch ein Label, zu dem verzweigt wird, wenn das Ende der zu lesenden Datei erreicht ist:

```
   read(*, *, err=33, end=34) i,j,k
   ...
33 write(*,*) 'Lesefehler!'
   ...
34 print *, 'Dateiende erreicht!'
```

13 Sichtbarkeit, Lebensdauer und Initialisierung von Variablen

Konstanten: Variablen können als konstant deklariert werden. Jeder Versuch, diesen Variablen nach der Initialisierung einen anderen Wert zuzuweisen, führt zu einer Compiler-Fehlermeldung.

```
const int N=100;
const double Pi=3.14159;
const char * p;    /* Zeiger zu einem konstanten Zeichen */
char * const p2;   /* konstanter Zeiger */
```

Konstanten tragen das `parameter`-Attribut:

```
integer, parameter :: N = 100
real(kind=8), parameter :: Pi = 3.14159
```

Sichtbarkeit und Lebensdauer: Man unterscheidet nach der Sichtbarkeit (scope) zwischen lokalen und globalen Variablen, nach der Lebensdauer zwischen automatischen und statischen Variablen:

- **Lokale Variablen** sind Variablen, die innerhalb von Funktionen (C: oder Blöcken) definiert werden. Sie sind nur in dieser Funktion/Block sichtbar.
- **Globale Variablen** sind in mehreren Funktionen sichtbar. Damit sind sie neben der Parameterübergabe durch Funktionsargumente eine Methode des Datenaustausches oder gemeinsamen Datenzugriffs verschiedener Funktionen.
- **Automatische Variablen** in einer Funktion behalten ihren Wert **nicht** zwischen zwei Funktionsaufrufen. Ihr Anfangswert bei jedem Funktionsaufruf ist (falls sie nicht explizit initialisiert werden) undefiniert.
- **Statische Variablen** behalten ihren Wert zwischen zwei Funktionsaufrufen.

Sowohl in C als auch in Fortran sind alle Variablen standardmäßig **lokal** und **automatisch**. Variablen können schon bei ihrer Deklaration mit konstanten Ausdrücken initialisiert werden. Konstante Ausdrücke sind im Wesentlichen solche, deren Wert der Compiler unmittelbar ermitteln kann:

```
int f1(int y){
    int x=1;
    double pi = 3.14159;
    int feld1[5] = {15,2,4,4,4};
    int * ip = &x;

    struct Datum heute = { 2004, "Feb", 23 };
    char str[]="Haha";          // Compiler ermittelt Länge selbst

    int a[6] = { [4] = 29, [2] = 15 };  // designated initializers,
               // äquivalent zu  int a[6] = { 0, 0, 15, 0, 29, 0 };
    ...
}
```

Diese Zuweisungen werden bei jedem Aufruf von `f1(y)` neu ausgeführt.

```
integer function f1(y)
   integer :: y, x=1
   real(kind=8) :: pi = 3.14159
   integer, dimension(5) :: feld1 = (/ 15,2,4,4,4 /)
   integer, pointer :: ip => null()
            ! bei Zeigern nur Initialisierung mit 0 erlaubt
   type(Datum) :: heute=Datum(2004, "Feb", 23)
   character(len=*), parameter :: str = "Haha";
            ! Compiler ermittelt Länge selbst, nur bei
            ! Parameter zulässig

   ...
end function f1
```

Durch die Initialisierung wird die entsprechende Variable **statisch**! Die Initialisierung wird nur einmal, beim Programmstart, durchgeführt.

In Fortran müssen in jeder Funktion/Subroutine alle Definitionen vor allen ausführbaren Anweisungen stehen.

In C können Definitionen beliebig mit ausführbaren Anweisungen gemischt werden, was aber nicht unbedingt guter Programmierstil ist. (Natürlich muß trotzdem eine Variable vor ihrer Anwendung definiert sein.) In C kann eine neue Variable auch innerhalb eines {}-Blocks definiert werden und ist dann nur in diesem Block verfügbar. Eine Sonderform ist die Definition in der `for`-Schleifeninitialisierung: `for(int k=0; k<k_end; k++){...}`
Die Variable `k` wird erst am Anfang der `for`-Schleife eingeführt und ist sichtbar bis zum Ende des `for`-Blocks `{...}`.

Statische Variablen:

Eine Variable kann als `static` deklariert werden. Dann bleibt ihr Wert zwischen 2 Funktionsaufrufen erhalten:

```
int f2(int x){
   static int j;
   ...
   j++;
   printf("f2 called %i times",j);
}
```

Statische Variablen werden beim Programmstart mit 0 initialisiert. Wird ein expliziter Initialisierungswert angegeben:
`static double g=3.3;`
so wird auch diese Initialisierung nur einmal beim Programmstart durchgeführt.

Variablen sind statisch, wenn sie
a) das `save`-Attribut tragen

```
      integer, save :: j
```

b) bei der Definition initialisiert werden (s. oben) oder
c) nach der Definition mit einer `data`-Anweisung initialisiert werden.
Auch diese Initialisierung erfolgt einmalig beim Programmstart.
Die `data`-Anweisung erlaubt einige praktische Tricks zur Initialisierung von Feldern:

```
  real, dimension(1000) ::  feld
  data feld/500*1.0, 500*2.0/
```

setzt die ersten 500 Elemente des Felds gleich 1, die anderen gleich 2.

Globale Variablen:

Eine Deklaration/Definition außerhalb von Funktionen (typischerweise am Dateianfang nach `#include`-Anweisungen) ist global. Globale Variablen sind immer statisch.
Eine globale Definition (d.h., eine Definition außerhalb von Funktionsblöcken) kann ebenfalls das Attribut `static` enthalten, welches bei globalen Variablen (die sowieso statisch sind) allerdings eine andere Bedeutung hat: es schränkt die Sichtbarkeit auf die Quelltextdatei ein.

```
  int k=1;        // eventuell auch in anderen
                  // Quelltextdateien verwendbar
  static int m=2; //  nur in dieser Datei sichtbar
```

Um von Funktionen in verschiedenen Quelltextdateien aus auf eine globale Variable zugreifen zu können, muß sie in jeder dieser Dateien deklariert werden: `int k;`
Es empfiehlt sich, solche Deklarationen in eine .h-Datei zu schreiben, die dann mittels `#include` überall eingebunden wird. Genau eine Datei kann dann eine zusätzliche Initialisierung enthalten: `int k=1;`
Wenn in einem Block eine lokale Variable gleichen Namens definiert wird, verdeckt sie die globale Variable. Auf letztere kann dann in diesem Block nicht mehr zugegriffen werden.

Globale Variablen werden durch Module vereinbart:

```
module my_globals
    real, dimension(4) :: ax = (/2, 3, 4, 5/)
    real, dimension(100), save :: bx
    integer :: kl, mn, jn
end module
```

In jeder Funktion oder Subroutine, die auf diese globalen Variablen zugreifen können soll, wird das Modul mittels der `use`-Anweisung verfügbar gemacht:

```
integer function f2(y)
     use my_globals
     ...
```

Seltsamerweise sind `module`-Blöcke lt. Fortran-Standard nicht unbedingt statisch. Der einfachste Weg, sie statisch zu machen, ist, eine `use`-Anweisung mit ins Hauptprogramm (`program ...`) aufzunehmen.

14 Funktionen, Prototypen, Module

14.1 Funktionen mit Rückgabewert

Die Funktion muß mindestens eine Anweisung `return <Ausdruck>` enthalten:

```
int timesec(int, int, int);   /* Prototyp */
...
int main(){
...
   seconds = timesec(hours, mins, secs);
...
}

int timesec(int h, int m, int s){
    int x;
    if(s<0) return -1;
    x = 60*((60*h)+m)+s;
    return x;
}
```

In der Funktion wird einer Variablen gleichen Namens der Rückgabewert zugewiesen. `return` ohne Argument kann zur Ablaufsteuerung verwendet werden.

```
program main2
integer timesec
...
seconds = timesec(hours, mins, secs)
...
end

integer function timesec(h, m, s)
integer h,m,s
if(s.lt.0) then
   timesec=-1
   return
endif
timesec=60*((60*h)+m)+s
end
```

14.2 Funktionen ohne Rückgabewert

```
void timesec(int*, int, int, int);  /* Prototyp */
...
int main(){
...
   timesec(&seconds, hours, mins, secs);
...
}

void timesec(int *x, int h, int m, int s){
    *x = 60*((60*h)+m)+s;
}
```

Funktionen ohne Rückgabewert heißen `subroutine`n. Sie müssen mit `call` aufgerufen werden.

```
program main2
...
call timesec(seconds, hours, mins, secs)
...
end

subroutine timesec(x, h, m, s)
integer h,m,s,x
x=60*((60*h)+m)+s
end
```

14.3 Argumentübergabe

Das letzte Beispiel illustriert einen fundamentalen Unterschied zwischen Fortran und C: In Fortran erfolgt die Parameterübergabe durch ”call by reference”. Wenn eine gerufene Funktion ihre Argumente verändert (wie die Variable x im letzten Beispiel), so ist diese Änderung anschließend auch im rufenden Programm (Variable seconds) wirksam.

In C erfolgt Parameterübergabe prinzipiell durch ”call by value”, die gerufene Funktion bekommt nur eine Kopie des Wertes mitgeteilt. Eine Zuweisung an ein Argument einer Funktion hat daher keinerlei Auswirkung auf die Variablen im rufenden Programm. ”call by reference” muß mit Zeigern simuliert werden: Im C-Beispiel bekommt void timesec(...) einen Zeiger auf die Variable seconds übergeben. Eine Änderung oder Neuzuweisung dieses Zeigers hätte ebenfalls wieder keine Auswirkungen auf das rufende Programm. Aber der Zeiger wird genutzt, um die Speicherzelle zu ändern, auf die er zeigt (*x=...) – und dies ist gerade die Variable seconds des rufenden Programms.

Dieses Verhalten ist auch der Grund dafür, daß die Funktionen scanf(), fscanf(),... Zeiger auf die einzulesenden Variablen als Argumente übermittelt bekommen müssen – sie sollen ja den Wert dieser Variablen setzen.

Die Tatsache, daß C von jedem Funktionsargument erst eine lokale Kopie für die gerufene Funktion herstellt, sollte beachtet werden, wenn man z.B. umfangreiche structs an Funktionen übergeben will. Hier kann die Verwendung von Zeigern wesentlich effizienter sein.

Eine Besonderheit stellen Felder als Argumente dar: hier übergibt C automatisch nur einen Zeiger (und keine Kopie des ganzen Feldes).

14.4 Module und Prototypen

Bei größeren Projekten empfiehlt es sich, den Quellcode auf verschiedene Dateien aufzuteilen, die getrennt compiliert werden können. Dazu braucht man einen Mechanismus, der Funktionen (und globale Variablen) bekannt macht.

In C stellt man Deklarationen globaler Variabler, Definitionen eigener Typen (struct) und die Prototypen von Funktionen in Header-Dateien zusammen (übliche Endung .h). Die Prototypen deklarieren den Typ der Funktion und ihrer Argumente und erlauben damit dem Compiler, zu überprüfen, ob externe Funktionen richtig verwendet werden.
Beispiel: myglobals.h enthält:

```
double feld[1000];  // globales Feld
int solver(int, double[*][*], double[*], int[10]); // Prototyp
```

In myprog.c steht:

```
#include "myglobals.h"
...
int f3(int a, float x) {
   ...
   feld[i] = ...
   ...
}
```

In Fortran kann Funktionen, globale Variablen und Typdeklarationen zu einem Modul zusammenfassen. Module können dann in anderen Programmteilen mittels der use <module_name> Anweisung verwendet werden.
Beispiel: solv.f90 enthält:

```
module solve
   real, dimension(1000) :: feld
contains
   integer function solver(n, a, b, m)
   ...
   end function
end module
```

In myprog.f90 steht:

```
subroutine h2(a, b, c)
    use solve
    ...
    g = solver(j, ka, kb, mm)
    feld(i) = ...
    ...
end subroutine
```

globals.h

```
#include <stdio.h>

int hfunc(int * );
int k;
```

main.c

```
#include "global.h"

int main(){
    int x,y,z;

    scanf("%i", &y);
    k = 3;
    x=hfunc(&y)
    k = 5;
    z=hfunc(&y);
    ...
    return 0;
}
```

hfunc.c

```
#include "global.h"

int hfunc(int * x){
    int y;
    static int i=0;
    i++;
    printf("hfunc %i mal gerufen\n",i);
    y = *x *(*x)+k;
    *x += 3;
    return y;
}
```

gcc -std=c99 -Wall -c hfunc.c
gcc -std=c99 -Wall -o myprog main.c hfunc.o -lm

m1.f90

```
program m1
use m1mod

implicit none
integer :: x, y, z

read *, y
k = 3
x = hfunc(y)
k = 5
z = hfunc(y)

print *, "x= ", x, &
    "y= ", y, "z= ", z
end program
```

m1mod.f90

```
module m1mod
   integer :: k
contains
   integer function hfunc(x)
   integer :: x
   integer, save :: i

   i= i + 1
   print *, "hfunc ", &
       i, "mal gerufen"
   hfunc = x * x + k
   x = x + 3
   end function
end module
```

Kompiliere mit:

```
ifort -warn all  -c m1mod.f90
ifort -warn all -o m1 m1.f90 m1mod.o
```

Aufruf von "m1" und Eingabe von "2" liefert folgende Ausgabe:

```
hfunc            1 mal gerufen
hfunc            2 mal gerufen
x=            7 y=            8 z=           30
```

15 Funktionen als Argumente von Funktionen

Manchmal möchte man einer Funktion den Namen einer anderen Funktion übergeben, die von dieser aufgerufen werden soll. Beispiele sind ein Sortieralgorithmus, dem man neben den zu sortierenden Daten auch eine Vergleichsfunktion übergibt oder eine Routine zur numerischen Integration.

C kennt neben Zeigern auf andere Objekte auch Zeiger auf Funktionen. Diese kann man an Unterprogramme übergeben. Im Unterprogramm kann man diesen Zeiger wie eine Funktion verwenden, man muß ihn nicht mit `*` dereferenzieren. Etwas Übung benötigt die Syntax der Typdeklarationen:

- `fp1` sei Zeiger auf eine Funktion vom Typ int f(int,int):
  `int (*fp1)(int, int);`
- `fun` sei eine Funktion, deren erstes Argument ein solcher Zeiger und das zweite Argument ein `int` ist:
  `int fun( int (*p)(int, int), int);`
- `fp1` wird ein Wert zugewiesen: die Adresse einer Funktion func2 vom entsprechenden Typ:
  `fp1 = &func2;`

Sei als Beispiel `intgr` eine primitive numerische Integrationsroutine:

```
/* Prototypen */
float bsp(float);
float intgr(float (*f)(float), float, float);

int main(){
   ...
   x=intgr(bsp, 0, 1);
   ...
}

float bsp(float x){
    return 2*x*x;
}

float intgr(float (*fp)(float), float x0, float x1){
     ...

     y = fp(x0+ (x1-x0)*i/100.);
     ...
}
```

Funktions- und Subroutine-Namen können einfach als Argumente übergeben werden.
Wenn in einem Programmteil nicht erkennbar ist, daß es sich um Funktionen handelt (weil sie nicht aufgerufen, nur weitergegeben werden), sind sie als `external` zu deklarieren. Ansonsten sollte man (ähnlich den Prototypen in C) die Funktion und ihre Argumente in einem `interface`-Block beschreiben.

```
module trapezregel
    contains
    real function intgr(myfunc, x0, x1)
        implicit none
        real x0, x1
        interface                        ! interface-Block beschreibt
            real function myfunc(x)      ! Typ der Funktion
                real, intent(in) :: x    ! und ihre Argumente
            end function
        end interface
        ...
        do  i=1,100
            intgr = intgr + myfunc( x0 + (x1-x0)*i/100. )
        end do
        ...
    end function
end module

program trapeztest
    use trapezregel
    external bsp
    ...
    x = intgr(bsp, 0., 2.)   ! "bsp" ist der Name einer Funktion
    ...
end

real function bsp(x)
    real x
    bsp = cos(x) + sqrt(x)
end
```

16 Felder als Argumente von Funktionen

Felder fester Größe können als Argumente an Unterprogramme übergeben werden:

```
void eigenvalues(double matrix[3][3], double evs[3])
{ ... }
```

```
subroutine eigenvalues(matrix, evs)
    real(kind=8), dimension(3, 3) :: matrix
    real(kind=8), dimension(3)    :: evs
```

Felder variabler Größe: Besser ist natürlich, wenn ein Unterprogramm (das z.B. Eigenwerte von Matrizen berechnet) mit Matrizen beliebiger Größe umgehen kann:

```
void eigenvalues(int n, double matrix[n][n], double evs[n])
{
     double tmp[2*n];  // Auch lokale Felder können eine Größe
     double ax[n][n];  // haben, die erst zur Laufzeit als
     ...               // Argument übergeben wird
}
```

Das Argument `n` muß vor dem Feldargument `matrix[n][n]` stehen, damit der Typ von `n` bereits deklariert ist.
Für den Prototyp solcher Funktionen gibt es eine spezielle Syntax:
`void eigenvalues(int, double [*][*], double[*]);`

```
subroutine eigenvalues(n, matrix, evs)
    integer :: n
    real(kind=8), dimension(n, n) :: matrix
    real(kind=8), dimension(n)    :: evs
    real(kind=8), dimension(3*n, 2*n+1) :: tmp ! Auch lokale Felder
         ! können eine Größe haben, die erst zur Laufzeit als
         ! Argument übergeben wird
    ...
end subroutine
```

Unterschiede zwischen C und Fortran:

Die oben beschriebenen Felder variabler Größe sind eine relativ neue Erweiterung von C99. Ältere C-Programme verwenden meist Zeiger, um Felder beliebiger Größe an Unterprogramme zu übergeben:
`void eigenvalues(int n, double ** matrix, double * evs){...}`
Die Indexsyntax `a[i]` ist auch für Zeiger anwendbar und wird interpretiert als `*(a+i)`, also: erhöhe die Adresse `a` um `i` und liefere den Wert, der an dieser Adresse steht.
Analog ist für zweidimensionale Felder `double **a` ein Zeiger auf ein Feld von Zeigern. Jeder Zeiger in diesem Feld zeigt auf eine Zeile der Matrix. Der Zugriff auf `a[i][j]` bedeutet nun: Addiere `i` zur Adresse von `a`; lese die Adresse, die an dieser Adresse steht; addiere `j` zu dieser Adressse, liefere den Wert, der dort steht: `a[i][j]` $\equiv$ `*( *(a+i) + j)`
Vom rufenden Programm muß sowohl das Feld für die Zeilenzeiger als auch die Zeilen selbst bereitgestellt und initialisiert werden.

In Fortran muß man die Feldgröße nicht explizit übergeben (assumed-shape arrays). Wenn man (z.B. zur Definition lokaler Felder oder für Do-Schleifen) die Feldgröße benötigt, kann man sie mittels der `size()`-Funktion ermitteln. `size(array, dim)` liefert die Ausdehnung von `array` in Dimension `dim` (also: für eine $n \times m$-Matrix `A` ist `size(A,1)` gleich `n` und `size(A,2)` gleich `m`).

```
subroutine eigenvalues(matrix, evs)
    real(kind=8) :: matrix(:,:)  ! 2-dim. Feld
    real(kind=8) :: evs(:)       ! 1-dim Feld
    real(kind=8), dimension(3*size(matrix,1), size(matrix,2)) :: tmp
    ...
    do i = 1, size(evs,1)
    ...
end subroutine
```

17 Optimierungsmöglichkeiten

- Funktionen können das Attribut `inline` erhalten:

  ```
  inline double hypot(double a, double b){
         return sqrt(a*a+b*b);
  }
  ```

  Der Compiler ist aufgefordert, den Funktionsaufruf wegzuoptimieren, d.h. `x=y*hypot(f1,f2)` durch `x=y*sqrt(f1*f1+f2*f2)` zu ersetzen. `inline`-Funktionen sind nur innerhalb einer Quelltextdatei sichtbar, d.h. wenn man sie universell verwenden will, sind sie gute Kandidaten für die Aufnahme in eine Header-Datei.

- `restrict`-Zeiger.
  Die Verwendung von Zeigern, insbesondere auch im Umgang mit Feldern/Matrizen führt in C zum "Aliasing-Problem": Der Compiler kann in der Regel nicht feststellen, ob zwei verschiedene Zeiger tatsächlich verschiedene Daten referenzieren, oder ob die Zeiger auf dieselben (oder überlappende) Daten zeigen. Damit sind viele Optimierungsstrategien, wie das Halten von Zwischenergebnissen in Registern oder das Umordnen von load/store-Operationen nicht möglich. Wenn nun ein Zeiger das `restrict`-Attribut trägt, wird dem Compiler signalisiert, daß auf den entsprechenden Datenbereich nur und ausschließlich mit diesem Zeiger zugegriffen wird:

  ```
  /* Vektoraddition a1 = a1 + a2 */
  void f1(int n, double * restrict a1, double * restrict a2){
       for(int i=0; i<n; i++) a1[i]+=a2[i];
  }
  ```

  Hier sind Code-Umordnungen, Parallelisierungen u.ä. möglich, die offensichtlich nicht möglich wären (es gäbe falsche Ergebnisse), wenn `a1` und `a2` auf einen überlappenden Speicherbereich weisen.

- `intent`-Attribut bei Funktionsargumenten:
  Die Argumente einer Funktion/Subroutine können das `intent(in)`- oder `intent(out)`-Attribut tragen.

  ```
  subroutine f3(a,b)
  integer, intent(in) :: a    ! f3 wird diesen Wert
                              ! nicht modifizieren
  integer, intent(out) :: b   ! f3 verwendet den Wert
                              ! von b nicht, wird ihn
                              ! aber modifizieren
  b = 3*a + 10
  end subroutine
  ```

- `pure`-Atribut bei Funktionen:
  Normalerweise darf der Compiler die Reihenfolge von Funktionsaufrufen nicht ändern und auch mehrere Aufrufe mit denselben Argumenten nicht durch einen Aufruf ersetzen: Die Funktion kann Nebeneffekte haben (Ein-/Ausgabeoperationen) oder sie kann globale oder statische Variable modifizieren und auf diese Weise bei zwei aufeinanderfolgenden Aufrufen mit denselben Argumenten verschiedene Antworten liefern (z.B. Zufallszahlengeneratoren).
  Wenn all dies nicht der Fall ist (keine Ein-/Ausgabeoperationen, kein Zugriff auf globale Variablen, keine statischen Variablen), kann man die Funktion als `pure` deklarieren und damit dem Compiler weitere Optimierungsmöglichkeiten geben:

  ```
  pure integer function f4(a, b, c)
  ...
  ```

18 Weitere Features von Fortran

18.1 Rekursive Funktionen

Wenn eine Funktion in der Lage sein soll, sich selbst aufzurufen, muß sie das `recursive` Attribut tragen. In Fortran wird normalerweise der Rückgabewert der Funktion einer Variablen mit dem Funktionsnamen zugewiesen. Wenn man eine Funktion schreiben will, die sich selbst aufruft, muß man für den Rückgabewert einen anderen Namen benutzen können. Dieser Name wird in der Funktionsdefinition mit einem `result()`-Attribut angegeben:

```
 recursive function factorial(n) result(ff)
    integer, intent(in) :: n
    integer :: ff                     ! Die Verwendung des result-Attributes zur Angabe einer vom Funktionsnamen
    if(n == 1) then                   ! verschiedenen Variablen für den Rückgabewert ist auch unabhängig
         ff = 1                       ! vom recursive-Attribut möglich
    else
         ff = n * factorial(n-1)
    end if
end function
```

18.2 Operator overloading

Für selbstdefinierte Typen erlaubt Fortran auch eine eigene Definition der arithmetischen Operatoren `+,-,*,/`. Dazu müssen Funktionen definiert werden, die 2 Argumente vom gewünschten Typ haben und ein Resultat dieses Typs zurückgeben. Die Argumente müssen das Attribut `intent(in)` tragen. Mittels einer `interface operator()`-Definition wird die Funktion dem gewünschten Operator zugeordnet.
Analog können auch die Vergleichsoperatoren `==,!=,>,<,<=,>=` definiert werden. Dazu muß die Funktion einen Wert vom Typ `logical` zurückgeben.
Auch der Zuweisungsoperator `=` kann umdefiniert werden. Dazu muß eine `subroutine` geschrieben werden, deren erstes Argument das Attribut `intent(out)` und deren zweites Argument das Attribut `intent(in)` hat. Die Zuordung der Funktion zum `=`-Operator geschieht mittels einer `interface assignment (=)`-Anweisung.

```
module vektorprodukt
    type vec
         real :: x,y,z
    end type

    interface operator (*)
        module procedure vektor_produkt
    end interface

contains
    function vektor_produkt(a, b)
        type(vec)             ::  vektor_produkt
        type(vec), intent(in) :: a, b
        vektor_produkt%x = a%y * b%z - a%z * b%y
        vektor_produkt%y = a%z * b%x - a%x * b%z
        vektor_produkt%z = a%x * b%y - a%y * b%x
    end function
end module

program test
    use vektorprodukt
    type(vec) :: a1,a2,a3

    a1 = vec(2,3,4)
    a2 = vec(7,8,9)

    a3 = a1 * a2     ! hier wird vektor_produkt verwendet!
    print *, a3
end program
```

19 Was fehlt

Die Standard-Bibliotheken beider Sprachen enthalten zahlreiche Funktionen, die nicht erwähnt wurden. Dazu zählen insbesondere Funktionen zur Ein- und Ausgabe, zur Arbeit mit Zeichenketten (strings) und zur Bitmanipulationen.
C: typedef, enum, register, volatile, Funktionen mit variabler Anzahl von Argumenten
Fortran: keyword and optional arguments, internal subprograms, public/private attributes, namelists, generic interfaces, unformated & direct access I/O, allocatable arrays alte F77-Features (common blocks, statement functions, block data, entry)

(Und einiges fehlt auch hier…)

20 Guter Programmierstil

- Die Implementationen eines Algorithmus einerseits und der erforderlichen Ein/Ausgabe- und Steuerroutinen andererseits sollen klar getrennt werden.
- Der eigentliche Algorithmus soll in wiederverwendbarer Form implementiert werden: gekapselt in eine Funktion (mit Unterfunktionen), die alle Parameter und Daten übergeben bekommt und selbst keinerlei Ein-/Ausgabe-Operationen durchführt. Nur so ist garantiert, daß sie auch unter anderen Bedingungen verwendbar ist.
- Fehlerhafte Daten o.ä. sollen weder zu einem Abbruch des Algorithmus noch zu einer Fehlermeldung führen: sie werden in geeigneter Form an das rufende Programm zurückgemeldet. Üblicherweise dient dazu eine ganzzahlige Variable `nerror` o.ä., deren Wert fehlerfreien Ablauf (`nerror=0`) oder einen bestimmten Fehlercode signalisiert.
- Nur in Ausnahmefällen sollen die Ein-/Ausgaberoutinen interaktiv arbeiten. Besser ist es, wenn die benötigten Daten aus einer Datei eingelesen und die Ergebnisse in eine Datei geschrieben werden.
- Es soll standardkonform und betriebssytemunabhängig programmiert werden. Die Notwendigkeit, auf einen anderen Rechner zu wechseln, kann sich unerwartet ergeben.
- Der Quelltext soll auf mehrere Dateien aufgeteilt werden - dies ermöglicht schnelleres Rekompilieren nach Änderungen.
- In C sollen Funktions-Prototypen und globale Deklarationen in Header-Dateien zusammengefaßt werden.
- In Fortran sollen Funktionen, die zusammen eine Algorithmus implementieren, zu Modulen zusammengefaßt werden.

Anhang I: Zahlensysteme

Zahlen werden im Computer durch Bitmuster einer festen Länge N dargestellt.

Ganze Zahlen

Positive ganze Zahlen werden in Binärdarstellung dargestellt:

$$\begin{aligned}
0000\cdots 0000 &= 0\\
0000\cdots 0001 &= 1\\
0000\cdots 0010 &= 2\\
&\cdots\\
0111\cdots 1111 &= 2^{N-1}-1
\end{aligned}$$

Für negative ganze Zahlen verwenden zahlreiche Prozessoren, darunter die Intel-Familie, die Zweierkomplement-Darstellung: $-x$ wird dargestellt, indem das Bitmuster von x invertiert und anschließend 1 addiert wird:

$$\begin{aligned}
1111\cdots 1111 &= -1\\
1111\cdots 1110 &= -2\\
&\cdots\\
1000\cdots 0001 &= -2^{N-1}-1\\
1000\cdots 0000 &= -2^{N-1}
\end{aligned}$$

Damit ist das höchstwertige Bit Vorzeichenbit in dem Sinne, daß seine Anwesenheit negative Zahlen charakterisiert. Sein Setzen oder Löschen entspricht allerdings nicht der Transformation $x \rightarrow -x$.

Fließkommazahlen

erlauben, einen viel größeren Zahlenbereich darzustellen, allerdings mit einer begrenzten Genauigkeit ("Anzahl der gültigen Ziffern"). Die Zahl wird dargestellt durch einen Exponenten E, eine Mantisse M ($0 \leq M < 1$) und ein Vorzeichenbit V:

$$x = (-1)^V \times 2^E \times M$$

Die genauen Regeln für die Darstellung, Rundungsregeln, Fehlerbehandlung etc. in der Fließkommaarithmetik sind im amerikanischen Standard IEEE 754 aus dem Jahr 1985 festgelegt, der als IEC 559 im Jahr 1989 auch internationaler Standard wurde.

Die Intel-32Bit-Architektur kennt drei Formate:

Einfach langes Format: 32 Bit

1	8 Bit	23 Bit
V	E	M

Doppelt langes Format: 64 Bit

1	11 Bit	52 Bit
V	E	M

Erweitertes Format: 80 Bit

1	15 Bit	64 Bit
V	E	M

Zahlen im erweiterten Format werden von GNU C als Typ `long double` unterstützt. Für sie werden 96 Bit Speicherplatz (drei 32Bit-Wörter) reserviert, von denen 16 ungenutzt bleiben.

Intern arbeitet die Intel-CPU anstelle des doppelt langen immer mit dem erweiterten Format. Dies kann z.B. zu leicht unterschiedlichen Verhalten zwischen mit und ohne Optimierung kompilierten Programmversionen führen: Ohne Optimierung wird ein Zwischenresultat eventuell im RAM zwischengespeichert und dabei auf `double`-Format gekürzt, während es in der optimierten Version in einem (80 Bit langen) Prozessorregister gehalten wird und damit genauer ist.

	einfach	doppelt	erweitert
Exponentenlänge E (Bits)	8	11	15
kleinster Exponent e_{min}	-125	-1021	-16382
größter Exponent e_{max}	128	1024	16384
Mantissenlänge	24	53	64
betragsmäßig kleinste normalisierte Zahl größer Null	1.18×10^{-38}	2.23×10^{-308}	3.36×10^{-4932}
größte darstellbare Zahl	3.40×10^{38}	1.79×10^{308}	1.19×10^{4932}
kleinste Zahl ϵ, für die $1 + \epsilon \neq 1$	1.19×10^{-7}	2.22×10^{-16}	1.08×10^{-19}

- Die Größe ϵ begründet die Sprechweise ”Einfach genaue Zahlen haben eine Genauigkeit von etwa 7 Dezimalstellen, doppelt genaue Zahlen etwa 16 gültige Dezimalstellen.”
- Der Exponent wird nicht im Zweierkomplement o.ä. kodiert, sondern als vorzeichenlose Zahl, von der eine feste Zahl (bias) abgezogen wird.
- Die Exponenten überdecken nicht den vollen Wertebereich von E Bits. Das Bitmuster ”111…11” ($e_{max} + 1$) im Exponentenfeld ist reserviert, um die speziellen Werte `NaN` (”Not a Number”, Mantisse ungleich 0) und ±`Unendlich` (Mantisse=0, V=0 oder 1) zu kodieren. Bei einfach- und doppeltgenauen Zahlen ist das Bitmuster ”000…00” ($e_{min} - 1$) reserviert für denormalisierte Zahlen.
- Normalisierte und denormalisierte Zahlen: Die Aufteilung in Exponent und Mantisse ist nicht eindeutig. So kann (der Einfachheit halber im Dezimalsystem, die Mantisse sei dreistellig) 0.1 geschrieben werden als $.100 \times 10^0$, $.010 \times 10^1$ und $.001 \times 10^2$. Die erste Form, bei der die erste Stelle der Mantisse ungleich Null ist, ist die normalisierte Form. Nun ist die Mantisse binär kodiert, also ist die erste Stelle bei normalisierten Zahlen immer 1. Daher kann

man auf die Speicherung der ersten Stelle der Mantisse verzichten (implizites erstes Bit) und dies wird bei einfach- und doppeltgenauen Zahlen auch so gemacht. Dies erklärt die Differenz zwischen den Angaben für die Mantissenlänge in der Tabelle (24 bzw. 53) und der vorhergehenden Angabe über die Zahl der Bits, die zum Speichern der Mantisse zur Verfügung stehen (23 bzw. 52). Im erweiterten Format wird das erste Bit mit kodiert.

- Denormalisierte Zahlen werden nur dann verwendet, wenn sie betragsmäßig so klein sind, daß eine normalisierte Darstellung nicht mehr möglich ist. Die kleinste normalisierte einfach genaue Zahl hat die Mantisse .1000…0 binär ($= 2^{-1} = 0.5$) und den Exponenten $e_{min} = -125$, ist also gleich $2^{-125} \times 2^{-1} = 2^{-126} \approx 1.18 \times 10^{-38}$. Wird diese Zahl nun z.B. durch 8 geteilt, so ist das Ergebnis nur noch denormalisiert mit der Mantisse .00010…0 binär darstellbar. Aufgrund des impliziten ersten Bits muß man denormalisierte Zahlen durch einen speziellen Wert im Exponentenfeld kennzeichnen. (Interpretiert werden sie natürlich mit dem Exponenten $e = e_{min}$.)
- Das Erreichen denormalisierter Zahlen ist natürlich mit einem Genauigkeitsverlust begleitet. Trotzdem hat es sich in der Praxis als vorteilhaft herausgestellt, daß dadurch die "Lücke um die Null" kleiner und ein "gradual underflow" möglich ist. Es gibt auch Computersysteme (z.B. Cray, IBM/390), die keine denormalisierten Zahlen zulassen.

Anhang II: Zeichensätze

ASCII

Der American Standard Code for Information Interchange (ASCII) ist ein Zeichensatz, der nur 7 von den 8 Bits eines Bytes benutzt und demzufolge 128 Zeichen umfaßt.

Die ersten 32 Zeichen 0x00...0x1F sind nicht druckbare Zeichen und enthalten z.B. das Nullbyte 0x00 (in C als `\0` darstellbar und als Endmarkierung von Zeichenketten verwendet), den Zeilentrenner 0x0A (`\n`) und das Tabulatorzeichen 0x09 (`\t`).

0x20 kodiert das Leerzeichen. Auf Position 0x30…0x39 stehen die Ziffern 0…9, auf Position 0x41 bis 0x5A die Großbuchstaben A…Z und auf Position 0x61…0x7A die Kleinbuchstaben.

Weitere druckbare Zeichen (neben 0-9, A-Z, a-z und Leerzeichen) sind `! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ ] \ ^ _ ‘ { } | ~`

ISO 8859

standardisiert eine Reihe weiterer 8-Bit-Zeichensätze durch Auffüllung der freien Hälfte des ASCII-Zeichensatzes.

ISO 8859-1 (Latin1) enthält die Zusatzzeichen der meisten westeuropäischen Sprachen, wie ä (0xE4), é (0xE9), ñ (0xF1) oder æ (0xE6).

ISO 8859-15 (Latin9) ist eine kleine Modifikation von Latin1, bei der u.a. das Eurosymbol € auf Position 0xA4 eingeführt wurde.

Unicode

wird als einheitliche Kodierung aller Schriften der Welt immer wichtiger. Es können bis zu ca. 2 Millionen Zeichen kodiert werden, davon wird jedoch erst ein kleiner Teil, ca. 100 000 Zeichen genutzt.[5]

Unicode umfaßt bereits alle wichtigen Schriftsprachen der Gegenwart (z.B. über 40 000 chinesisch/japanisch/koreanische Schriftzeichen, Arabisch, Hebräisch, Tamil,…) und zahlreiche tote Sprachen und wird zur Zeit noch ständig weiterentwickelt.

Unicode selbst ist keine Kodierung vergleichbar mit ISO 8859, sondern es gibt verschiedene "Unicode transformation formats" (UTF), die einem Unicode-Zeichen eine eindeutige Sequenz von Bytes zuordnen. Die wichtigsten sind UTF-8, UTF-16 und UTF-32. UTF-8 enthält den ASCII-Zeichensatz als Untermenge. Zeichen, die nicht zu ASCII gehören, werden durch 2 bis 4 Bytes kodiert, wobei im ersten Byte a) das höchste Bit gesetzt ist (dadurch als nicht-ASCII erkennbar) und b) kodiert ist, wieviele der folgenden Bytes zu diesem Zeichen gehören.

[5] `http://www.unicode.org`