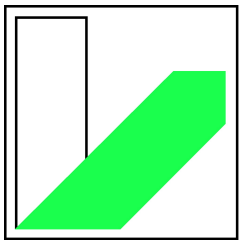

UNIVERSITÄT
BAYREUTH

Programmieren in Fortran 90/95

Skript: Dr. Heidrun Kolinsky, Universität Bayreuth[1]

der Bundeswehr
Universität München

LaTeX-Satz: Marcus Wohler, Universität der Bundeswehr München

[1]Dr. Heidrun Kolinsky, Rechenzentrum der Universität Bayreuth, Universitätsstraße 30, D-95440 Bayreuth, Heidrun.Kolinsky@uni-bayreuth.de

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Die Programmiersprache FORTRAN

FORTRAN ist die Ursprache aller algorithmisch orientierten wissenschaftlichen Programmiersprachen. In Laufe der Zeit wurde FORTRAN mit der Entwicklung der Computertechnik kontinuierlich verbessert und immer wieder standardisiert. Dabei wurden die Sprachelemente kontinuierlich weiterentwickelt, wobei die Rückwärtskompatibilität gewahrt wurde. So ist Fortran 90/95 eine sehr moderne, strukturierte Sprache, die es allerdings noch erlaubt, unsaubereren FORTRAN 77-Code und Stil zu verwenden. Der Aspekt der Rückwärtskompatibilität wurde bei den ANSI-Standardisierungen der Sprache 1966, 1977, 1991 und 1997 bewusst gewählt: durch das Einsatzgebiet der Sprache im technischen- wissenschaftlichen Bereich steckt in jedem Programmpaket ein immenses Know-How, das stets gewahrt und weiter nutzbar bleiben sollte. Fortran 90/95 selbst ist eine sehr moderne Sprache, die dazugehörigen Compiler finden sich auf jedem Hochleistungsrechner und meist auch auf den Workstations.

1.2 Zur historischen Entwicklung von FORTRAN

Der Name **FORTRAN** wurde von **FOR**mula **TRAN**slator abgeleitet und markiert, dass die Sprache entwickelt wurde, um es den damaligen Programmieren zu erleichtern, wissenschafliche Formeln in Computercode umzusetzen. Zur Erinnerung: in der Zeit, als die ersten Wurzeln von FORTRAN bei IBM in den Jahren 1954 bis 1957 entwickelt wurden, musste noch in Maschinensprache (0er und 1er) oder in Assembler (CPU abhängiger Code aus 3 Buchstaben mit expliziter zu programmierender Speicherverwaltung) programmiert werden. Das Programmieren in Maschinensprache und Assembler war, verglichen mit der neu entwickelten Programmiersprache, äußerst umständlich und fehleranfällig. Mit FORTRAN jedoch konnte der Programmierer nun seine Programme in einer der englischen Sprache angepassten Syntax schreiben. Der Programmcode war klar strukturiert, Formeln konnten in nahezu mathematischer Syntax eingegeben werden. Hinzu kam dass sich der wissenschaftliche Anwender um die explizite Speicherverwaltung nicht mehr zu kümmern brauchte: ein riesengroßer Fortschritt! Die Umsetzung der Gleichungen in eine Problemlösung war so viel einfacher geworden und die Programmentwicklung ging im Vergleich zu vorher rasend schnell. Kein Wunder, dass sich der Einsatz von FORTRAN sehr schnell durchsetzte.

Bald wurden für andere Rechnerarchitekturen als den ursprünglichen IBM 704 FORTRAN-Compiler entwickelt.

Anfangs war der Sprachumfang von FORTRAN noch sehr bescheiden. Im Laufe der Zeit wurde er kontinuierlich erweitert und die Sprache verbessert. Um die Kompatibilität zwischen verschiedenen Rechnerarchitekturen zu wahren, wurden die Sprachelemente in FORTRAN 66, FORTRAN 77, Fortran 90 und Fortran 95 vom American National Standards Institute (ANSI) genormt.

Die kontinuierliche Erweiterung und Verbesserung von Fortran findet auch noch heute statt. Die Entwicklungrichtung der Sprachelemente geht in Richtung Parallelisierung, High Performance Computing (HPF - High Performance Fortran) und Distibuted Computing.

Auch hier hier hat sich in den letzten Jahren ein Wikipedia-Eintrag zu Fortran entwickelt, der sowohl die historische Entwicklung als auch den aktuellen Stand der Programmiersprache umfassend und klar diskutiert:

http://en.wikipedia.org/wiki/Fortran.

1.3 Zwei einfache Programmbeispiele in Fortran 90

Das klassische erste Programm „Hello World!“ ist manchen von Ihnen wahrscheinlich bekannt. Im Vergleich zu C schaut „Hello World!“ in Fortran 90 etwas weniger kryptisch und einfacher aus.

```
1 program hello
2 write(*,*) 'Hello World!'
3 end program hello
```

Als Beispiel noch ein einfaches Fortran 90 - Programm, welches eine Umrechnungstabelle von Grad Fahrenheit nach Grad Celsius erzeugt.

```
1  program celsius_table
2
3  implicit none
4  real ::  Fahrenheit, Celsius
5
6  write(*,*) '  Fahrenheit     Celsius'
7  write(*,*) '-------------------------'
8  do Fahrenheit = 30.0, 220.0, 10.0
9          Celsius = (5.0/9.0) * (Fahrenheit-32.0)
10         write(*,*) Fahrenheit,Celsius
11 end do
12 end program celsius_table
```

Während mit den `write(*,*)`-Anweisungen wiederum Text auf die Standardausgabe (Bildschirm) geschrieben wird, sorgt der Abschnitt mit

```
do Fahrenheit = 30.0, 220.0, 10.0
  Celsius = (5.0/9.0) * (Fahrenheit-32.0)
  write(*,*) Fahrenheit,Celsius
```

```
    end do
```

dafür, dass mittels einer Programmschleife der Celsiuswert zu den Fahrenheitwerten `30.0, 40.0, 50.0` usf. bis zum Endwert `220.0` berechnet werden.

Kapitel 2

Die Struktur von Fortran 90/95

Wie wir gesehen haben, versteht man nach DIN 44300 Teil 4 unter einer **Programmiersprache** allgemein eine zum Abfassen von Computerprogrammen geschaffene Sprache. Unter einem **Programm** versteht man wiederum eine von einem Computer interpretierbare vollständige Arbeitsanweisung zur Lösung einer Aufgabe.
Wie bei jeder anderen höheren Programmiersprache gelten auch in Fortran 90 strenge, formale Regeln für Syntax und Semantik.

2.1 Syntax und Semantik - Wiederholung

Die **Syntax** bezeichnet die nach bestimmten Regeln festgelegten Verknüpfungsmöglichkeiten von Zeichen und Befehlen aus dem Zeichen- und Befehlsvorrat einer Programmiersprache.
Die **Semantik** entspricht der Bedeutung, die mit der Abfolge der Sprachelemente in der Programmiersprache verknüpft wird.
kurz:

- Die **Syntax** legt die zugelassenen Zeichen und Zeichenfolgen in einem Programm fest.
- Die **Semantik** legt die Bedeutung der Zeichenfolgen in einem Programm fest.
- Die Semantik legt die Bedeutung der Zeichenfolgen in einem Programm fest.
- Mittels eines Hilfsprogramms (Compiler oder Interpreter) kann das Programm in die für den Computer verständliche Maschinensprache übersetzt und danach ausgeführt werden.
- Die höheren Programmiersprachen sind weitestgehend von der Hardware-Architektur des Rechners unabhängig. (Dies gilt insbesondere für die nach ANSI standardisierten Programmiersprachen, zu denen auch Fortran 90 und Fortran 95 gehören).

- Wenn Sie Ihren Fortran 90/95 Code mit einem Compiler übersetzen, wird ein direkt ausführbares Programm erzeugt. Das ausführbare Programm wird auch *executable* genannt und wurde in der prozessorabhängigen Maschinensprache generiert. Damit sind Executables nur zwischen Rechnern mit gleicher Architektur portierbar. Ihr nach ANSI-Standard erstelltes Fortran-Quelltext-Programm ist jedoch rechnerunabhängig. Sobald auf Ihrem Computer ein Fortran 90-Compiler vorhanden ist, können Sie Ihr Fortran-Programm dort übersetzen (*compilieren*) und ausführen lassen.

2.2 Die äußere Struktur von Programmen

Die „free source form“ von Fortran 90/95
Ein einfaches Fortran-Programm (nur ein Hauptprogramm) besteht im wesentlichen aus 4 Programmteilen:

- Programmkopf
- Vereinbarungs- oder Deklarationsteil
- eigentlicher Anweisungsteil
- Programmende

Später werden zu dem Hauptprogramm noch Unterprogramme hinzukommen, die einen vergleichbaren strukturellen Aufbau aufweisen. Die Unterprogramme stehen in der Regel hinter dem Hauptprogramm oder werden als separate Dateien realisiert.
„free source form“ bedeutet übersetzt soviel wie „freier Quelltext“ und besagt, dass in Fortran 90/95 anders als im Vorgänger FORTRAN 77 das dort notwendige spaltenorientierte Format nicht mehr eingehalten werden muss.
Der Befehlssatz von Fortran 90/95 stellt eine Obermenge der Befehle von FORTRAN 77 dar, so dass mit minimalen Änderungen FORTRAN 77 - Programme vom Fortran 90/95 - Compiler übersetzt werden können.

2.3 Die Struktur einer Fortran 90/95-Anweisung

Programme bestehen aus einer Folge von Anweisungen, die sowohl in sich als auch im Gesamtkontext betrachtet, den zugehörigen Syntax- und Semantik-Regeln der jeweiligen Programmiersprache genügen müssen. Dies gilt selbstverständlich auch für Fortran 90/95 - Programme.
Man unterscheidet:

- nicht-ausführbare Anweisungen (engl. *non-executable statements*)
- ausführbare Anweisungen (engl. *executable statements*)

2.3.1 Nicht ausführbare Anweisungen

Nicht ausführbare Anweisungen liefern Informationen, die für das korrekte Arbeiten des Programms notwendig sind (z.B. Programmkopf-Deklaration, Variablen-Deklaration, Format-Beschreiber).

2.3.2 Ausführbare Anweisungen

Ausführbare Anweisungen beschreiben die Aktionen, die das Programm durchführt, wenn die entsprechende Programmanweisung abgearbeitet wird (z.B. Durchführung einer Wertzuweisung, Addition, Multiplikation etc.).

2.3.3 Wertzuweisungen

Wertzuweisungen in Fortran sind gemäß der folgenden Grundstrukur möglich.

```
<Variablenname> =  <Wert, Variable oder mathematischer Ausdruck>
```

Eine Zeile eines Fortran 90/95-Programms darf bis zu 132 Zeichen lang sein.
Falls eine Anweisung zu lange ist, um in eine Zeile zu passen, kann man die Anweisung in der Folgezeile fortsetzen, indem man die fortzusetzende Zeile mit dem Zeichen `\&` am Zeilenende markiert.
Wenn man möchte, könnte man zusätzlich zu Beginn der Folgezeile ebenfalls ein `\&` setzen (optional).
Natürlich muss man nicht unbedingt 132 Zeichen in einer Zeile vollschreiben, um sich für die Fortsetzung einer Anweisung in der nächsten Zeile zu entscheiden. Oft erhöht ein Zeilenumbruch die Lesbarkeit des Programms und ist aus diesem Grunde sinnvoll. Prinzipiell hat man also Möglichkeiten der Zeilenfortsetzung, wobei das Zeichen `\&` am Ende der fortzuführenden Zeile immer notwendig und das `\&` zu Anfang der fortgesetzten Anweisung optional ist.
Die Zeile

```
output = input1 + input2
```

soll statt in einer Zeile in 2 Zeilen geschrieben werden. Eine Möglichkeit der Zeilenfortsetzung wäre z.B.

```
output = input1 \&
   + input2
```

Als andere Möglichkeit ginge z.B. auch zu Beginn der Fortsetzungszeile das Fortsetzungszeichen zusätzlich zu wiederholen:

```
output = input1 \&
\& + input2
```

Eine Zeilenfortsetzung darf sich nach dem Standard von Fortran90/95 über bis zu 40 Zeilen erstrecken.

Kommentare werden in Fortran 90/95 durch ein `!` (Ausrufezeichen) markiert. Innerhalb einer Zeile wird alles rechts vom Ausrufezeichen ignoriert. Deshalb können in einer Zeile hinter Anweisungen noch mit Ausrufezeichen eingeleitete Kommentare stehen. Fortran 90/95-Programme können auch beliebig viele Leerzeilen aufweisen. Diese werden vom Compiler ignoriert.
Die meisten Anweisungen können mit einer Anweisungsnummer zwischen `1` und `99999` versehen werden. Dabei handelt es sich um eine Markierungen, die für den gezielten Zugriffen auf die Anweisung mit dieser Markierung vorgesehen ist.

```
program hello
90000 write(*,*) 'Hello'
1     write(*,*) 'World !'
end program hello
```

Diese Anweisungsnummern markieren nicht die Reihenfolge, in der die Anweisungen abgearbeitet werden, sondern stellen eine Art „Markierung“ innerhalb des Programmcodes dar (engl. *statement labels*). Dieses Beispielprogramm wird ungeachtet der gewählten Zahlenwerte von oben nach unten abgearbeitet. Und ungeachtet dessen, dass 90000 sehr viel größer als 1 ist, erfolgt die Bildschirmausgabe weiterhin in der Reihenfolge `'Hello'` gefolgt von `'World !'`. Der englische Ausdruck *statement label* macht sehr viel klarer, dass es sich bei diesen Zahlen weniger um eine Nummerierung sondern vielmehr um eine „Markierung“ handelt.
Ihre Programme in Fortran 90/95 sollten stets in der „free source form“ geschrieben werden und nicht mehr nach dem alten FORTRAN 77 - Standard, der nach leichten Modifikationen durchaus noch von einem Fortran 90/95 - Compiler verarbeitet werden kann. Allerdings ist damit zu rechnen, dass bei einer der nächsten Standardisierungen von Fortran die Abwärtskompatibilität zur „fixed source form“ von FORTRAN 77 entfallen wird, da diese bei der Standardisierung von Fortran 95 als *obsolescent* (engl. für „allmählich außer Gebrauch kommend“) bezeichnet wurde. Anders als in Fortran 90/95 gilt bei seinem Vorgänger FORTRAN 77 noch ein aus der Lochkarten-Ära stammendes festes Spaltenformat.

2.3.4 Zwei einfache Beispiele in Fortran 90/95 und in FORTRAN 77

Es sollen zwei ganze Zahlen multipliziert und der Wert des Produkts ausgegeben werden.
Lösung in Fortran 90/95

```
1  program v02_1
2  !
3  ! einfaches Beispiel zur Ein- und Ausgabe
4  !
5  implicit none       ! Schluesselzeile um die implizite
6                                  ! Datentypdeklaration auszuschalten
7  integer :: i, j, k  ! Variablen fuer 3 ganze Zahlen
8
9  ! Text auf der Standardausgabe (Bildschirm) ausgeben
10 write(*,*) 'Berechnung des Produkts zweier ganzer Zahlen'
11 write(*,*)
12 write(*,*) 'Bitte geben Sie zwei ganze Zahlen'
13 write(*,*) '(bitte mindestens durch ein Leerzeichen getrennt) ein:'
14
```

```
read(*,*) i, j        ! Die eingegebenen Zahlen werden eingelesen
                      ! und den Variablen i und j zugewiesen

k = i * j             ! Der Variablen k wird das Produkt aus
                      ! den Werten i und j zugewiesen

                      ! Der Wert, den k jetzt angenommen hat,
                      ! wird ausgegeben

write(*,*) 'Das Produkt der beiden Zahlen ist',  k

! Befehl zur Kennzeichnung des Programmendes
end program v02_1
```

Lösung in FORTRAN 77

```
      PROGRAM V02F77
C
C Das v02_1.f90 Programm in FORTRAN 77
C
C einfaches Beispiel zur Ein- und Ausgabe
C
C Anweisung, um die implizite Datentypdeklaration auszuschalten
      IMPLICIT NONE
C
C Variablen fuer 3 ganze Zahlen
      INTEGER I, J, K
C
C Text ausgeben (Befehle duerfen nur bis einschliesslich Spalte 72 gehen
C                deshalb muss die Zeichenkette umgebrochen werden
C ein * in Spalte 6 markiert in FORTRAN 77 die Fortsetzung der
C darueber liegenden Zeile
C
      WRITE(*,*) 'Berechnung des Produkts zweier ganzer Zahlen'
      WRITE(*,*)
      WRITE(*,*) 'Bitte geben Sie zwei ganze Zahlen'
      WRITE(*,*) '(bitte mindestens durch ein Leerzeichen getrennt)'//
     *           ' ein:'
C
C Die eingegebenen Zahlen werden eingelesen und den Variablen i und j
C zugewiesen
C
      READ(*,*) I, J
C
C Der Variablen K wird das Produkt aus den Werten I und J zugewiesen
C
      K = I * J
C
C Der Wert, den K jetzt angenommen hat, wird ausgegeben
C
      WRITE(*,*) 'Das Produkt der beiden Zahlen ist: ', K
C
C Befehl zur Kennzeichnung des Programmendes
C
      END
```

2.4 Der Fortran 90/95 -Zeichensatz und der Zeichensatz von FORTRAN 77

Wie jede natürliche Sprache hat auch jede Computersprache einen eigenen speziellen Zeichensatz. Nur Zeichen aus diesem Zeichensatz können in Zusammenhang mit der Sprache verwendet werden. Die in rot dargestellten Zeichen waren in FORTRAN 77 noch nicht enthalten.

Anzahl	Art des Symbols	Werte
26	Großbuchstaben	A-Z
26	Kleinbuchstaben	a-z
10	Ziffern	0-9
1	Unterstrich	_
5	Arithmetische Symbole	+ - * / **
17	verschiedene weitere Symbole	( ) . = , ’ $: Leerzeichen (engl. *blank* genannt) ! “ % & ; < > ?

Tabelle 2.1: Der Fortran 90/95 - Zeichensatz

Die Gruppe der Buchstaben zusammen mit den Ziffern werden auch als die alphanumerischen Zeichen bezeichnet.
Obwohl anfangs nur Großbuchstaben im FORTRAN 77 - Zeichensatz vorgesehen waren, wurden im Laufe der Jahre die Compiler so erweitert, dass Kleinbuchstaben als gleichwertiger Ersatz für einen Großbuchstaben akzeptiert wurden.
Wichtig: Ein Fortran-Compiler unterscheidet nicht zwischen Groß- und Kleinschreibung! Würden Sie in Ihrem Programm als Variablennamen wert und Wert verwenden, so würde der Fortran-Compiler anders als ein C-Compiler davon ausgehen, daß es sich um die gleiche Variable handelt.

2.5 Basis-Elemente von Fortran (syntax)

Aus den zulässigen Zeichen sind die Basis-Elemente zusammengesetzt.

- Konstanten bzw. Werte („Direktwertkonstanten“), z.B. `1.0`
- symbolische Namen, z.B. `wert`
- Befehlsworte, z.B. `write`
- Anweisungsnummern (engl. *statement labels*: ganze Zahlen zwischen 1 und 99999), z.B. `20`
- Operatorsymbole, z.B. `+`

2.5.1 Namen in Fortran 90/95 und Namen in FORTRAN 77

Um z.B. Variablen und Konstanten mit einer Bezeichnung versehen zu können, sieht Fortran 90/95 vor, dass diese einen Namen erhalten.
Ein **Name** darf in Fortran 90/95 bis zu 31 Zeichen lang sein. Der Name muss mit einem Buchstaben beginnen. Für die auf das erste Zeichen folgenden Zeichen dürfen alle alphanumerischen Zeichen sowie Unterstriche verwendet werden. Groß- und Kleinschreibung kann eingesetzt werden, bleibt aber vom Compiler unberücksichtigt.
Eine Anmerkung zu FORTRAN 77: Hier sind Namen bis zu einer Länge von 6 Zeichen zulässig. Auch in der Vorgängerversion FORTRAN77 muss der Name mit einem Buchstaben beginnen. An den bis zu 5 folgenden Stellen dürfen alphanumerische Zeichen folgen. Der Unterstrich kann nicht verwendet werden. In FORTRAN 77 können auch Namen verwendet werden, die aus mehr als 6 Zeichen bestehen, jedoch werden vom FORTRAN77-Compiler dem Standard gemäß nur die Namen bis einschließlich des 6. Zeichens voneinander unterschieden. Das heißt: ein FORTRAN 77 - Compiler geht aufgrund der verwendeten Namensregelung davon aus, dass es sich bei RADIUS1 und RADIUS2 um ein und diesselbe Variable RADIUS handeln würde.
Hinweis: Da die Vergabe von Namen unter Fortran 90/95 sehr viel mehr Gestaltungsmöglichkeiten bietet, sollten Sie dieses Angebot nutzen und die von Ihnen vergebenen Namen so wählen, dass sie

- zum einen die Bedeutung des Sachverhalts widerspiegeln, für den sie eingesetzt werden,
- und zum anderen, sich hinreichend voneinander unterscheiden, so dass wenig Verwechslungsgefahr zwischen den einzelnen Benennungen in Ihrem Programmcode besteht.

2.6 Der allgemeine Aufbau eines Fortran 90/95 - Programms

Jedes Fortran - Programm besteht aus einer Mischung ausführbarer und nicht-ausführbarer Anweisungen, die in einer bestimmten Reihenfolge auftreten müssen. Das Programm selbst (und wie wir später sehen werden, auch jede Unterprogrammeinheit) gliedert sich in 3 Teile.

1. Deklarationsteil
2. Anweisungsteil
3. Programmende

Der Deklarationsteil besteht aus einer Gruppe nichtausführbarer Anweisungen am Anfang eines jeden Fortran 90-Programms. Zunächst wird der Name des Programms definiert, dann werden die in den Programmen verwendeten Konstanten und Variablen (mit dem zugehörigen Datentyp) deklariert.
Ein Fortran 90/95 - Programm beginnt mit der nicht-ausführbaren Anweisung

```
program <Programmname>
```

Statt `<Programmname>` ist der von Ihnen gewählte Programmname einzusetzen, der selbstverständlich der Fortran 90/95 - Namenssyntax genügen muss. Zum Deklarationsteil gehören auch die **Konstanten- und Variablenvereinbarungen**.
Eine **Konstante** ist ein Datenobjekt, welches vor der Programmausführung definiert wird. Während der Abarbeitung des Programms bleibt der Wert einer Konstante unverändert. Die Zuweisung eines Wertes zu einer Konstanten gehört deshalb in den Deklarationsteil. Der Fortran-Compiler erzeugt eine Maschinensprachanweisung, durch die bei der Programmabarbeitung des Executables der Wert der Konstante in einen bestimmten Speicherplatz des Hauptspeichers geschrieben wird. Bei der Abarbeitung des Executables wird im folgenden bei einer Konstante der dort abgespeicherte Wert, sobald der Name der Konstante in dem Programm wieder auftritt, aus dem Speicherplatz der Konstanten ausgelesen.
Im Gegensatz zu einer Konstanten ist eine Fortran-**Variable** ein Datenobjekt, dessen Wert sich bei der Ausführung des Programms verändern kann. Wenn ein Fortran-Compiler im Deklarationsteil einer Variablen begegnet, reserviert er an einer bestimmten Stelle im Speicher einen Speicherplatz in der für den angegebenen Datentyp notwendigen Größe als Speicherplatz für die möglichen Werte dieser Variablen. Immer wenn der Compiler auf den Namen dieser Variablen trifft, erzeugt er eine Referenzierung auf diesen Speicherplatz. Bei der Ausführung des compilierten Programms (des Executables) wird je nach vorliegender Anweisung der Inhalt dieses Speicherplatzes entweder gelesen oder es wird dort ein aktualisierter Wert abgelegt.
Jede **Variable** und jede **Konstante** muss innerhalb einer Programmeinheit einen eindeutigen Namen erhalten, z.B. `time, distance, z123456789, ergebnis\_der\_multiplikation`. Die Namen der Variablen müssen der Fortran 90/95 - Namenssyntax-Regel genügen. Nicht zulässig ist es, Leerzeichen im Namen zu schreiben (verwenden Sie statt dessen den Unterstrich) oder den Programmnamen als Variablennamen verwenden zu wollen. Dies würde dazu führen, dass der Compiler beim Übersetzen Ihres Programms mit einer Fehlermeldung abbricht.
Hinweis: Gute Programmierpraxis ist es, sich am Anfang des Programms ein Verzeichnis der im Programm auftretenden Variablen und Konstanten nach Namen, Datentyp und Bedeutung innerhalb des Programms zu erstellen.
Der Anweisungsteil besteht aus einer oder mehreren Anweisungen, die die Aktionen beschreiben, die beim Abarbeiten des Programms ausgeführt werden sollen.
Das Programmende besteht aus einer (oder mehreren) Anweisungen, die die Ausführung des Programms unterbricht und markiert, dass an dieser Stelle der entsprechende Programmteil abgeschlossen ist.

Kapitel 3

Die in Fortran 90/95 intrinsisch enthaltenen Datentypen

Datentyp	Erläuterung	Beispiele von Werten dieses Datentyps
`integer`	ganze Zahlen	`10 -2 1 +1`
`real`	reelle Zahlen	`12.3 1.23 1.23e-1 -1.E-6 -.98`
`complex`	komplexe Zahlen	`(1.0,2.e-3)`
`logical`	logische Werte	`.TRUE. .FALSE.`
`charactar`	Zeichenketten	`'Name' "Zahl" 'Hälfte'`

Tabelle 3.1: Intrinsisch enthaltene Datentypen

Neben diesen intrinsischen Datentypen gibt es noch eine Reihe abgeleiteter Datentypen. Diese werden in einem der folgenden Kapitel behandelt.

3.1 Die Bedeutung von `implicit none`

Historisch bedingt enthält Fortran 90 noch die Möglichkeit zur impliziten Datentypvereinbarung der Vorgängerversionen.
Bei der impliziten Datentypvereinbarung wird vom Compiler anhand des ersten Buchstabens eines Variablen- oder Konstantennamens angenommen, dass der zugehörige Datentyp `real` oder `integer` sei.
Und zwar gilt bei der impliziten Datentypvereinbarung die Regel:
Beginnt der Name mit i, j, k, l, m oder n (oder dementsprechend mit I, J, K, L, M oder N), so wird bei der impliziten Datentypvereinbarung angenommen, dass es sich um eine Variable vom Datentyp integer handelt. Falls ein anderer Anfangsbuchstabe vorliegt, so wird vom Compiler angenommen, dass es sich um eine Variable des Datentyps `real` handelt.
Dringende Empfehlung zur Vermeidung von ungewollten Programmierfehlern: Variablen stets explizit vereinbaren (d.h. jede Variable im Deklarationsteil zusammen mit dem dazugehörigen Datentyp explizit auflisten) und die implizite Typvereinbarungregel von vorneherein ausschalten durch

```
implicit none
```

unmittelbar hinter der `program`-Anweisung.

3.2 Konstanten und Variablen vom Datentyp `integer`

Konstanten und Variablen vom Datentyp integer können Werte aus einen Teilbereich der ganzen Zahlen annehmen. In der Regel stehen für den Datentyp integer 4-Byte Speicherplatz zur Verfügung. Da im Rechner Zahlen im Binär- oder Dual-System codiert werden, stehen somit insgesamt 4 Byte = 4 mal 8 Bit = 32 Bit zur Verfügung. Dies schränkt den Bereich der ganzen Zahlen bei einer 4-Byte-integer-Darstellung ein auf ganze Zahlen im Wertebereich von -2**31+1 bis 2**31-1 = 2147483647. Abhängig von der Hardware-Architektur Ihres Rechners und von dem eingesetzten Compiler können Ihnen auch mehr als 4-Byte (z.B. 8-Byte intrinsisch für den Datentyp `integer` zur Verfügung stehen. Durch die Verwendung eines abgeleiteten Datentyps können Sie den Darstellungsbereich des Datentyps `integer` in der Regel auf das Doppelte (8 Byte) ausdehnen.
Beispiele zum Datentyp `integer`:

```
1  program integer_beispiele
2
3  implicit none
4  integer :: i, j, k, l, m, n
5
6  i = 1
7  j = -30000000
8  k = 123456789
9  l = 2**30              ! 2**31 fuehrt bei einer internen 4 Byte-integer-
10                        ! Darstellung zu compiler error: overflow
11 write(*,*) i, j, k, l
12
13 end program integer_beispiele
```

3.3 Konstanten und Variablen vom Datentyp `real`

Auch hier sind in der Regel 4-Byte Speicherplatz pro Variable/Konstante vorgesehen. Von den 32 Bit stehen normalerweise 24 Bit für die Darstellung der Mantisse und 8 Bit für den Exponenten zur Verfügung. Auch aufgrund der hardwareseitigen Vorgaben ist der darstellbare Bereich der reellen Zahlen eingeschränkt. Die größte darstellbare reelle Zahl wäre in etwa 3.37*10**38, die kleinste reelle positive Zahl 8.43*10**-37. Bei den negativen reellen Zahlen sind die Schranken -3.37*10**-38 und -8.43*10**37. Wird versucht, einer Variablen oder einer Konstanten innerhalb des Programms einen Wert außerhalb dieses Darstellungsbereichs zuzuweisen, erhält man einen „overflow“ oder „underflow“ error.
Zahlenwerte von Variablen und Konstanten des Datentyps `real` kann man in Fortran 90/95 in der Gleitpunkt- oder in der wissenschaftlichen Exponentendarstellung schreiben. Damit haben Zahlen das Format a.b, wobei a für die Zahl vor dem „Komma“ und b für die Zahl nach dem „Komma“ steht.

```
1.2345 0.999 123.12 -450.3273 23.0 23. -0.99 -.99
```

Grundsätzlich schreibt man in Fortran statt dem im Deutschen verwendeten Komma, in Anlehnung an die englische Zahlendarstellung auch in Fortran einen Punkt um Vor- und Nachkommastellen zu trennen. Ein negatives Vorzeichen muss der Zahl direkt vorangestellt werden. Mit einem positiven Vorzeichen kann man dies tun oder es einfach weglassen. Reine Nullen als Nachkommastellen können ganz weggelassen werden. Eine führende Null als einzige Vorkommastelle kann ebenfalls weggelassen werden.
Die von der Wissenschaft und Technik bekannte Darstellung einer Zahl mit Vor- und Nachkommastellen zusammen mit einem Exponenten zur Basis 10 wird auch in Fortran verwendet. So stellt z.B. mancher Taschenrechner eine Milliarde als 1.E+9. Der Ausdruck 'E+9' steht für den Exponenten zur Basis 10 und besagt hier nichts anderes, als dass die 1 noch mit 10**9 = 1000000000 zu multiplizieren wäre. Das Zahlenformat mit dem Exponenten zur Basis 10 aus der wissenschaftlichen Darstellung ist natärlich auch in Fortran „eingebaut“. Reelle Zahlen innerhalb des Darstellungsbereichs lassen sich schreiben als a.bEc oder a.bE-c oder -a.bEc oder a.bE-c. Wahlweise ließe sich wieder ein Pluszeichen zusätzlich aufführen und/oder führende Nullen oder folgende Nullen in den Nachkommastellen vernachlässigen.

```
12.3 1.23 1.23e-1 -1.E-6 -.98  +1.23e-1 5.e+1
```

Beispiele zum Datentyp `real`:

```
1  program real_beispiele
2
3  implicit none
4  real :: w, x, y, z
5
6  w = 5.0e-1
7  x = 1.e-9
8  y = 123456789.012
9
10 z = x * y
11
12 ! Das listengesteuerte Format fuer die Ausgabe [write(*,*)] gibt
13 ! real-Werte immer so aus, dass die Stellen der
14 ! der real-Zahl gut dargestellt werden, die im Genauigkeitsbereich
15 ! (der intrinsische real-Datentyp ist bei einer internen
16 ! 4-Byte-Darstellung auf ca. 7 Stellen genau) liegen
17
18 write(*,*) 'w␣=␣', w
19 write(*,*)
20 write(*,*) 'x␣=␣', x
21 write(*,*) 'y␣=␣', y
22 write(*,*) 'z␣=␣x␣*␣y␣=␣', z
23
24 end program real_beispiele
```

3.4 Konstanten und Variablen vom Datentyp `charactar`

Der `character`-Datentyp besteht aus einer Kette (Aneinanderreihung) von Zeichen, die in ' (einfache Anführungszeichen) oder " (doppelte Anführungszeichen) eingeschlossen sind. In

FORTRAN 77 sind nur die einfachen Anführungszeichen bei der Definition von Zeichenketten (engl. *strings*) als Begrenzer möglich. Zeichen, die innerhalb einer Zeichenkette stehen, werden in diesem Kontext betrachtet, z.B. wird ein in der Zeichenkette eingeschlossenes Ausrufzeichen ! nicht als Einleitungszeichen eines Kommentars behandelt. Innerhalb der Zeichenkette dürfen auch Zeichen ausserhalb des Fortran-Zeichensatzes verwendet werden, z.B. deutsche Umlaute etc.
Die minimale Länge einer Zeichenkette ist 1 und die maximale Länge der Zeichenkette ist compilerabhängig. Beispiele für zulässige Zeichenketten:

```
'Radius = '
"Radius = "
'This is a test!'
' '
'{ }'
'Klöße und Bier'
'Peter''s Backstube'
"Peter's Backstube"
'Er wunderte sich: "Was ist denn das?"'
```

Falsche, nicht zulässige Zeichenketten wären beispielsweise `This is a test!` (die Begrenzer der Zeichenkette wurden vergessen), `'This is a test!"` (inhomogene Begrenzer der Zeichenkette). Beispiele zum Datentyp `character`:

```
1  program character_beispiele
2  implicit none
3
4  write(*,*) 'Peter''s Backstube' ! Ein eingeschlossenes ' muss
5                                  ! wiederholt werden
6                                  ! (Fortran 77 und 90/95)
7  write(*,*) "Peter's Backstube"  ! uebersichtlichere Alternative
8                                  ! (nur in Fortran 90/95)
9
10 end program
```

Deklaration von Zeichenketten
Beispiel 1:

```
character :: buchstabe
```

Hier wurde eine Zeichenkette mit der Länge 1 deklariert.
Beispiel 2:

```
character(15) :: Zeichenkette_mit_15_Zeichen
```

Damit wird eine Zeichenkette mit 15 Zeichen deklariert.
Beispiel 3 (die ausführliche Deklaration einer Zeichenkette):

```
character(len=15) :: Zeichenkette_mit_15_Zeichen
```

Bei der ausführlichen Version steht explizit `len` als Abkürzung des englischen Wortes *length* für die Länge der Zeichenkette.
Vorgriff: Bei Zeichenketten ist es unbedingt empfehlenswert, unter Angabe des Formatbeschreibers `A` in der `read`-Anweisung einzulesen, weil so gewährleistet werden kann, dass auch ohne Angabe der Zeichenkettenbegrenzer `'` bzw. `"` Zeichenketten mit eingebetteten Leerzeichen (z.B. eine Zeichenkette aus 2 Worten) vollständig gelesen werden.
Nachdem z.B. eine Zeichenkette der Länge 40 deklariert wurde

```
character(len=40) :: zeichenkette
```

kann mit

```
read(*,'(A)') zeichenkette
```

in die Zeichenkette z.B. `Guten Morgen` ohne Angabe der Zeichenkettenbegrenzer `' "` eingelesen werden. Würde listengesteuert eingelesen werden

```
read(*,*) zeichenkette
```

müsste zum Einlesen der vollständigen Zeichenkette der Anwender `'Guten Morgen'` bzw. `"Guten Morgen"` eingeben. Denn bei der Eingabe von `Guten Morgen` ohne die Zeichenkettenbegrenzer würde nur `Guten` als Einlesewert für die Zeichenkette übernommen werden.
Beispielprogramm zum Selbst-Ausprobieren:

```
1  program character_einlesen
2  implicit none
3
4  character(len=40) zeichenkette
5
6  write(*,*) 'Geben␣Sie␣bitte␣eine␣Zeichenkette␣ein!'
7  ! read(*,*) zeichenkette
8  read(*,'(A)') zeichenkette      !formatiertes Einlesen von Zeichenketten
9  write(*,*)
10 write(*,*) 'Eingelesen␣wurde:␣', zeichenkette
11
12 end program character_einlesen
```

3.5 Deklaration benannter Konstanten

Hinweis: Am besten oberhalb des Blocks mit den Variablendeklarationen hinter der Zeile mit der `implicit none`-Anweisung durchführen!
Benannte Konstanten werden durch die Angabe des Datentyps, der um das `parameter`-Attribut ergänzt wurde, gefolgt von `::` und der Angabe des Namens gefolgt von einer Wertzuweisung deklariert. Beispiele:

```
real,parameter :: pi = 3.141593
character(len=13), parameter :: fehlerstring = 'unknown error'
```

Die Länge des Strings braucht bei der Deklaration einer benannten Zeichenketten-Konstante nicht unbedingt berechnet zu werden. Dies kann auch der Compiler übernehmen. Nur ist dann an Stelle der Längenangabe ein `*` einzufügen

```
character(*), parameter :: fehlerstring = 'unknown error'
```

Hinweis: Die Deklaration benannter Konstanten sollte in einem Programm vor der Deklaration der Variablen erfolgen. Bereits vereinbarte Konstanten können bei Bedarf bei der Variablendeklaration wieder eingesetzt werden. Zum Beispiel kann man die Länge einer Zeichenkette als Konstante vereinbaren und diese dann zur Längenangabe einer Zeichenkette verwenden.

```
integer, parameter :: zeilenlaenge = 80
character(len=zeilenlaenge) :: erste_zeile, zweite_zeile, dritte_zeile
```

3.6 Wertzuweisungen und arithmetische Berechnungen

Eine Wertzuweisung hat die Form

```
< Variablenname > = < Ausdruck >
```

Bei einer Wertzuweisung wird zunächst der Wert auf der rechten Seite des =-Zeichens berechnet und dann der Variablen auf der linken Seite zugewiesen. Damit ist auch klar, dass bei einer Wertzuweisung auf der linken Seite nur der Name einer Variablen stehen kann - (und weder der Name einer Konstanten noch ein Ausdruck).
Bei dem =-Zeichen handelt es sich im mathematischen Sinne keineswegs um ein Gleichheitszeichen, sondern um einen **Zuweisungsoperator**. Um z.B. den Wert einer `integer`-Variablen um 1 zu erhöhen, schreibt man

```
i = i + 1
```

was im mathematischen Sinne Unsinn wäre, im Kontext von Fortran jedoch die richtige Form einer Wertzuweisung darstellt.
Auf der rechten Seite einer Wertzuweisung darf jede gültige Kombination von Konstanten, Variablen, Werten, Klammern, arithmetischen und logischen Operatoren stehen.

3.6.1 Die arithmetischen Operatoren

Zeichen	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
**	Exponentiation

Tabelle 3.2: Arithmetische Operatoren in Fortran

Diese Operatoren kommen normalerweise als binäre Operatoren vor und verknüpfen

```
< Variable1 > < binärer Operator > < Variable1 >
```

z.B. `a + b`, `a - b`, `a * b`, `a / b`, `a ** b`. + und - können auch als unitäre Operatoren auftreten, z.B. `+a` oder `-b`. Es gelten die im Folgenden aufgeführte Regeln.

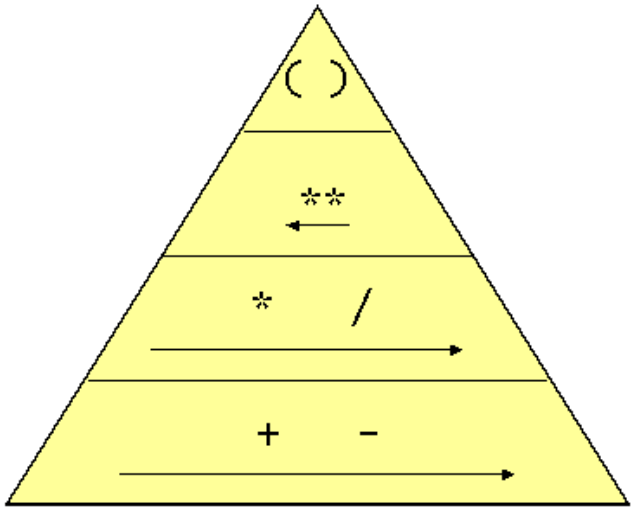

Abbildung 3.1: Hierarchie der Operatoren mit Auswerterichtung gleichrangiger arithmetischer Ausdrücke

3.6.2 Regeln für die arithmetischen Operatoren

1. Zwei Operatoren dürfen nicht unmittelbar aufeinander folgen. Unzulässig ist z.B. `a*-b`, richtig ist: `a*(-b)`

2. Der Multiplikationsoperator darf anders als in der Mathematik nicht weggelassen werden. Unzulässig ist z.B. `a(x+y)`, richtig ist: `a*(x+y)`

3. Klammern werden in der aus der Mathematik bekannten Reihenfolge ausgewertet (von innen nach außen). Z.B. `2**((7*2)/5) = 2**(14/5) = 2**(2) = 2**2 = 4`

3.6.3 Hierarchie der Operatoren

1. Klammern in mathematischer Reihenfolge (s.o.)

2. Exponentiation, z.B. ist die Formel für den freien Fall durchaus richtig implementiert als
   `Ort = 0.5 * Erdbeschleunigung * Zeit ** 2`

3. Bei gemischten Ausdrücken mit Mulitplikation bzw. Division und Addition bzw. Subtraktion gilt die Regel „Punkt“ vor „Strich “

4. Arithmetische Ausdrücke auf gleicher Hierarchiestufe (z.B. nur Multiplikationen und Divisionen im arith. Ausdruck) werden von „links nach rechts“ ausgewertet
   `b / c * e / f = ( ( b / c ) *e ) / f`
   Für Ausdrücke nur mit Additionen und Subtraktionen gilt ebenso die Auswerteregel „von links nach rechts“ .
   Ausnahme: Auf der reinen Hierarchiestufe der Exponentiationen wird von rechts nach links ausgewertet.

3.7 Integer-Arithmetik

Werden in einem arithmetischen Ausdruck zwei Werte von Datentyp `integer` mit einem Operator miteinander verknüpft, so ist das Resultat ebenfalls von Datentyp `integer`.
Achtung: Bei der Division ganzer Zahlen wird dabei nur der ganzzahlige Anteil als Ergebnis verwendet. So ergibt `3/4` als Resultat `0`, `4/4` liefert `1` und `7/4` hat als Ergebnis ebenfalls `1`.
Beispielprogramm:

```
1  program integer_arithmetik
2
3  implicit none
4
5  write(*,*) "␣3/4␣=", 3/4
6  write(*,*) "␣4/4␣=", 4/4
7  write(*,*) "␣5/4␣=", 5/4
8  write(*,*) "␣6/4␣=", 6/4
9  write(*,*) "␣7/4␣=", 7/4
10 write(*,*) "␣8/4␣=", 8/4
11 write(*,*) "␣9/4␣=", 9/4
12
13 end program integer_arithmetik
```

Es gelten die im folgenden Kapitel erläuterten Regeln.

3.8 Regeln für die interne Datentyp-Konversion

Werden zwei Variablen, Konstanten oder Werte mit gleichen oder unterschiedlichen Datentypen mit einem der Operatoren `+`, `-`, `*` oder `/` verknüpft, dann ergibt sich der Datentyp des resultierenden Ausdrucks aus der folgenden Tabelle. `<op> = +, -, *, /`

a<op>b	`integer :: b`	`real :: b`
`integer :: a`	`integer`	`real`
`real :: a`	`real`	`real`

Tabelle 3.3: Regeln der internen Datentyp-Konversion

Beispiele:

```
1 + 1 / 4 = 1
1. + 1 / 4 = 1.000000
1 + 1. / 4 = 1.250000
```

Beispielprogramm:

```
1  program conv_demo
2
3  implicit none
4  integer :: ergebnis
5
6  ergebnis = 1.25 + 9 / 4
7  write(*,*) 'ergebnis␣(Datentyp␣integer)␣=␣1.25␣+␣␣9␣/␣4␣=␣', ergebnis
8  write(*,*) 'Zum␣Vergleich␣:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣1.25␣+␣␣9␣/␣4␣=␣', 1.25 + 9 / 4
```

```
9
10 end program conv_demo
```

Bei der Wertzuweisung

```
ergebnis = 1.25 + 9 / 4
```

wird aufgrund der Operator-Hierarchie („Punkt vor Strich“) zunächst die Division `9/4` (zweier Zahlen von Datentyp `integer`) ausgeführt. Wegen der Integer-Arithmetik-Regeln hat dieser Ausdruck den Wert `2`. Im nächsten Schritt wird die `real`-Zahl `1.25` zu der `2` hinzugezählt. Dabei wird vorher die `(integer) 2` aufgrund der Datentyp-Konversions-Regeln in den Datentyp `real` des ersten Summanden `(real) 1.25` gewandelt. Das Ergebnis der Addition ist von Datentyp `real` und hat den Wert `3.25`. Sie stellt den Wert dar, die der `integer`-Variablen `ergebnis` zugewiesen werden soll. Die Variable `ergebnis` hat den Datentyp `integer`. Deshalb wird `(real) 3.25` nochmals in die `integer`-Zahl `3` gewandelt, bevor die Wertzuweisung durchgeführt und die `3` am Speicherplatz der `integer`-Variablen `ergebnis` abgespeichert wird.
Merke: Beim Programmieren sollte man darauf achten, dass man Zahlen ohne Nachkommastellen, die in Ausdrücken mit dem Datentyp `real` verwendet werden, stets im `real`-Format als Zahl mit dem `.` oder noch besser wegen der besseren Übersichtlichkeit als Zahl gefolgt von `.0` eingibt (z.B. `2.` oder `2.0` statt `2`), um die in der `integer`-Arithmetik versteckten Gefahren zu umgehen.

3.8.1 Explizite Datentyp-Umwandlungen

Will man einen `integer`-Wert explizit in einen Wert vom Datentyp `real` umwandeln oder umgekehrt, so kann man dies tun indem man den Wert oder die Variable in `int( )` bzw. `real( )` einschließt.

Ausgangs-Datentypen	Umwandlungsfunktion	neuer Datentyp
`integer`	`real( )`	`real`
`real`	`int( )`	`integer`

Tabelle 3.4: Explizite Datentyp-Umwandlungen

3.8.2 Exponentiationen

Keine Regel ohne Ausnahme. Bei einem ganzzahligen (`integer`) Exponenten sollte man auch bei einer Basis vom Datentyp `real` den Exponenten als `integer` d.h. als ganze Zahl schreiben. Eine Wertzuweisung der Form

```
p = x**n
```

mit `x` und `p` vom Datentyp `real` und `n` vom Datentyp `integer` (`n > 0`) wird bei der Auswertung die Exponentiation in eine Multiplikation verwandelt, d.h. `x` n-mal mit sich selbst multipliziert. Dies geht zum einen schneller, zum anderen erhält man exaktere Resultate. Hier findet auch keine implizite Datentypkonversion statt, weil stets mit dem Datentyp `real` gearbeitet wird.
Eine Wertzuweisung der Form

```
p = x**y
```

wobei sowohl `x` und `p` als auch `y` vom Datentyp `real` sind, wird intern anders ausgewertet. Dies wäre der Fall, wenn z.B. `y` den Wert `2.5` hätte. Der arithmetische Ausdruck auf der rechten Seite wird ausgewertet, indem die Umrechnungsformel für den Logarithmus verwendet wird. Statt `x**y` direkt zu berechnen, wird dieser Ausdruck in `exp(y*log(x))` umgewandelt und dieser Ausdruck ausgewertet. `exp` lautet der Name der in Fortran intrinsisch enthaltene Berechnungsroutine für die Exponentialfunktion, `log` ist der Funktionsname für den natürlichen Logarithmus, den man in der Mathematik gemeinhin als *ln* schreibt.
Da der Logarithmus nur für positive Zahlen definiert ist, können in Fortran somit auch keine nichtganzzahligen (`reellen`) Exponenten für negative Basen verwendet werden.
Beispielprogramm:

```
1  program exp_error
2
3  implicit none
4  real    :: x, y
5
6  x = -8.0
7  y = 1.0/3.0
8  write(*,*)
9  write(*,*) 'x = ', x, '; y = ', y
10 write(*,*) 'x**y = ', x**y
11
12 end program exp_error
```

Bei der Programmausführung erhält man einen so genannten *runtime error*.
Merkregel: Zu einer negativen Basis sollten in einem FORTRAN-Programm keine reellen Exponenten stehen. Sie müssten hier problemangepasste Lösungen implementieren, die im speziellen Fall diese Problematik umgehen. Allerdings gibt es hier wiederum Situationen, in denen spezielle Compiler diese ungünstige Situation durch Wegoptimierung umgehen, falls zu einer negativen Basis der reelle Exponent einer ganzen Zahl entspricht (z.B. `(-8.0)**2.0`)
Probieren Sie hierzu, wie sich Ihr Compiler bei folgendem Beispielprogramm verhält.

```
1  program exponentiation
2
3  implicit none
4  integer :: n
5  real    :: x, y
6
7  n = 2
8  y = 2.0
9  x = 8.0
10
11 write(*,*) 'x = ', x, '; n = ', n, '; y = ', y
12 write(*,*) 'x**n = ', x**n
13 write(*,*) 'x**y = ', x**y
14
15 n = 2
16 y = 2.0
17 x = -8.0
18
19 write(*,*)
20 write(*,*) 'x = ', x, '; n = ', n, '; y = ', y
```

```
21 write(*,*) 'x**n = ', x**n
22 write(*,*) 'x**y = ', x**y
23
24 write(*,*)
25 write(*,*) 'Jetzt wird eine 1.0/3.0 als Exponent verwendet'
26
27 x = 8.0
28 y = 1.0/3.0
29 write(*,*)
30 write(*,*) 'x = ', x, '; y = ', y
31 write(*,*) 'x**y = ', x**y
32
33 x = -8.0
34 y = 1.0/3.0
35 write(*,*)
36 write(*,*) 'x = ', x, '; y = ', y
37 write(*,*) 'x**y = ', x**y
38
39 end program exponentiation
```

3.9 Wertzuweisungen und Ausdrücke vom Datentyp `logical`

Eine „logische“ Wertzuweisung hat die Form

```
< logische Variable > = < logischer Ausdruck >
```

Unter einem logischen Ausdruck versteht man in diesem Zusammenhang einen Ausdruck, der zum Datentyp `logical` gehört. Der logische Ausdruck kann sich aus jeder zulässigen Kombination logischer Konstante, logischer Variabler oder logischer Werte zusammensetzen. Ein **logischer Operator** ist ein Operator, der auf numerische, `character` oder logische Daten angewandt wird und als Ergebnis einen logischen Wert hat.
Es gibt 2 Arten logischer Operatoren:

1. Vergleichsoperatoren
2. Verknüpfungsoperatoren

3.9.1 Vergleichsoperatoren

Diese Operatoren stehen zwischen zwei arithmetischen oder Zeichenausdrücken oder Werten und ergeben als Resultat einen logischen Ausdruck. Der Wert des resultierenden Ausdrucks ist entweder wahr (`.TRUE.`) oder falsch (`.FALSE.`). Beispiele:

3.9.2 Hierarchie der logischen Operatoren

Der logische Ausdruck

```
7 + 3  < 2 + 11
```

wird aufgrund der in Fortran 90/95 implementierten Operatorhierarchie in der Form

Fortran 90/95	FORTRAN 77	Bedeutung
==	.EQ.	gleich
/=	.NE.	ungleich
>	.GT.	größer als
>=	.GE.	größer oder gleich
<	.LT.	kleiner als
<=	.LE.	kleiner oder gleich

logischer Vergleich	Ergebnis
3 < 4	.TRUE.
3 <= 4	.TRUE.
3 == 4	.FALSE.
3 > 4	.FALSE.
4 <= 4	.TRUE.
3 /= 4	.TRUE.
'A' < 'B'	.TRUE.

Tabelle 3.5: Vergleichsoperatoren

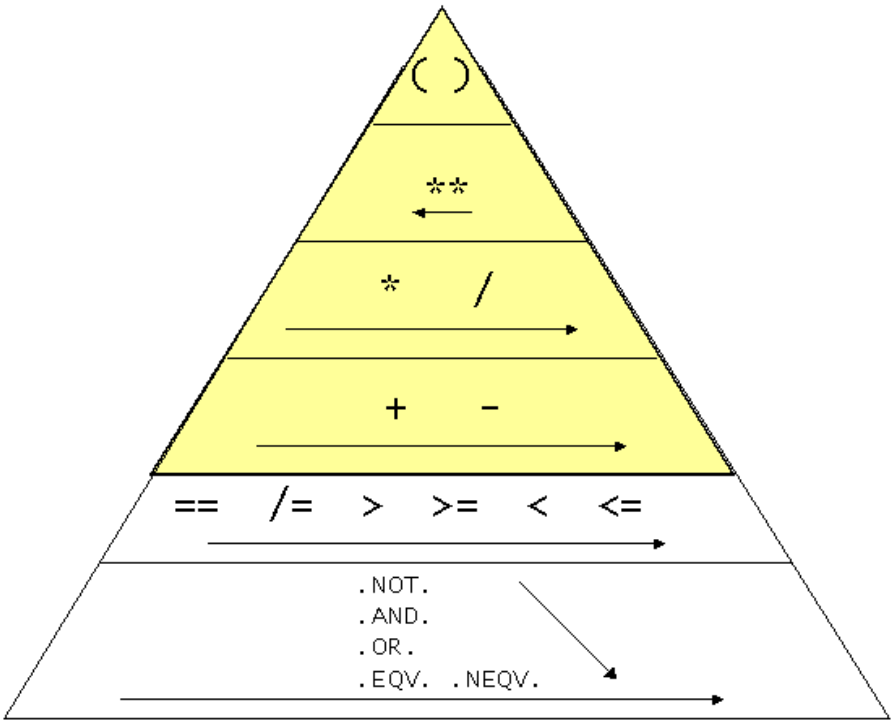

Abbildung 3.2: Hierarchie der logischen Operatoren

```
( 7 + 3 )  < ( 2 + 11 )
```

ausgewertet. Der logische Vergleich

```
4 == 4.
```

ist möglich und ergibt `.TRUE.`. Hier wird die `integer`-Zahl in einen Wert vom Datentyp `real` umgewandelt und dann wird verglichen. Der Versuch

```
4 <= 'A'
```

führt jedoch bei den meisten Compilern zu einem Abbruch des Übersetzungs-Vorgangs mit einem *compiler error*.

3.9.3 Logische Verknüpfungsoperatoren

Logische Ausdrücke lassen sich mit den logischen Verknüpfungsoperatoren miteinander verbinden. Die binären Operatoren zeigt die Tabelle 3.6 auf Seite 29. Die Tabelle 3.7 zeigt Ergebnisse von logischen Verknüpfungen. Des weiteren gibt es noch einen unitären logischen Operator (`.NOT.`), der logischen Ausdrücken, Variablen oder Werten vorangestellt werden kann. Er hat die Bedeutung der Negation.

Verknüpfungsoperator	Bedeutung
`.AND.`	logisches Und
`.OR.`	logisches Oder
`.EQV.`	wahr, wenn beide Ausdrücke denselben logischen Wert haben
`.NEQV.`	wahr, wenn beide Ausdrücke einen unterschiedlichen logischen Wert haben

Tabelle 3.6: Logische Verknüpfungsoperatoren

l1	l2	l1 `.AND.` l2	l1 `.OR.` l2	l1 `.EQV.` l2	l1 `.NEQV.` l2
`.FALSE.`	`.FALSE.`	`.FALSE.`	`.FALSE.`	`.TRUE.`	`.FALSE.`
`.FALSE.`	`.TRUE.`	`.FALSE.`	`.TRUE.`	`.FALSE.`	`.TRUE.`
`.TRUE.`	`.FALSE.`	`.FALSE.`	`.TRUE.`	`.FALSE.`	`.TRUE.`
`.TRUE.`	`.TRUE.`	`.TRUE.`	`.TRUE.`	`.TRUE.`	`.FALSE.`

Tabelle 3.7: Wertetabelle für die logischen Verknüpfungsoperatoren

3.9.4 Auswerteregeln für Operatoren

Gemischte Ausdrücke werden nach den folgenden Regeln ausgewertet.

1. Die arithmetischen Operatoren werden nach den bereits bekannten hierarchischen Regeln (wie oben dargestellt) ausgewertet.
2. Dann folgen die Ausdrücke mit den Vergleichsoperatoren (`==`, `/=`, `>`, `>=`, `<`, `<=`) von links nach rechts.
3. Daraufhin folgt in der Auswertehierarchie der Negationsoperator `.NOT.` (von links nach rechts), gefolgt von `.AND.`, danach von `.OR.` (von links nach rechts) und von `.EQV.` und `.NEQV.` auf gleicher Hierarchiestufe (auch hier von links nach rechts).

Die Regeln verdeutlicht außerdem die Abbildung 3.2 auf Seite 28.

3.10 Zeichenketten-Verarbeitung

Zeichenketten (engl. *strings*) entstehen durch die Aneinanderreihung mehrerer Zeichen einer Zeichen des Datentyps `character`. Zum Beispiel kann eine Zeichenkette mit 10 Zeichen Länge durch

```
character(len=10) :: ausdruck
```

deklariert werden (Fortran 90/95). Zulässig wäre auch, dies in der Form

```
character(10) :: ausdruck
```

zu tun (Fortran 90/95 und Fortran 77). Veraltet und nicht mehr verwendet werden sollte die (alte) Fortran 77 - Version

```
character*10 ausdruck
```

Variablen vom Datentyp einer Zeichenkette lassen sich genauso wie Variablen der anderen Datentypen durch eine Wertzuweisung mit Werten belegen, z.B.

```
ausdruck = 'diesunddas'
```

Nochmals zur Wiederholung: character|-Daten müssen in einfache (Fortran 77 und Fortran 90/95) oder doppelte Anführungszeichen (Fortran 90/95) eingeschlossen werden. Die Länge einer Zeichenkette lässt sich mit

```
len(ausdruck)
```

bestimmen. Durch die Angabe zweier Indexgrenzen lassen sich Unterzeichenketten innerhalb einer Zeichenkette auswählen, z.B.

```
ausdruck(5:7)
```

ergibt und als Unterzeichenkette. Will man das erste Auftreten eines bestimmten Zeichens finden, gibt es hierzu

```
index(<Name eines Ausdrucks>,character-Zeichen)
```

Diese Funktion kann man zusammen mit der Möglichkeit, Unterzeichenketten durch die Angabe zweier Indexgrenzen zu extrahieren, kann man auch nutzen, um z.B. in einer Zeichenkette einzelne Worte zu separieren (siehe Beispielprogramm).
Achtung: Wenn Sie Zeichenketten, die Leerstellen enthalten können, einlesen wollen, sollten Sie statt listengesteuert (`read(*,*)`) formatiert einlesen, z.B.

```
read(*,'(A)') <Variablenname einer Zeichenkettenvariable>
```

Will man bei einer Zeichenkette evtl. sich am Ende befindliche Nullstellen nicht mit ausgegeben haben, so kam man

```
trim(ausdruck)
```

einsetzen. Das folgende Beispiel illustriert das unterschiedliche Verhalten der beiden Einleseformate (formatiert, unformatiert) und einiger der Befehle zur Zeichenkettenverarbeitung. Beispielprogramm:

```
program strings

implicit none
character(len=20), parameter :: messlatte = "....5....|....5....|"
character(len=20)            :: zeichenkette1, zeichenkette2

write(*,*)
write(*,*) 'Bitte␣geben␣Sie␣einen␣max.␣20␣Zeichen␣langen␣Text␣ein!'
write(*,*) 'listengesteuertes␣Einlesen␣mit␣read(*,*)'
read(*,*)  zeichenkette1
write(*,*)
write(*,*) 'Ausgabe␣der␣Zeichenkette␣nach␣Einlesen␣mit␣read(*,*)'
write(*,*)  zeichenkette1
write(*,*)  messlatte
write(*,*)
write(*,*) 'Bitte␣geben␣Sie␣erneut␣einen␣max.␣20␣Zeichen␣langen␣Text␣ein!'
write(*,*) 'formatiertes␣Einlesen␣mit␣read(*,''(A)'')'
read(*,'(A)') zeichenkette2
write(*,*)
write(*,*) "Ausgabe␣der␣Zeichenkette␣nach␣Einlesen␣mit␣read(*,'(A)')"
write(*,*)  zeichenkette2
write(*,*)  messlatte

write(*,*)
write(*,*) 'weitere␣Verarbeitung␣von␣Zeichenketten:'
write(*,*)
write(*,*) 'len(zeichenkette1):␣', len(zeichenkette1)
write(*,*) 'len(zeichenkette2):␣', len(zeichenkette2)
write(*,*) 'len_trim(zeichenkette1):␣', len_trim(zeichenkette1)
write(*,*) 'len_trim(zeichenkette2):␣', len_trim(zeichenkette2)

write(*,*)
write(*,*) 'das␣1.␣bis␣einschliesslich␣das␣7.␣Zeichen␣ausgeben␣lassen'
write(*,*) 'zeichenkette1(1:7):␣', zeichenkette1(1:7)
write(*,*) 'zeichenkette2(1:7):␣', zeichenkette2(1:7)
write(*,*)
write(*,*) 'die␣Stelle␣anzeigen␣lassen,␣an␣der␣das␣1.␣Leerzeichen␣auftritt'
write(*,*) "index(zeichenkette1,'␣'):␣", index(zeichenkette1,'␣')
write(*,*) "index(zeichenkette2,'␣'):␣", index(zeichenkette2,'␣')

end program strings
```

Das Programm liefert folgende Bildschirmausgabe:

```
Bitte geben Sie einen max. 20 Zeichen langen Text ein!
listengesteuertes Einlesen mit read(*,*)
123 56789

Ausgabe der Zeichenkette nach Einlesen mit read(*,*)
123
....5....|....5....|


Bitte geben Sie erneut einen max. 20 Zeichen langen Text ein!
```

```
formatiertes Einlesen mit read(*,'(A)')
123 56789

Ausgabe der Zeichenkette nach Einlesen mit read(*,'(A)')
123 56789
....5....|....5....|

weitere Verarbeitung von Zeichenketten:

len(zeichenkette1): 20
len(zeichenkette2): 20
len_trim(zeichenkette1): 3
len_trim(zeichenkette2): 9

das 1. bis einschliesslich das 7. Zeichen ausgeben lassen
zeichenkette1(1:7): 123
zeichenkette2(1:7): 123 567

die Stelle anzeigen lassen, an der das 1. Leerzeichen auftritt
index(zeichenkette1,' '): 4
index(zeichenkette2,' '): 4
```

Muss man einzelne Zeichenketten miteinander verknüpfen, gibt es hierzu den **Verkettungsoperator** //.
Durch eine geschickte Kombination von Verkettungsoperator und

```
iachar( )   ! gibt die Nummer eines Zeichens in der ASCII-Tabelle an
```

und

```
achar( ) ! gibt zu der Nummer eines Zeichens in der ASCII-Tabelle ! das
zugehoerige character-Zeichen zurueck
```

lassen sich einfache sukzessive numerierte Dateinamen schaffen.
Beispielprogramm:

```
1  program zeichenketten
2
3  implicit none
4  character(len=10) :: ausdruck1
5  character(len=4)  :: ausdruck2 = '.erg'
6  character(len=10) :: ausdruck3
7  integer :: i
8
9  ausdruck1 = 'diesunddas'
10 ausdruck3 = 'dies das'
11 write(*,*)  'ausdruck1          = ', ausdruck1
12 write(*,*)  'ausdruck2          = ', ausdruck2
13 write(*,*)  'ausdruck3          = ', ausdruck3
14 write(*,*)
```

```
15 write(*,*)  'len(ausdruck1) ======= ', len(ausdruck1)
16 write(*,*)  'len(ausdruck2) ======= ', len(ausdruck2)
17 write(*,*)  'len(ausdruck3) ======= ', len(ausdruck3)
18 write(*,*)
19 write(*,*)  'ausdruck1(5:7) ======= ', ausdruck1(5:7)
20 write(*,*)  'ausdruck2(1:1) ======= ', ausdruck2(1:1)
21 write(*,*)
22 write(*,*)  'ausdruck1//ausdruck2 = ', ausdruck1//ausdruck2
23 write(*,*)
24 write(*,*)  'Die Stelle (den Index) des 1. Leerzeichens in ausdruck3 finden:'
25 write(*,*)  "index(ausdruck3,' ') = ", index(ausdruck3,' ')
26 write(*,*)
27 write(*,*)  'Die Nummer eines Zeichens in der ASCII-Tabelle finden:'
28 write(*,*)  "iachar('a') ========== = ", iachar('a')
29 write(*,*)
30 write(*,*)  'Zu der Nummer in die ASCII-Tabelle das korresp. Zeichen finden:'
31 write(*,*)  'achar(65) ============ = ', achar(65)
32 write(*,*)
33 write(*,*)  "In der ASCII-Tabelle, das auf das 'A' folgende Zeichen finden:"
34 write(*,*)  "achar(iachar('A')+1) = ", achar(iachar('A')+1)
35 write(*,*)
36 write(*,*)  "von ausdruck3 das erste Wort ausgeben:"
37 i =  index(ausdruck3,' ')
38 write(*,*) ausdruck3(1:(i-1))
39 write(*,*)
40 write(*,*) 'Sukzessive Dateinamen zusammensetzen: '
41 do i= 1, 3
42         write(*,*) 'datei'//achar(iachar('0')+i)//'.dat'
43 end do
44
45 end program zeichenketten
```

Das Programm liefert folgende Bildschirmausgabe:

```
ausdruck1 = diesunddas
ausdruck2 = .erg
ausdruck3 = dies das

len(ausdruck1) = 10
len(ausdruck2) = 4
len(ausdruck3) = 10

ausdruck1(5:7) = und
ausdruck2(1:1) = .

ausdruck1//ausdruck2 = diesunddas.erg

Die Stelle (den Index) des 1. Leerzeichens in ausdruck3 finden:
index(ausdruck3,' ') = 4

Die Nummer eines Zeichens in der ASCII-Tabelle finden:
iachar('a') = 97
```

```
Zu der Nummer in die ASCII-Tabelle das korresp. Zeichen finden:
achar(65) = A

In der ASCII-Tabelle, das auf das 'A' folgende Zeichen finden:
achar(iachar('A')+1) = B

von ausdruck3 das erste Wort ausgeben:
dies

Sukzessive Dateinamen zusammensetzen:
datei1.dat
datei2.dat
datei3.dat
```

Noch ein weiteres Beispiel, wie Dateinamen zusammengesetzt werden können:

```
program dateiname

implicit none
character(len=40) :: filename
character(len=44) :: neuer_filename
character(len=3)  :: zahlenstring
integer :: zahl

write(*,*) 'Demoprogramm: zusammengesetzte Dateinamen'
write(*,*) 'Geben Sie den gewuenschten Dateinamen ein (max. 40 Zeichen):'
read(*,'(A)') filename
write(*,*) 'Geben Sie eine positive ganze Zahl (max. 3 Zeichen)'
write(*,*) 'die als Endung des Dateinamens verwendet werden soll:'
read(*,*)  zahl

write(zahlenstring,'(I3)') zahl
! internal write (Umwandung integer -> character)
! hier ist die Formatangabe zwingend notwendig
! zahlenstring wird allerdings von rechts aufgefuellt
! so dass weitere Tricks angewendet werden,
! damit spaeter in neuer_filename keine Leerzeichen auftreten

if ( zahl < 0 .or. zahl > 999 ) then
        stop 'Eingabefehler'
else if ( zahl > 0 .and. zahl < 10 ) then
        neuer_filename = trim(filename)//'.00'//adjustl(zahlenstring)
  ! trim( )schneidet Leerzeichen am Ende der Zeichenkette ab
  ! adjustl( ) fuehrende Leerzeichen in einer Zeichenkette werden
  !            nach hinten gesetzt
else if (zahl >= 10 .and. zahl < 100) then
  neuer_filename = filename(1:(len_trim(filename)))&
  &//'.0'//adjustl(zahlenstring)
  ! len_trim gibt die Stelle des 1. Leerzeichens an
  ! alternative, aufwendigere Realisierung mit gleicher Funktion wie trim( )
else
```

```
36    neuer_filename = trim(filename)//'.'//adjustl(zahlenstring)
37  end if
38
39  write(*,*)
40  write(*,*) 'Der zusammengesetzte Dateiname lautet:'
41  write(*,*)  neuer_filename
42
43  end program dateiname
```

Das Programm liefert folgende Bildschirmausgabe:

```
Demoprogramm: zusammengesetzte Dateinamen
Geben Sie den gewuenschten Dateinamen ein (max. 40 Zeichen):
ausgabedatei
Geben Sie eine positive ganze Zahl (max. 3 Zeichen)
die als Endung des Dateinamens verwendet werden soll:
95

Der zusammengesetzte Dateiname lautet:
ausgabedatei.095
```

Kapitel 4

Intrinsisch in Fortran enthaltene Funktionen

In Fortran stehen eine Reihe mathematischer Funktionen, die bei der Auswertung von mathematischen, numerischen, physikalischen und technischen Zusammenhängen benötigt werden, generisch zur Verfügung.
Wichtige Funktionen sind in Tabelle 4.1 aufgelistet.
Als Argumente der obigen Funktionen lassen sich

- Werte
- Konstante
- Variablen
- arithmetische Ausdrücken aus den obigen Objekten einschließlich weiterer Funktionen

einsetzen.
Beispielprogramm zu `sin(x)`:

```
1  program sinus
2
3  ! was man vermeiden sollte:
4  ! real-Schleifenvariablen koennen aufgrund von Rundungsfehlern
5  ! aufgrund der begrenzten Zahlendarstellungsgenauigkeit zu
6  ! Abweichungen in den Durchlaufzahlen fuehren
7  ! (system- und compilerabhaengig; testen z.B. mit a=0.0, b=1.0, s = 0.001)
8  ! besser => Schleifenindizes vom Datentyp integer
9
10 implicit none
11 real :: a, b, s, x
12 integer :: n = 0
13
14 write(*,*) 'Tabellierung von sin(x) im Intervall [a,b] mit Schrittweite s'
15 write(*,'(A$)') 'Bitte geben die untere Intervallgrenze a ein: '
16 read(*,*) a
17 write(*,'(A$)') 'Bitte geben die obere Intervallgrenze b ein:  '
18 read(*,*) b
19 write(*,'(A$)') 'Bitte geben die Schrittweite s ein:           '
```

Funktionsnamen in Fortran	**mathematische Bedeutung**
`exp(x)`	Exponentialfunktion von `x`
`log(x)`	der natürliche Logarithmus von `x` (math: *ln(x)*)
`log10(x)`	der dekadische Logarithmus von `x` (math: *log(x)*)
`sqrt(x)`	die Quadratwurzel aus `x`
`abs(x)`	der Betrag von `x`
`sin(x)`	der Sinus von `x` (math: *sin(x)*) Achtung: x in 'Radiant' angeben
`cos(x)`	der Cosinus von `x` (math: *cos(x)*) Achtung: x in 'Radiant' angeben
`tan(x)`	der Tangens von `x` (math: *tan(x)*) Achtung: x in 'Radiant' angeben
`asin(x)`	*arcsin(x)*, die Umkehrfunktion von *sin(x)*
`acos(x)`	*arccos(x)*, die Umkehrfunktion von *cos(x)*
`atan(x)`	*arctan(x)*, die Umkehrfunktion von *tan(x)*

Tabelle 4.1: Intrinsisch enthaltene Funktionen

Funktionsnamen in Fortran	**mathematische Bedeutung**
`int(x)`	Umwandlung der `real`-Zahl `x` in den Datentyp `integer` durch Abschneiden der Nachkommastellen
`nint(x)`	Umwandlung der `real`-Zahl `x` in den Datentyp `integer` durch Runden
`real(i)`	Umwandlung der `integer`-Zahl `i` in den Datentyp real|
`achar(i)`	gibt das `character`-Zeichen zurück, welches an der `i`-ten Stelle in der ASCII-Tabelle steht
`iachar(c)`	gibt die Stelle aus, an der das `character`-Zeichen `c` in der ASCII-Tabelle steht

Tabelle 4.2: Funktionen zur Datentyp-Konversion

Funktionsnamen in Fortran	**mathematische Bedeutung**
`min(a,b)`	gibt als Funktionswert die kleinere Zahl von `a` oder `b` zurück
`max(a,b)`	gibt als Funktionswert die größere Zahl von `a` oder `b` zurück

Tabelle 4.3: Minimum und Maximum zweier Zahlen

```
read(*,*) s

write(*,*) '␣␣␣␣x␣␣␣␣␣␣␣␣␣sin(x)␣'
write(*,*) '---------------------'

do x = a, b, s
        write(*,'(F12.6,3X,F6.4)') x, sin(x)
        n = n + 1
end do

write(*,*) 'Anzahl␣der␣Schleifendurchlaeufe:␣␣', n
write(*,*) 'berechnete␣Zahl␣an␣Durchlaeufen:'
write(*,*) 'Theorie:␣max(int(␣(b-a+s)/s␣),␣0):', max(int( (b-a+s)/s ), 0)

end program sinus
```

Zur expliziten Datentyp-Konversion lassen sich u.a. die Befehle aus Tabelle 4.2 verwenden.
Beispielprogramm zu `achar` und `iachar`:

```
program ascii_tabelle

implicit none
integer :: istart, n

istart = iachar('A')
do n = istart, istart+60
        write(*,*) n,'␣',achar(n)
end do
end program ascii_tabelle
```

Auch Funktionen zur Bestimmung des Minimums und des Maximums zweier Zahlen sind in Fortran eingebaut (siehe Tabelle 4.3).
Beispielprogramm zu `min` und `max`:

```
program min_max

implicit none
real :: a, b

write(*,*) 'Das␣Programm␣berechnet␣das␣Miniumum␣zweier␣&
␣␣␣␣␣␣␣␣␣␣␣&␣reeller␣Zahlen␣a␣und␣b'
write(*,'(A$)') 'Bitte␣geben␣Sie␣a␣und␣b␣durch␣eine␣&
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣&␣Leerstelle␣getrennt␣ein:␣'
read(*,*)  a, b
write(*,*)
write(*,*) 'Das␣Minimum␣beider␣Zahlen␣ist:␣', min(a,b)
write(*,*) 'Das␣Maximum␣beider␣Zahlen␣ist:␣', max(a,b)

end program min_max
```

Hinweis: Bei den Funktionen `min()` und `max()` können auch mehr als 2 Argumente angegeben werden, so dass z.B. auch leicht mit `max()` das Maximum dreier und mehr Zahlen bestimmt werden kann.

Kapitel 5

Entwurfsstrategien für Programme

Nur einfachste Programme zeigen einen sequentiellen oder linearen Ablauf (z.B. ein Programm, um einen DM-Betrag einzulesen und diesen in Euro umzurechnen). Programme zur Lösung komplexerer Probleme weisen fast immer Verzweigungen und/oder Schleifen auf, so dass nur einzelne Teile des Programms ausgeführt werden, wenn bestimmte Bedingungen erfüllt sind (Verzweigungen) oder einzelne Anweisungsblöcke werden wiederholt ausgeführt (Schleifen).
Bei der Entwicklung komplexerer Programme hat sich unter Aspekten der Aufwandsminimierung und der Fehlerreduzierung als Methode das Top-Down-Entwurfs-Prinzip bewährt.

5.1 Top-Down-Entwurfs-Prinzip

Bei den algorithmisch orientierten Programmiersprachen kommt der Entwicklung des Lösungsverfahrens eine zentrale Bedeutung zu. Der erste Schritt hierzu besteht darin, zuerst die zugrunde liegende Problemstellung zu verstehen. Danach wird festgelegt, welche Eingaben das Programm benötigt und was das Programm ausgeben soll. Größere Probleme werden in kleine Teilschritte zerlegt, die wiederum in weitere Teilaufgaben zergliedert werden können. Zu jeden Teilschritt entwickelt der Programmierer den dazugehörigen Lösungsalgorithmus. Zur Entwicklung der Lösungsalgorithmen ist es fast immer sinnvoll, zunächst für einen einfachen Datensatz die Lösung per Hand/Taschenrechner/Computeralgebrasystem herzuleiten und die daraus gewonnenen Erkenntnisse danach weiter zu verallgemeinern. Als Hilfsmittel zur Entwicklung der Algorithmen und des Programmablaufs können Sie einen sogenannten Programmablaufplan (engl. *flow chart*) zeichnen oder den Algorithmus zunächt in Pseudocode (eine Mischung aus Fortran und Englisch) aufschreiben.
Erst nachdem im Detail der Lösungsweg auf dem Papier ausgearbeitet wurde, beginnt die Umsetzung in Programmcode. Durch die intensive gedankliche Vorarbeit - so zeigt die Erfahrung vieler Programmentwickler - werden bereits im Vorfeld die wichtigen Details geklärt - und es entstehen übersichtlichere, struktuiertere und weniger fehleranfällige Programme als bei einem überhasteten, spontanem Programmieren. Der methodische Ansatz ist in der Regel auch schneller, das sehr viele Fehler, die bei spontanem Vorgehen erst gefunden und eliminiert werden müssen aufgrund der intensiven gedanklichen Vorarbeit gar nicht erst auftreten. Wurde der Programmcode geschrieben, werden am besten die einzelnen Teilschritte separat getestet und dann die Teilschritte zu dem Gesamtprogramm zusam-

mengesetzt. Es ist absolut notwendig, das Gesamtprogramm mit allen möglichen Arten von zugelassenen Eingabedaten zu testen und sicherzustellen, dass sämtliche Verzweigungen und Schleifen richtig abgearbeitet werden.
Deshalb sollte auf die Auswahl der Testdatensätze größte Sorgfalt verwendet werden. Die richtige Lösung für die Testdaten sollte in einem unabhängigen Verfahren (je nach Problemstellung per Hand, Taschenrecher, Computeralgebrasystem, Lösungen aus anderen (numerischen) Verfahren, von anderen Wissenschaftlern aus Publikationen, abgeleitete Werte aus einer reinen theoretischen Beschreibung) hergeleitet und mit dem Ergebnis des entwickelten Programms verglichen werden.Erst wenn Sie absolut sicher sind, dass Ihr Programm in allen Fällen richtig arbeitet, sollten Sie es einsetzen bzw. freigeben.
Nochmals die Grundprinzipien des Top-Down-Entwurfs in der Übersicht:

1. Fertigen Sie eine eindeutige und klare Problembeschreibung an.
2. Beschreiben Sie genau, welche Informationen eingegeben und welche ausgegeben werden sollen.
3. Arbeiten Sie für einen einfachen Datensatz die Lösung per Hand/Taschenrechner/Computer-Algebra-System aus.
4. Strukturieren Sie nun für das generalisierte Problem die einzelnen Schritte. Teilen Sie dazu das Gesamtproblem in immer feinere Teilschritte auf, die sukzessive weiter verfeinert werden können. („vom Groben zum Feinen = Top-Down-Entwurf“) (Bei diesem Schritt können Sie Datenfluß-Diagramm oder Pseudocode zu Hilfe nehmen).
5. Setzen Sie erst jetzt den von Ihnen entwickelten Lösungsalgorithmus in Programm-Code um. Realisieren Sie evtl. einzelne, in sich abgeschlossene Programmteile als Unterprogramme.
6. Durchlaufen Sie den oberen Fehlersuch- und Eliminierungszyklus so oft wie nötig.
7. Speziell bei der Auswahl Ihrer Testdaten sollten Sie sich Gedanken machen, ob die verwendeten Testdatensätze soweit wie möglich voneinander unabhängig sind und die Spezifität des Problems hinreichend verkörpern.

5.2 Gängige Symbole für Programmablaufpläne (engl. *flow charts*)

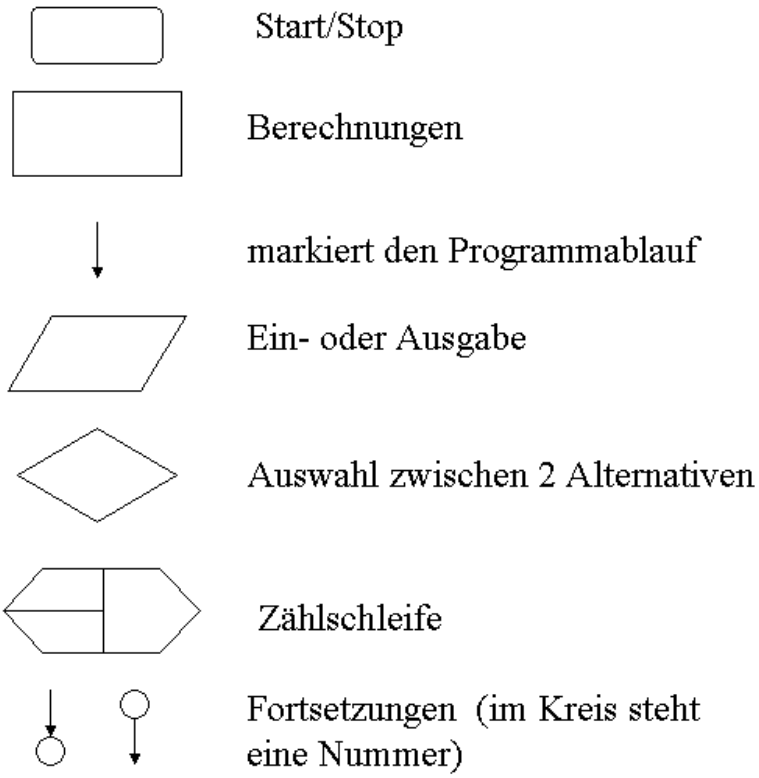

Abbildung 5.1: Gängige Symbole für Programmablaufpläne

Kapitel 6

Kontrollstrukturen

6.1 Verzweigungen

Unter Verzweigungen versteht man in Fortran Anweisungen, die es erlauben, bestimmte Teile eines Programms auszuführen, während andere übersprungen werden.

6.1.1 Die Block-`if`-Konstruktion (`if then - end if`-Konstruktion)

Die Struktur einer einfachen Verzweigung ist die folgende

```
if ( < logischer Ausdruck > ) then

    Anweisung 1
    ...
    Anweisung n

end if
```

Falls der logischer Ausdruck wahr ist (`< logischer Ausdruck > == .TRUE.`), wird der zwischen `if` und `end if` eingeschlossene Anweisungsblock ausgeführt. Falls der logische Ausdruck den Wert `.FALSE.` haben sollte, so wird die nach dem `end if` folgende Anweisung ausgeführt. In einem Programmablaufplan (oft auch Flußdiagramm oder engl. *flow chart* genannt) veranschaulicht Abbildung 6.1 den zugrunde liegenden Sachverhalt. Ein konkretes Beispiel:
Zu der quadratischen Gleichung

```
a*x**2 + b*x + c = 0, a ungleich 0
```

sollen die reellen Nullstellen berechnet werden. Je nach Wert der Diskriminante `D = b**2 - 4*a*c` exisitieren zwei (`D > 0`), eine doppelte (`D = 0`) oder keine reellen Nullstellen (`D < 0`). Als Programmcode stellt sich das wie folgt dar (den Programmablaufplan zeigt Abbildung 6.2):

```
if ( b**2 - 4.0*a*c < 0.0) then

    write(*,*) 'Keine reellen Nullstellen'

end if
```

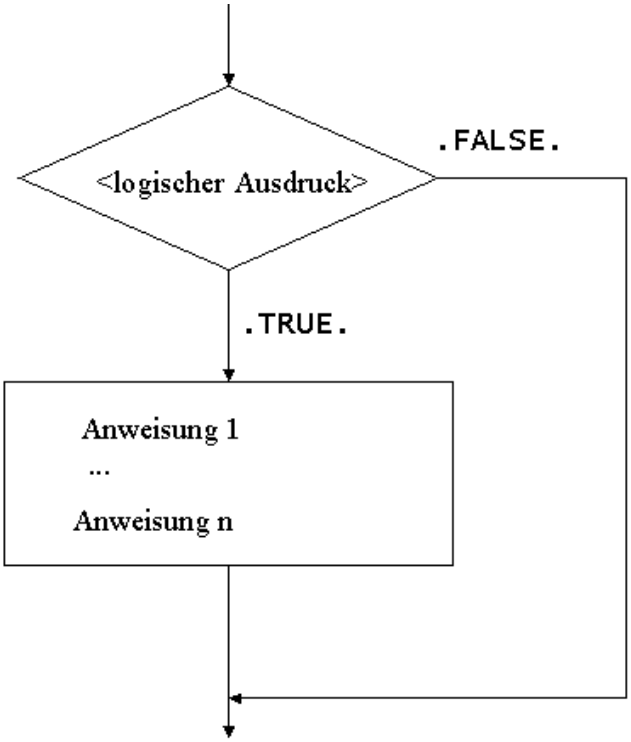

Abbildung 6.1: Block-`if`-Konstruktion

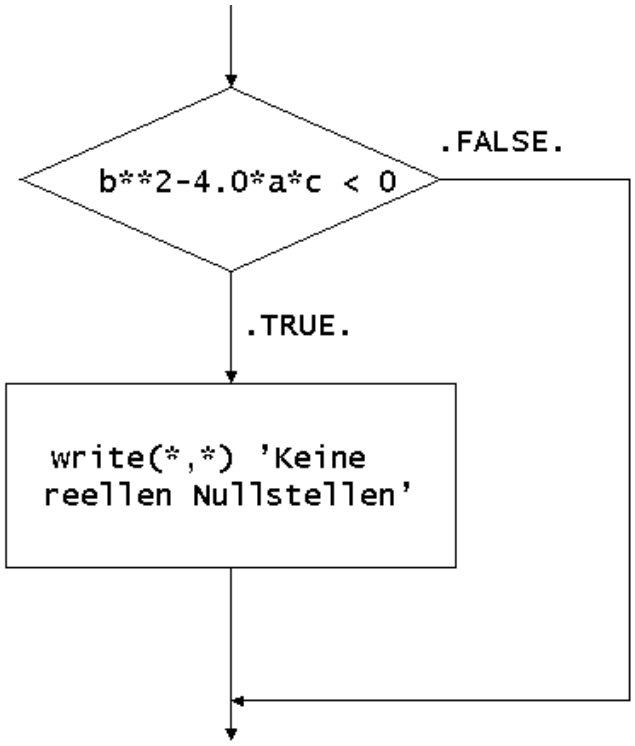

Abbildung 6.2: Beispiel zur Block-`if`-Konstruktion

6.1.2 Das `if - else if - end if`-Konstrukt

Dieses Konstrukt hat folgende Form:

```
if ( < logischer Ausdruck 1 > ) then

    Anweisung 1
    ...
    Anweisung n

else if ( < logischer Ausdruck 2 > ) then

    Anweisung m
    ...
    Anweisung o

else

    Anweisung p
    ...
    Anweisung q

end if
```

Die grafische Dartellung im flow chart zeigt Abbildung 6.3. Im Programmcode könnte man z.B. im konkreten Beispiel schreiben:

```
implicit none
real :: diskriminante
diskriminante = b**2 - 4.0*a*c

if ( diskriminante < 0.0) then

    write(*,*) 'Keine reellen Nullstellen'

else if ( diskriminante == 0.0) then

    write(*,*) 'Die doppelte Nullstelle lautet:'
    ! Nullstelle berechnen und ausgeben

else


    write(*,*) 'Die beiden Nullstellen sind:' ! die beiden Nullstellen
    berechnen und ausgeben


end if
```

Bemerkungen:
Diese Verzweigungs-Konstruktion ist sehr flexibel:

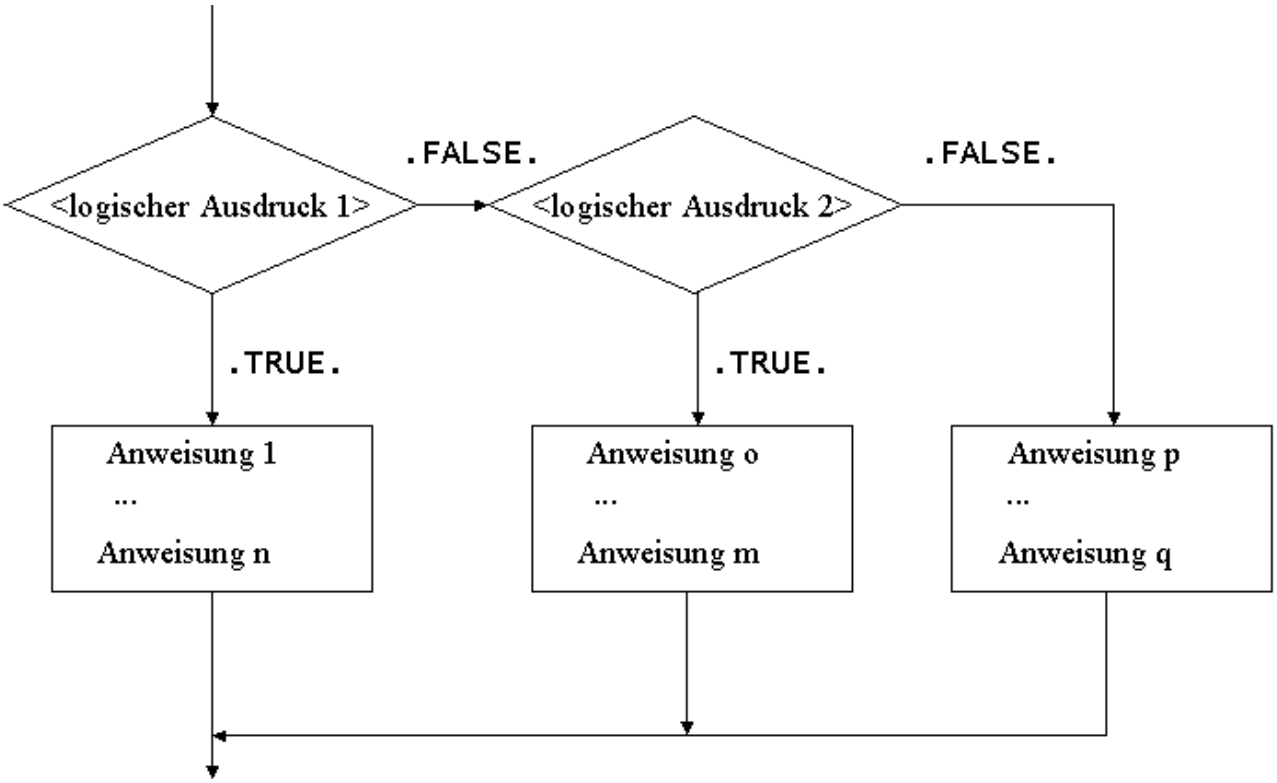

Abbildung 6.3: Beispiel zur `if - else if - end if`-Konstruktion

- Im `if - else if - else - end if`-Konstrukt ist der `else`-Block optional.
- Es könnten weitere `else if`-Blöcke „eingehängt“ werden.
- Verwenden Sie - wo möglich - das `if - else if -else - end if`-Konstrukt statt separater `if then`-Abfragen. Auf diese Art können Sie, falls diese Konstrukte in Ihrem Programm (z.B. Vektor-Matrix-Auswertungen in hochdimensionalen Matrizen) oft durchlaufen werden müssen, beträchtlich Rechenzeit einsparen.
- Dem `if - else if -else - end if`-Konstrukt kann ein Name zugewiesen werden. Dies ist besonders vorteilhaft bei sehr langen Schleifen-Konstruktionen mit langen Anweisungsblöcken und ineinander verschachtelten Schleifen, z.B.

```
aussen : if ( x < 0.0 ) then
   ...
   innen : if ( y < 0.0 ) then
      ...
   end if innen
   ...
end if aussen
```

Vergleichen Sie die gute (`program nested_ifs`) und die schlechte Realisierung (`program nested_ifs_bad`) insbesondere bezüglich des Informationsgehalts der Fehlermel-

dungen, die Ihnen Ihr Compiler liefert, wenn Sie das zu der mittleren Schleife gehörende end if „zufällig“ versehentlich löschen würden.

```
program nested_ifs
! Die verschachtelten Schleifen werden nun benannt,
! um das Programm ein klein wenig uebersichtlicher zu gestalten

implicit none
real :: x = 1.0
real :: y = 2.0
real :: z = 4.0

write(*,*) x, y, z
aussen: if ( x >= 0.0) then
        x = 2.0 * z
        mitte: if ( x >= 0.0 ) then
                x  = 0.5 * y - z
                innen: if ( z >= 0.0) then
                        z= x*y
                end if innen
                y = z * x
        end if mitte
        y = y**2
end if aussen
write(*,*) x, y, z
end program nested_ifs
```

```
program nested_ifs_bad

! so unuebersichtlich sollte man es tunlichst nicht machen

implicit none
real :: x = 1.0
real :: y = 2.0
real :: z = 4.0
write(*,*) x, y, z
if ( x >= 0.0) then
x = 2.0 * z
if ( x >= 0.0 ) then
x  = 0.5 * y - z
if ( z >= 0.0) then
z= x*y
! Bitte loeschen Sie das 'end if' der folgenden Zeile und
! vergleichen Sie die Compiler-Fehlermeldung mit derjenigen
! des Programms mit den benannten if-Schleifen
! wenn Sie die korrespondierende Zeile loeschen
!    end if
y = z * x
end if
!    end if
y = y**2
end if
write(*,*) x, y, z
end program nested_ifs_bad
```

6.1.3 Die einzeilige `if`-Konstruktion

Braucht man nur eine Anweisung, die als Folge einer logisch wahren Abfrage ausgeführt werden soll, so kann man als Einzeiler

```
if ( < logischer Ausdruck > ) < Anweisung >
```

als kürzere Version von

```
if ( < logischer Ausdruck > ) then

    < Anweisung >

end if
```

verwenden.

6.1.4 Das `select case`-Konstrukt

Will man eine Art Auswahlmenü programmieren, könnte man dies über eine `if then - else if - else if - ...- else - end if`-Konstruktion tun. In Fortran 90/95 existiert dazu eine Alternative. Falls ein Ausdruck, nach dessen Wert Sie jeweils verschiedene weitere Programmanweisungsblöcke abarbeiten lassen wollen, vom Datentyp `integer`, `character` oder `logical` ist, bietet das `case`-Konstrukt dafür eine übersichtliche Alternative an:

```
<Name des case-Konstr.>: select case ( <case-Ausdruck> )

    case (<case-Fall 1>) <Name des case-Konstrukts>
       Anweisungsblock 1
    case (<case-Fall 2>) <Name des case-Konstrukts>
       Anweisungsblock 2
    case (<case-Fall 3>) <Name des case-Konstrukts>
       Anweisungsblock 3
    case ...
       ...
    case default
       alternativer Anweisungsblock
       falls keiner der oben abgefragten
       Fälle zutreffen sollte

end select <Name des case-Konstrukts>
```

Die Benennung des `select case`-Konstrukts ist optional, aber wiederum bei Auswahlmenüs mit sehr langen Anweisungsblöcken empfehlenswert!
Beispielprogramm:

```
program select_case

implicit none
integer :: temp_wert

write(*,*) 'Geben Sie den abgelesenen Temperaturwert ein!'
read(*,*) temp_wert

temp_aussage: select case (temp_wert)
        ! der case-selector temp_wert muss von Datentyp
        ! integer, character oder logical sein
        case (:-1)
                write(*,*) 'unter 0 Grad Celsius'
        case (0)
                write(*,*) 'Genau 0 Grad Celsius'
        case (1:20)
                write(*,*) 'Noch ziemlich kuehl heute'
        case (21:30)
                write(*,*) 'Es scheint heute warm zu sein'
        case (31:36)
                write(*,*) 'Es ist reichlich heiss draussen'
        case default
                write(*,*) 'Legen Sie Ihr Thermometer besser in den Schatten!'
end select temp_aussage
end program select_case
```

Wie man in dem Beispiel sieht, können hinter den einzelnen `case` Einträgen statt einzelner selektiver Werte auch Intervalle (diese werden mit einem Doppelpunkt angegeben) verwendet werden.
Statt Intervallen kann man bei Bedarf auch mehrere durch Kommatas angegebene Werte verwenden, z.B. wenn auf eine Frage ein Zeichen eingelesen wird, das `j,J,y` oder `Y` sein darf, wenn im folgenden ein bestimmter Anweisungsblock ausgeführt werden soll:

```
...
character :: antwort
...
read(*,*) antwort

select case (antwort)

    case ('j','J','y','Y')
    ! Anweisungsblock fuer den Fall, dass die Antwort positiv war

end select
...
```

Wird wie im obigen Beispiel der Default-Fall nicht benötigt, kann der Teil ab `case default` ganz weggelassen werden.

6.1.5 Die `stop`-Anweisung (Programmabbruch)

Manchmal ist es notwendig, sobald ein Ausdruck einen bestimmten Wert annimmt, keinerlei weitere Programmanweisungen ausführen zu lassen, sondern stattdessen die Abarbeitung des Programms sofort zu beenden. Dazu dient die `stop`-Anweisung:

```
stop
```

Manchmal ist es sinnvoll den Anwender zu informieren, warum die Programmabarbeitung abgebrochen wird. Dazu kann man vorher noch eine `write`-Anweisung mit den zu übermittelnden Informationen einschieben. Will man nur einen einzigen Satz ausgeben könnte man dies tun, indem man eine Zeichenkette hinter dem `stop` anfügt, z.B.

```
stop 'Wert von a=0 (Programmabbruch, sonst division by zero - error)'
```

6.2 Schleifen

Möchte man, dass einzelne Anweisungsblöcke unter bestimmten Voraussetzungen mehrmals abgearbeitet werden, so kann man dies durch Schleifenkonstruktionen realisieren.

6.2.1 Die `do - if exit - end do`-Schleife

Dies ist die flexibelste Form der Schleifen. Die allgemeine Syntax lautet

```
do

    Anweisungsblock 1

if ( < logischer Ausdruck > ) exit

    Anweisungsblock 2

end do
```

und im Programmablaufplan kann man die Funktionsweise dieser Schleife wie in Abbildung 6.4 gezeigt darstellen. Zunächst werden die Anweisungen abgearbeitet (Anweisungsblock 1), die unmittelbar auf den Kopf der Schleife, nach dem `do` folgen. Mit der `if`-Abfrage wird geprüft, ob der Wert eines logischen Ausdruck wahr ist; wäre dies der Fall, würde wegen des `exit` das Konstrukt verlassen werden und als nächste Anweisung die unmittelbar hinter dem `end do` folgende Anweisung abgearbeitet werden.
Ist der logische Ausdruck bei der Abfrage nicht wahr, so wird der Anweisungsblock 2 abgearbeitet und danach anschließend wiederum Anweisungsblock 1, bevor erneut geprüft wird, ob der logische Ausdruck jetzt den Wert `.TRUE.` hat. Solange dies nicht der Fall ist, wird abermals Anweisungsblock 2 ausgeführt und danach Anweisungsblock 1. Erst wenn der logische Wert wahr wird, kann die Schleife verlassen werden.
Je nach Abfrage kann es passieren, dass die Schleife nie verlassen werden kann. In diesem Fall hätte man eine Endlos-Schleife programmiert.
Beispielprogramm:

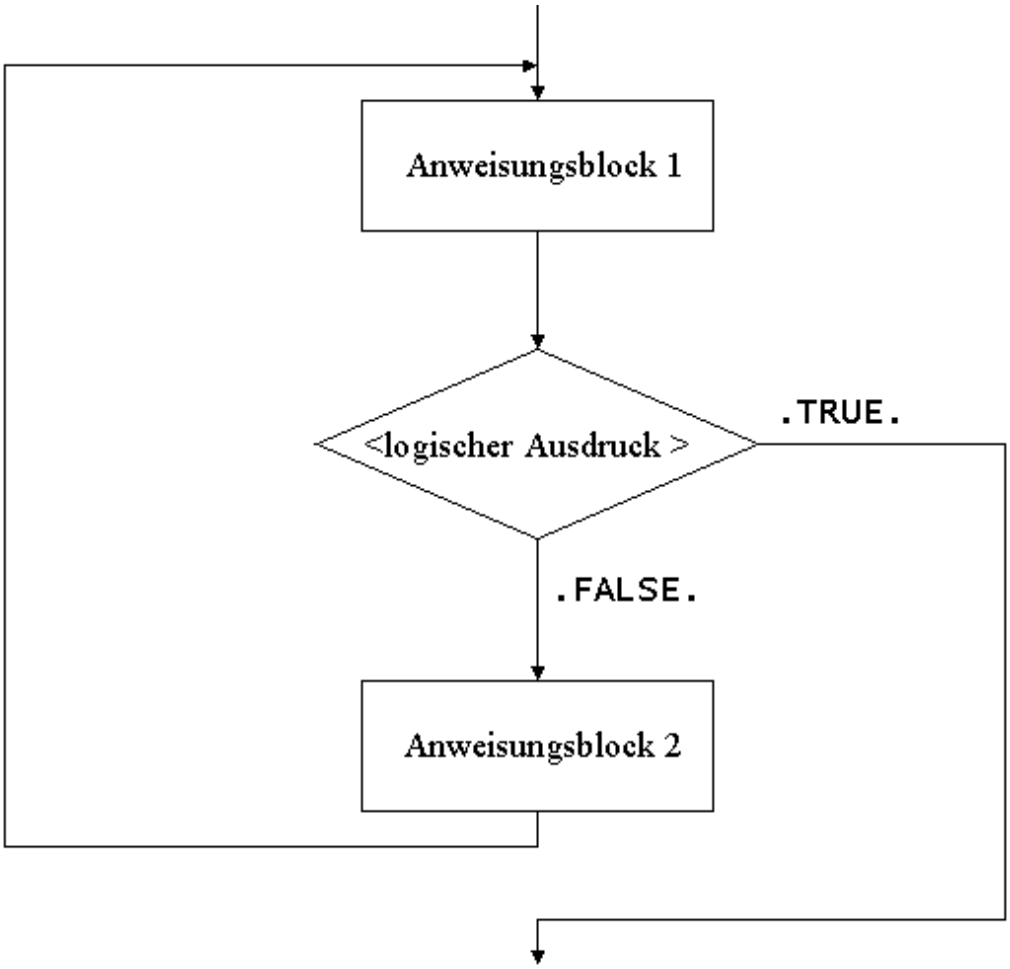

Abbildung 6.4: Beispiel zur `do - if exit - end do`-Schleife

```
program do_schleife_mit_exit

implicit none
integer :: i

i = 1
do
        write(*,*) 'i = ', i
        if ( i == 5 ) exit
        write(*,*) 'i = ', i, '    i**2 = ', i**2
        i = i + 1
end do
write(*,*) 'nach end do angekommen'
write(*,*) ' i = ', i
end program do_schleife_mit_exit
```

Ein weiteres Anwendungsbeispiel:
Um den Mittelwert und die Standardabweichung bei einer beliebigen Zahl an nicht-negativen Werten zu berechnen (z.B. um eine Punktezahl-Statistik von Klausurpunkten zu erstellen), kann als „Trigger“ für den Ausstieg aus der Daten-Einlese-Schleife die Eingabe eines negativen Punktwerts, der in diesem Fall in der Praxis nicht auftreten kann, eingesetzt werden.
Der Mittelwert von N Daten ist gegeben durch

$$x_{mittel} = \frac{\sum_{i=1}^{N} x_i}{N}$$

und die Standardabweichung durch

$$s = \sqrt{\frac{N\left(\sum_{i=1}^{N} x_i^2\right) - \left(\sum_{i=1}^{N} x_i\right)^2}{N\,(N-1)}}.$$

Beispielprogramm:

```
program statistik

! Berechnet den Mittelwert und die Standardabweichung
! einer beliebigen Anzahl reeller Daten

implicit none
integer :: n = 0        ! Anzahl der Daten
real :: x               ! ein Datenwert
real :: x_mittel        ! der Mittelwert der eingelesenen Daten
real :: std_dev = 0.0   ! die Standardabweichung
real :: sum_x   = 0.0   ! die Summe der eingegebenen Datenwerte
real :: sum_x_2 = 0.0   ! die Summe der quadrierten Datenwerte

write(*,*) 'Berechnung des Mittelwertes und der Standardabweichung'
write(*,*) 'nichtnegativer reeller Datenwerte'
write(*,'(A//)') 'Abbruch nach Eingabe einer negativen Zahl'

```

```
do
        write(*,'(A$)') 'Datenwert␣eingeben:␣␣'
        read(*,*) x
        if ( x < 0 ) exit    ! nach Eingabe negativen einer Zahl wird das
        !Programm nach end do fortgesetzt
        n       = n + 1
        sum_x   = sum_x + x
        sum_x_2 = sum_x_2 + x**2
end do

if ( n < 2 ) then
write(*,*) 'Es␣muessen␣mindestens␣2␣Datenwerte␣eingegeben␣werden!'
else
x_mittel = sum_x / real(n)
std_dev  = sqrt( (real(n) * sum_x_2 - sum_x**2) / (real(n)*real(n-1)))
write(*,*)
write(*,*) 'Der␣Mittelwert␣der␣Daten␣betraegt:␣', x_mittel
write(*,*) 'Die␣Standardabweichung␣betraegt:␣␣␣', std_dev
write(*,*) 'Anzahl␣der␣eingelesenen␣Datensaetze', n
end if

end program statistik
```

Wie das Programm `statistik` arbeitet und insbesondere wie die `do - if exit - end do`-Schleife eingesetzt wird, zeigt der Programmablaufplan in Abbildung 6.5. Generell gilt: Die `do - if exit - end do`- Schleife ist recht flexibel.
Wenn man keine Anweisungen vor dem `if (..) exit` benötigt, kann dieser Anweisungsblock auch weggelassen werden. In diesem Fall kann man die `do - if exit - end do` in eine `do while - end do`-Schleife umbauen.

6.2.2 Die `do while - end do`-Schleife

Diese Schleife hat die allgemeine Syntax

```
do while (< logischer Ausdruck >)

    Anweisungsblock

end do
```

Zu Beginn des Schleifenkopfes wird geprüft, ob der logische Ausdruck den Wert `.TRUE.` aufweist. Nur wenn dies der Fall sein sollte, wird der Anweisungsblock ausgeführt und dann erneut geprüft, wie der Wert des logischen Ausdrucks jetzt ist. Solange dieser Wert wahr ist, wird wird jeweils der Anweisungsblock abgearbeitet, ist der Wert des logischen Ausdrucks hingegen `.FALSE.` wird die Programmabarbeitung nach dem `end do` fortgesetzt.
Beispielprogramm:

```
program do_while

implicit none
integer :: i = 1, n = 1

do while ( n < 20)
```

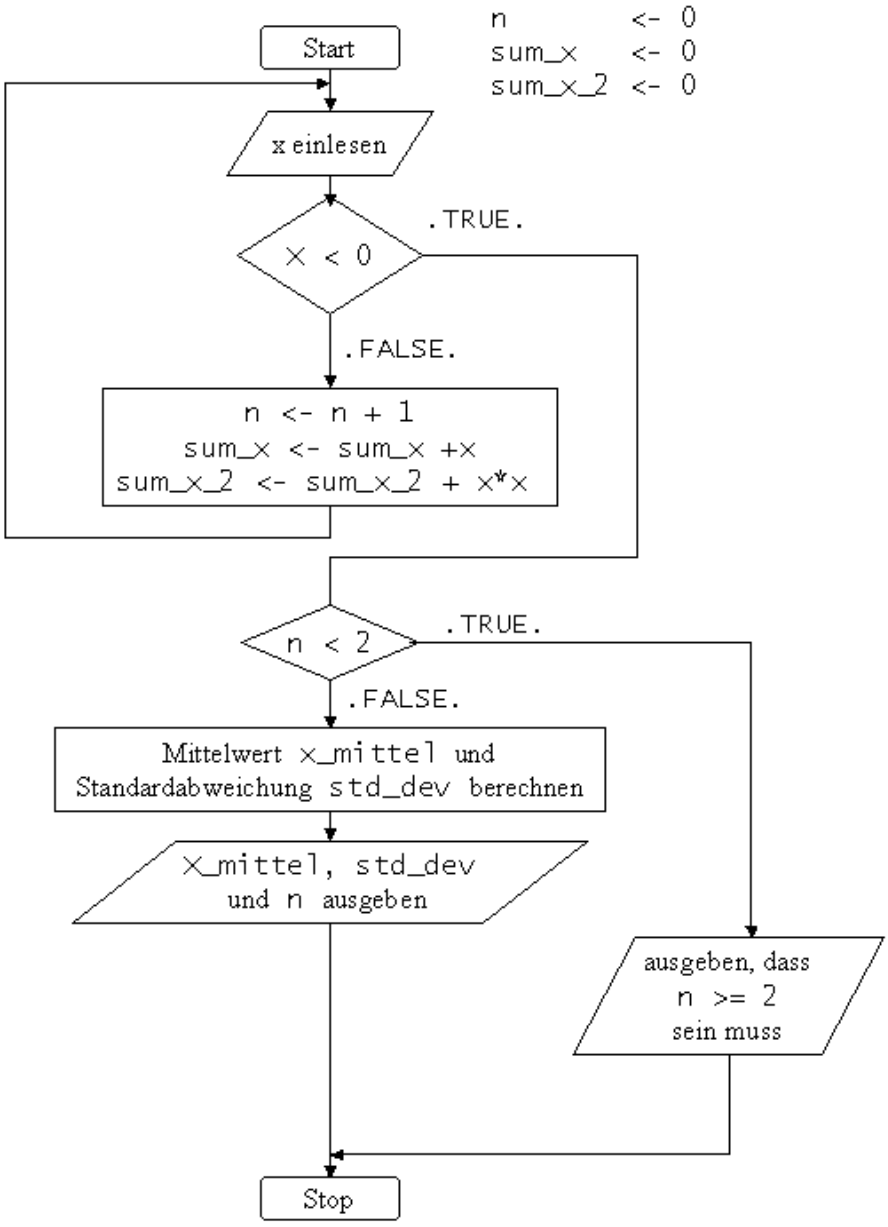

Abbildung 6.5: Beispielprogramm `statistik`

```
7           write(*,*) i, n
8           n = n + i
9           i = i + 1
10  end do
11  end program do_while
```

Das Programm liefert folgende Bildschirmausgabe:

```
1 1
2 2
3 4
4 7
5 11
6 16
```

Alternative Realisierung: Beispielprogramm:

```
1   program do_ersatz_do_while
2
3   implicit none
4   integer :: i = 1, n = 1
5
6   do
7           if (.NOT. ( n < 20)) exit
8           write(*,*) i, n
9           n = n + i
10          i = i + 1
11  end do
12  end program do_ersatz_do_while
```

Diese Realisierung liefert folgende Bildschirmausgabe:

```
1 1
2 2
3 4
4 7
5 11
6 16
```

Die `do while`- Konstruktion lässt sich ersetzen durch:

```
do

    if ( .not. < logischer Ausdruck > ) exit Anweisungsblock

end do
```

6.2.3 Die Zählschleife

Im Gegensatz zu den obigen flexiblen Schleifenkonstruktionen ist bei Zählschleifen (engl. counting loops) bereits beim Einstieg in die Schleife festgelegt, wie oft die Schleife durchlaufen werden soll. Eine Zählschleife folgt der Syntax

```
do schleifenvariable = anfangswert, endwert, schrittweite

    Anweisung 1 ... Anweisung n

end do
```

Der sogenannte Index der Zählschleife, die `schleifenvariable` kann vom Datentyp `integer` oder `real` sein. An der Stelle von `anfangswert`, `endwert`, `schrittweite` können Werte, Konstanten, Variablen und Ausdrücke des entsprechenden Datentyps stehen. Wird die `do`-Zeile ausgewertet, so werden die Ausdrücke für `anfangswert`, `endwert`, `schrittweite` ausgewertet und sind damit festgelegt. Das Flowchart-Symbol für die Zählschleife zeigt Abbildung 6.6. Zu Beginn der Schleife wird der Wert der Schleifenvariable auf den Anfangswert ge-

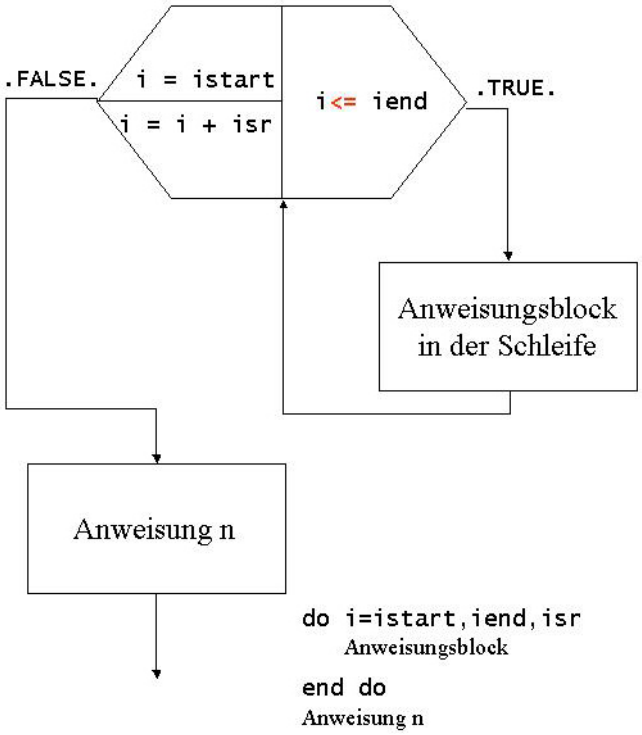

Abbildung 6.6: Zählschleife

setzt. Ist dieser Wert kleiner als der Endwert (bei positiver Schrittweite), so wird der in der Schleife enthaltene Anweisungsblock abgearbeitet. Mit jedem Schleifendurchlauf wird am Ende des Anweisungsblocks zu dem Wert der Schleifenvariablen der Zahlenwert der angegebenen Schrittweite hinzuaddiert.
Im Diagramm gilt das rot eingezeichnete Symbol <= für positive Schrittweiten. Ist die Schrittweite negativ wird mit jedem Schleifendurchlauf die Schleifenvariable um die Schrittweite reduziert und der Block an Fortran-Anweisungen abgearbeitet, bis die Schleifenvariable kleiner als der Endwert geworden ist.
Sobald der Wert der Schleifenvariable größer als der an 2. Position im Schleifenkopf angegebene Endwert sein sollte (bei positiver Schrittweite bzw. kleiner bei negativer Schrittweite), wird das Programm unterhalb von end do fortgesetzt.
Gibt man im Zählschleifen-Kopf keinen Zahlenwert für die Schrittweite an, so wird als

Schrittweite 1 angenommen.
Beispielprogramm:

```
program zaehlschleife

implicit none
integer :: i, n = 1

do i = 1, 5
        write(*,*) i, n
        n = n + i
end do
end program zaehlschleife
```

Das Programm liefert folgende Bildschirmausgabe:

```
1 1
2 2
3 4
4 7
5 11
```

Man beachte, dass der Wert der Schleifenvariable in Fortran (anders als in C) innerhalb des in der Schleife nicht durch eine explizite Anweisung verändert werden darf. Ein derartiger Versuch würde beim Compilieren zu einem Fehler führen.

Warum der Schleifenindex vom Datentyp `integer` sein sollte

Um Fehler in der Anzahl der Schleifendurchgänge aufgrund der endlichen Darstellungsgenauigkeit der Zahlen des Datentyps `real` zu vermeiden, sollten in Fortran 90 Schleifen besser nur über `integer`-Konstruktionen realisiert werden. (Fortran 95 lässt bereits Zählschleifen nur noch über `integer`-Indizes zu).
Ein Beispiel, bei dem die Schleife über `real` zu einem unerwünschten Verhalten führen kann, lässt sich z.B. mit dem Programm `sinus` aufzeigen.
Beispielprogramm:

```
program sinus

! was man vermeiden sollte:
! real-Schleifenvariablen koennen aufgrund von Rundungsfehlern
! aufgrund der begrenzten Zahlendarstellungsgenauigkeit zu
! Abweichungen in den Durchlaufzahlen fuehren
! (system- und compilerabhaengig; testen z.B. mit a=0.0, b=1.0, s = 0.001)
! besser => Schleifenindizes vom Datentyp integer

implicit none
real :: a, b, s, x
integer :: n = 0

write(*,*) 'Tabellierung von sin(x) im Intervall [a,b] mit Schrittweite s'
write(*,'(A$)') 'Bitte geben die untere Intervallgrenze a ein: '
read(*,*) a
write(*,'(A$)') 'Bitte geben die obere Intervallgrenze b ein:  '
```

```
18 read(*,*) b
19 write(*,'(A$)') 'Bitte␣geben␣die␣Schrittweite␣s␣ein:␣␣␣␣␣␣␣␣␣␣␣'
20 read(*,*) s
21
22 write(*,*) '␣␣␣␣x␣␣␣␣␣␣␣␣␣sin(x)␣'
23 write(*,*) '----------------------'
24
25 do x = a, b, s
26         write(*,'(F12.6,3X,F6.4)') x, sin(x)
27         n = n + 1
28 end do
29
30 write(*,*) 'Anzahl␣der␣Schleifendurchlaeufe:␣␣', n
31 write(*,*) 'berechnete␣Zahl␣an␣Durchlaeufen:'
32 write(*,*) 'Theorie:␣max(int(␣(b-a+s)/s␣),␣0):', max(int( (b-a+s)/s ), 0)
33
34 end program sinus
```

Zum Beispiel erhält man auf einzelnen Systemen und einzelnen Compilern, wenn man die als Kommentar eingegebenen Werte angibt, aufgrund von Rundungsfehlern einen Schleifendurchlauf weniger als man erwarten würde.
Die Ursache für dieses Verhalten liegt in der notwendigerweise endlichen Darstellungsgenauigkeit von Zahlen das Datentyps `real` begründet (meist 4 Byte), so dass beim Aufbau von Schleifen über den Datentyp `real` die Gefahr besteht, dass aufgrund von Rundungsfehlern Schleifen nicht immer mit der erwarteten Präzession (d.h. mit der erwarteten Zahl an Durchläufen) abgearbeitet werden.
Deshalb sollten Schleifen immer auf `integer`-Indizes umgestellt werden.
Beispielprogramm:

```
1  program sinus_integer_schleife
2
3  ! die Verbesserung von sinus.f90
4
5  ! was man vermeiden sollte: (sinus.f90)
6  ! real-Schleifenvariablen koennen aufgrund von Rundungsfehlern
7  ! aufgrund der begrenzten Zahlendarstellungsgenauigkeit zu
8  ! Abweichungen in den Durchlaufzahlen fuehren
9  ! (system- und compilerabhaengig; testen z.B. mit a=0.0, b=1.0, s = 0.001)
10 ! besser => Schleifenindizes vom Datentyp integer
11
12 ! hier die korrigierte Version
13
14 implicit none
15 real :: a, b, s, x
16 integer :: n = 0
17
18 integer :: i, schrittanzahl
19
20 write(*,*) 'Tabellierung␣von␣sin(x)␣im␣Intervall␣[a,b]␣mit␣Schrittweite␣s'
21 write(*,'(A$)') 'Bitte␣geben␣die␣untere␣Intervallgrenze␣a␣ein:␣'
22 read(*,*) a
23 write(*,'(A$)') 'Bitte␣geben␣die␣obere␣Intervallgrenze␣b␣ein:␣␣'
24 read(*,*) b
25 write(*,'(A$)') 'Bitte␣geben␣die␣Schrittweite␣s␣ein:␣␣␣␣␣␣␣␣␣␣␣'
```

```
26 read(*,*) s
27
28 write(*,*) '␣␣␣␣x␣␣␣␣␣␣␣␣␣sin(x)␣'
29 write(*,*) '---------------------'
30 !
31 ! der unguenstige Teil aus sinus.f90
32 !
33 !       do x = a, b, s
34 !              write(*,'(F12.6,3X,F6.4)') x, sin(x)
35 !              n = n + 1
36 !       end do
37
38 ! dieser wird besser ersetzt durch
39
40 schrittanzahl = max(nint((b-a+s)/s),0)
41 x = a
42 n = 0
43 do i = 1, schrittanzahl
44         write(*,'(F12.6,3X,F6.4)') x, sin(x)
45         x = x + s
46         n = n + 1
47 end do
48
49 write(*,*) 'Anzahl␣der␣Schleifendurchlaeufe:␣␣', n
50 write(*,*) 'berechnete␣Zahl␣an␣Durchlaeufen:'
51 write(*,*) 'Theorie:␣max(int(␣(b-a+s)/s␣),␣0)␣:', max(int( (b-a+s)/s ), 0)
52 write(*,*) 'Theorie:␣max(nint(␣(b-a+s)/s␣),␣0):', schrittanzahl
53
54 end program sinus_integer_schleife
```

Nach dem neuen Fortran 95 - Standard wird vorausgesetzt, dass Zählschleifen immer über den Datentyp `integer` aufgebaut werden. Dadurch kann gewährleistet werden, dass stets die gewünschte Anzahl an Schleifendurchgängen abgearbeitet wird, weil der Rechner auf Basis des Datentyps `integer` exakt „zählt“.
Will man z.B. ein Intervall mit `x` von `xu` bis `xo` mit der Schrittweite `dx` durchlaufen, sollte man dies nicht über

```
real :: x, xu, xo, dx

do x = xu, xo, dx

    ! Anweisungsblock

end do
```

tun, sondern über einen Schleifenindex vom Datentyp `integer` die Schleife durchlaufen. Dazu muss die Anzahl der Intervallschritte `s` berechnet werden:

```
s = max ( nint( (xo-xu+dx)/dx ), 0)
```

Die Schleife lässt sich dann realisieren über

```
integer :: i, s
real :: x, xu, xo, dx
```

```
s = max ( nint( (xo-xu+dx)/dx ), 0)

do i = 0, s-1

    x = xu + real(i) * dx
    ! Anweisungsblock

end do
```

6.2.4 Das Verlassen von `do`-Schleifen mit `exit`

`do`-Schleifen können bei Bedarf jederzeit mit `exit` und `cycle` verlassen werden. Wird innerhalb einer `do`-Anweisung eine `exit`-Anweisung abgearbeitet, so wird die Programmausführung unmittelbar nach dem zugehörigen `end do` fortgesetzt. Siehe dazu das Beispiel `statistik` von oben.
Sind die `do`-Schleifen ineinander geschachtelt, so wird mit `exit` die aktuelle Schleife verlassen, und das Programm hinter dem `end do` der aktuellen Schleife fortgesetzt.
Beispielprogramm:

```
1  program do_do_do_exit
2
3  implicit none
4  integer :: i, m, n
5
6  do i = 1, 3
7          do m = 1, 3
8                  if ( m == 2) exit
9                  do n = 1, 3
10                 write(*,*) 'i␣=␣', i, ';␣m␣=␣', m, ';␣n␣=␣', n
11         end do
12         end do
13 end do
14
15 end program do_do_do_exit
```

Das Programm liefert folgende Bildschirmausgabe:

```
i = 1 ; m = 1 ; n = 1
i = 1 ; m = 1 ; n = 2
i = 1 ; m = 1 ; n = 3
i = 2 ; m = 1 ; n = 1
i = 2 ; m = 1 ; n = 2
i = 2 ; m = 1 ; n = 3
i = 3 ; m = 1 ; n = 1
i = 3 ; m = 1 ; n = 2
i = 3 ; m = 1 ; n = 3
```

Handelt es sich allerdings um benannte `do`-Schleifen und hinter der `exit`-Anweisung wird der Name einer Schleife aus der Schachtelungshierarchie angegeben, so wird genau diese Schleife beendet und das Programm am Ende einer Schleife des angegebenen Namens und

damit in der Hierarchiebene oberhalb dieser angegebenen Schleife fortgesetzt.
Beispielprogramm:

```
program do_do_do_exit

implicit none
integer :: i, m, n

aussen: do i = 1, 3
        mitte: do m = 1, 3
                if ( m == 2) exit aussen
                innen: do n = 1, 3
                        write(*,*) 'i = ', i, '; m = ', m, '; n = ', n
                end do innen
        end do mitte
end do aussen

end program do_do_do_exit
```

Das Programm liefert folgende Bildschirmausgabe:

```
i = 1 ; m = 1 ; n = 1
i = 1 ; m = 1 ; n = 2
i = 1 ; m = 1 ; n = 3
```

6.2.5 Die Verwendung von `cycle` in `do`-Schleifen

Die Abarbeitung von Teilen von `do`-Schleifen lässt sich ebenfalls mit `cycle` umgehen. Im Vergleich zu `exit` hat `cycle` die umgekehrte Wirkung: bei `cycle` wird an den Kopf der `do`-Schleife gesprungen.
Beispielprogramm:

```
program do_with_cycle

implicit none
integer :: i

do i = 1, 5
        write(*,*)
        write(*,*) 'vor der Schleife:  i = ', i
        if ( i == 3) cycle
        write(*,*) 'in der Schleife:    i = ', i
end do
write(*,*)
write(*,*) 'nach der Schleife: i = ', i
end program do_with_cycle
```

Das Programm liefert folgende Bildschirmausgabe:

```
vor der Schleife:  i = 1
in der Schleife:   i = 1

vor der Schleife:  i = 2
in der Schleife:   i = 2
```

```
vor der Schleife: i = 3

vor der Schleife: i = 4
in der Schleife: i = 4

vor der Schleife: i = 5
in der Schleife: i = 5

nach der Schleife: i = 6
```

Auch hier lässt sich bei geschachtelten `do`-Schleifen über die Benennung der Schleifen und der Angabe des entsprechenden Schleifennamens hinter `exit` gezielt an den Kopf der entsprechenden Schleife springen.
Grundstruktur einer `do`-Schleife mit `cycle` und `exit`:

```
do

    ! Anweisungsblock 1

if ( ) cycle ! falls die Bedingung erfuellt ist, gehe zum Schleifenkopf

    ! Anweisungsblock 2

if ( ) exit ! falls diese Bedingung erfuellt ist, verlasse die Schleife

    ! Anweisungsblock 3

end do
```

Bei Bedarf kann natürlich dieses Konstrukt angepasst werden, z.B. kann die Reihenfolge von cycle und exit vertauscht und/oder weitere Abfragen eingefügt werden.

6.2.6 Mit `do - exit - cycle - end do` Eingabefehler des Anwenders abfangen

Wird von einem Anwender die Eingabe von Werten erwartet, kann es passieren, dass sich dieser vertippt und z.B. statt des erwarteten Zahlenwertes einen Buchstaben eingibt. Dieser Fehler würde normalerweise zu einem Programmabbruch aufgrund eines Laufzeitfehlers (*runtime error*) führen. Dies lässt sich z.B. mit dem Programm `eingabefehler` testen. Beispielprogramm:

```
1  program eingabefehler
2
3  implicit none
4  real    :: x
5
6  write(*,'(A$)') 'Geben␣Sie␣eine␣reelle␣Zahl␣ein:␣'
7  read(*,*) x
8  write(*,*) 'Eingelesen␣wurde␣:␣', x
9
10 end program eingabefehler
```

Der bisher zum Einlesen (von der Standardeingabe) eingesetzte Befehl, z.B.

```
real :: x

read(*,*) x
```

bietet eine Erweiterung, die es uns ermöglicht, Eingabefehler des Anwenders abzufangen

```
real :: x
integer :: io_err

read(*,*,iostat=io_err) x
```

Durch die Erweiterung `iostat= >Variable_vom_Datentyp_integer<` wird beim Einlesen durch `iostat=io_err` der Variable `io_err` dieses selbstgewählten Namens ein Zahlenwert vom Datentyp `integer` zugewiesen, die einem Fehlercode entspricht.
Nur wenn der Rückgabewert von `iostat`, welcher nun in der Variablen `io_err` gespeichert wurde, dem Wert `0` entspricht, ist konnte beim Einlesevorgang der Variablen `x` ein reeller Zahlenwert zugewiesen werden.
Hat der Anwender nun statt eines Zahlenwertes z.B. versehentlich einen Buchstaben eingegeben, so erfolgt bei Verwendung von `iostat=io_err` innerhalb der `read`-Anweisung diesmal kein Programmabbruch aufgrund eines Laufzeitfehlers - vielmehr wird stattdessen der hinter `iostat` angegebenen Variablen (hier: `io_err`) ein Fehlercode ungleich `0` zugewiesen.
Dieser Wert dieses Fehlercodes kann innerhalb einer geschickt konstruierten `do`-Schleife mit als Bedingung für `exit` bzw. `cycle` eingesetzt werden, um bei einer fehlerhaften Eingabe von Anwender erneute Eingabe zu erzwingen.
Beispielprogramm:

```
1  program eingabefehler_abfangen
2
3  implicit none
4  real    :: x
5  integer :: io_err
6
7  einlesen: do
8          write(*,'(A$)') 'Geben␣Sie␣eine␣reelle␣Zahl␣ein:␣'
9          read(*,*,iostat=io_err) x
10         if ( io_err == 0) exit
11         write(*,*) '=>␣Achtung:␣Eingabefehler,␣bitte␣Eingabe␣wiederholen!'
12         cycle
13 end do einlesen
14
15 write(*,*) 'Eingelesen␣wurde␣:␣', x
16
17 end program eingabefehler_abfangen
```

6.2.7 `do`-Ersatz für das veraltete `goto` aus FORTRAN 77

Mit `goto` und Angabe einer Zeilennummer (engl. *statement label*) wurden in FORTRAN 77 u.a. bedingungsabhängige Sprünge innerhalb des Anweisungsteils einer Programmeinheit

realisiert. Da die FORTRAN 77-Befehle weitestgehend in Fortran 90/95 enthalten sind, ist der Einsatz von `goto` zwar noch möglich, sollte aber zugunsten der besseren Übersichtlichkeit und Klarheit der Fortran 90/95 - Programme durch (benannte) `do`-Konstruktionen ersetzt werden.
Im folgenden soll in dem zugegebenermaßen einfachen und übersichtlichen FORTRAN 77 - Programm die `goto`-Anweisung ersetzt werden.
Beispielprogramm:

```
      PROGRAM ALT
      IMPLICIT NONE
      REAL X, SUMME
C
      WRITE(*,*)
     *    'Das␣Programm␣addiert␣die␣von␣Ihnen␣eingegebenen␣Zahlen'
      WRITE(*,*)
        WRITE(*,*) 'Wollen␣Sie␣die␣Summation␣abschliessen,'
      WRITE(*,*) 'so␣geben␣Sie␣als␣Zahl␣0␣ein'
      WRITE(*,*)
C
      SUMME = 0.0
      X = 0.0
C
100   CONTINUE
      SUMME = SUMME + X
      WRITE(*,'(1X,A$)') 'Geben␣Sie␣die␣Zahl␣ein:␣'
      READ(*,*) X
      IF ( X .NE. 0.0) GOTO 100
C
      WRITE(*,*)
      WRITE(*,*) 'Die␣Summe␣betraegt:␣', SUMME
      END PROGRAM
```

In dem Beispielprogramm wird, falls die Bedingung `X .NE. 0.0` wahr ist, zu der Anweisung mit dem *statement label* `100` gesprungen und der Programmablauf mit der Anweisung `CONTINUE` fortgesetzt. Die Befehlszeile, an die mit `GOTO` gesprungen werden soll, muss eine ausführbare Anweisung sein. Manchmal fungiert eine mit der entsprechenden Zeilenmarkierung versehene `CONTINUE`-Anweisung als „Einsprung-Markierung “ zu dem korrespondierenden `GOTO`-Befehl. Das `GOTO` lässt sich in Fortran 90/95 vollständig ersetzen durch eine `do`-Schleife. Beispielprogramm:

```
program neu

implicit none
real :: x = 0.0, summe = 0.0

write(*,*) 'Das␣Programm␣addiert␣die␣von␣Ihnen␣eingegebenen␣Zahlen'
write(*,*)
write(*,*) 'Wollen␣Sie␣die␣Summation␣abschliessen,␣so␣geben␣Sie␣als␣Zahl␣0␣ein'
write(*,*)

schleife: do
        summe = summe + x
        write(*,'(1X,A$)') 'Geben␣Sie␣die␣Zahl␣ein:␣'
        read(*,*) x
```

```
15          if ( x /= 0.0) cycle schleife
16          exit schleife
17  end do schleife
18
19  write(*,*)
20  write(*,*) 'Die␣Summe␣betraegt:␣', summe
21
22  end program neu
```

Kapitel 7

Ein- und Ausgabe von Dateien (File-I/O)

Die Ausgabe auf den Bildschirm reicht nicht mehr aus, sobald der Umfang der von einem Programm ermittelten Informationen größer wird. Schnell taucht der Wunsch auf, den Programmoutput dauerhaft zu sichern, um ihn später jederzeit wieder darauf zugreifen und die Ausgabedaten gegebenfalls mit weiteren Programmen weiterverarbeiten zu können. Was wir somit benögen, ist ein Verfahren, um von einem Programm heraus Informationen auf dauerhafte Speichermedien, wie z.B. die Festplatte zu sichern. Wünschenswert ist es ebenfalls, dass bei Bedarf Informationen direkt von einem Speichermedium eingelesen und von unserem Programm weiterverarbeitet werden können, ohne das jedesmal riesige Datensätze per Hand eingegeben werden müssen.
Was wir also benötigen, ist das Einlesen von und das Ausgeben auf Dateien. Im Englischen spricht man von *File - Input/Output* oder kurz von *File - I/O*.
Während im Hauptspeicher des Rechners (dem RAM oder Memory des Computers) auf jeden einzelnen Speicherplatz direkt zugegriffen werden kann (*direct access*), erfolgt der Zugriff auf die externen Speichermedien sequentiell (*sequential access*). D.h. die einzelnen Informationsbits müssen von einem Startpunkt aus der Reihe nach (*sequential*) eingelesen werden. Bei der Besprechung der `write(*,*)`-Anweisung wurde bereits erwähnt, dass das erste `*` für die Standardausgabe steht. Dies ist in der Regel der Bildschirm, das zweite Sternchen steht bisher für das listengesteuerte (oder Default-)-Ausgabeformat. An Stelle des ersten Sternchens kann nun eine sogenannte *i/o unit number* treten. Diese kann entweder einem Ausgabegerät z.B. einem Drucker oder einer Datei mit einem vorher zugewiesenen Namen zugeordnet sein. Das Verfahren, Ausgabegeräte über Ziffern kleiner 10 (z.B. den Drucker) anzusteuern, sollte nicht mehr verwendet werden. Erschwerend kommt hinzu, dass je nach Rechnerhersteller ist den Ziffern unter 10 jeweils ein anderes Peripheriegerät zugeordnet ist. Die `unit numbers` werden allerdings weiterhin benötigt und genutzt, und zwar als sogenannte *logical units* zur Datei-Ein- und Ausgabe auf die Festplatte. Durch eine `open`-Anweisung wird eine Zuordnung zwischen einer *unit number* und einer Datei eines bestimmten Namens auf der Festplatte geschaffen. Durch die Angabe dieser *unit number* in einer `write` oder `read`-Anweisung kann auf diese Datei sequentiell geschrieben oder gelesen werden.
Beispielprogramm:

```
1 program file_ein_und_ausgabe
2
3 implicit none
```

```
integer :: io_error, read_error
integer :: n, i
real    :: y

!--------------------------------------
! eine neue Datei zum Schreiben anlegen
!--------------------------------------

open(unit=25,file='beispiel.dat',status='new',action='write', &
iostat=io_error)

if ( io_error == 0) then
        do n = 1, 5
                write(25,*) n, sqrt(real(n))
        end do
else
        write(*,*) 'Beim OEffenen der Datei ist ein Fehler Nr.', &
        io_error,' aufgetreten'
end if
close(unit=25)  ! die Zuordung unit number 25 mit der Datei 'beispiel.dat'
! wieder aufheben
!----------------------------------------
! Inhalt einer vorhandenen Datei bei
! unbekannter Zeilenanzahl einlesen und
! auf dem Bildschirm ausgeben
!----------------------------------------

open(unit=20,file='beispiel.dat',status='old',action='read', &
iostat=io_error)

if ( io_error == 0) then
        n = 1
        do
                read(20,*,iostat=read_error) i, y
                if ( read_error > 0) then
                        stop "Datenfehler auf der Datei"
                else if (read_error < 0 ) then
                        exit ! Dateiende erreicht
                else
                        write(*,*) n,'-te Zeile:', i, y
                        n = n + 1
                end if
        end do
else
        write(*,*) 'Beim OEffenen der Datei ist ein Fehler Nr.', &
        io_error,' aufgetreten'
end if
close(unit=20)

!------------------------------------------------------------
! ein altes (bereits vorhandenes) File zum Schreiben oeffnen
! dabei werden die bereits vorhanden Daten ueberschrieben
!------------------------------------------------------------

open(unit=25,file='beispiel.dat',status='replace',action='write', &
```

```
59 iostat=io_error)
60
61 if ( io_error == 0) then
62         do n = 1, 5
63                 write(25,*) n, real(3*n)
64         end do
65 else
66         write(*,*) 'Beim␣OEffenen␣der␣Datei␣ist␣ein␣Fehler␣Nr.', &
67         io_error,'␣aufgetreten'
68 end if
69 close(unit=25)
70
71 end program file_ein_und_ausgabe
```

7.1 Die `open`-Anweisung

Durch eine `open`-Anweisung wird eine feste Zuordnung zwischen einer Datei und einer logischen (von Ihnen zu wählenden) *i/o unit number* hergestellt. Als Angaben innerhalb der `open`-Anweisung stehen der Reihe nach Angaben zu

- `unit=`<*unit number*>
  Als *unit number* muss eine nichtnegative `integer`-Zahl eingesetzt werden. Da die Zahlen unterhalb von 10 oft (je nach Hardwarearchitektur und Compiler verschiedenen) Peripheriegeräten (z.B. Drucker, Bildschirm) zugeordnet sein können, empfiehlt es sich, erst Werte ab 11 oder höher als *unit number* zu verwenden.

- `file=`<Dateiname>
  Der Dateiname muss in Form einer Zeichenkette mit den entsprechenden Anführungs- und Schlusszeichen angegeben werden. Wird nur ein einfacher Dateiname (ohne Pfadangabe) angegeben, so bezieht sich der Compiler auf das aktuelle Verzeichnis, aus dem heraus er aufgerufen wurde. Sollen Dateien in anderen Verzeichnissen angesprochen werden, so kann man den systemspezifischen Pfad angeben. Auf die entsprechende Unix- bzw. Windows-Notation ist zu achten.

- `status=`<„Schlüsselwort “>
  Hier sollte man den zu erwartenden Zustand der Datei angeben. Mögliche Optionen sind:

  - `'old'`
  - `'new'`
  - `'unknown'`
  - `'replace'`
  - `'scratch'`

  Die Option `'old'` verwendet man für Dateien, die bereits vorhanden sein sollten. Ist die Datei dieses Namens in dem angegebenen Verzeichnis auf der Festplatte nicht vorhanden, so liegt eine Fehlersituation vor. Bei der Option `'new'` darf die Datei noch

nicht vorhanden sein und wird durch die `open`-Anweisung neu angelegt. `'replace'` und `'unknown'` sind einander insofern gleich, als dass es keine Rolle spielt, ob die Datei des angegebenen Namens vorhanden ist oder nicht. Bei der Option `'replace'` wird, wenn eine Datei dieses Namens noch nicht exisitieren sollte, diese angelegt. Ist eine Datei dieses Namens bereits vorhanden, wird mit dem Öffnen deren Inhalt gelöscht. Auf `'unknown'` sollte aufgrund mangelnder Standard-Vorgaben aus Portabilitätsgründen `'unknown'` möglichst verzichtet werden. `'scratch'` bezeichnet eine temporäre Datei, die ausschliesslich zur Zwischenspeicherung von Daten während des aktuellen Programmdurchlaufs verwendet werden kann. Will man eine `'scratch'`-Datei anlegen, so reicht `open(unit=`*unit number*`,status='scratch')` aus.

- `action=<`„Aktionsangabe “`>`
  Hier gibt man an, was mit der Datei geschehen soll. Als Optionen sind möglich:

  - `'read'`
  - `'write'`
  - `'readwrite'`

  Die Angabe kann auch fehlen, dann wird die Datei sowohl zum Lesen als auch zum Schreiben geöffnet.

- `iostat=<` Name einer Variablen vom Datentyp `integer>`
  Die Variable vom Datentyp `integer` muss vorher natürlich vereinbart worden sein. In dieser Variablen wird gespeichert, wie erfolgreich das Betriebssystem, eine Verknüpfung zwischen der *unit number* und der angegebenenen Datei auf der Festplatte herstellen konnte. Ging dies reibungslos, ist der Rückgabewert gleich `0` (Null). Der Rückgabewert des Betriebssystems wird in der `integer`-Variablen gespeichert. Trat bei der `open`-Anweisung ein Fehler auf Betriebssystem-Ebene auf, so entspricht der Rückgabewert dem Fehler-Code (engl. *error code*) des Systems. Die vom Betriebssystem zurückgegebene Zahl ist abhängig vom Betriebssystem und unter Windows abhängig vom Compiler und steht jeweils für eine spezifische Fehlersituation.
  Unter Windows können Sie die Bedeutung der Fehler-Codes in der integrierten Entwicklungsumgebung des Compaq Visual Fortran Compilers in dem Hilfemenü mit dem Suchbegriff „`iostat` “ nachgucken. Sie erhalten dann eine Liste der relevanten Kapitelüberschriften. Eine Tabelle mit einer genaueren Erklärung der Fehlercodes finden Sie unter dem Titel „`Visual Fortran Runtime Errors`“ und speziell unter „`Run Time Errors Having No Numbers and Errors 1 Through 30`“.

Wurde mit `open` erfolgreich eine *unit number*, z.B. `20` eine Datei auf der Festplatte zum Schreiben geöffnet , so kann mit

```
write(20,*) <Variablenliste>
```

die in der Variablenliste gespeicherten Werte der Variablen nacheinander in eine Zeile geschrieben werden.

7.2 Ein Beispiel zum Erstellen einer Datei

Mit dem Programm `schreiben` lässt sich die Datei `wertetabelle.txt`

```
1 1.000000
2 2.000000
3 3.000000
4 4.000000
5 5.000000
```

in das aktuelle Verzeichnis auf der Festplatte schreiben.
Beispielprogramm:

```
program schreiben

implicit none
integer :: io_error
integer :: n

open(unit=20,file='wertetabelle.txt',status='new',action='write', &
iostat=io_error)

if ( io_error == 0) then
        do n = 1, 5
                write(20,*) n, real(n)
        end do
else
        write(*,*) 'Beim␣OEffenen␣der␣Datei␣ist␣ein␣Fehler␣Nr.', &
        io_error,'␣aufgetreten'
end if
close(unit=20)

end program schreiben
```

7.3 Ein einfaches Beispiel zum Lesen von Informationen aus einer vorhandenen Datei

Dieses Programm prüft nur auf Fehler, die in Zusammenhang mit `open` auftreten können.
Beispielprogramm:

```
program lesen

implicit none
integer :: io_error
integer :: n
integer :: i
real    :: y

open(unit=20,file='wertetabelle.txt',status='old',action='read', &
iostat=io_error)

if ( io_error == 0) then
```

```
13          do n = 1, 5
14                  read(20,*) i, y
15                  write(*,*) n,'-te Zeile:', i, y
16          end do
17  else
18          write(*,*) 'Beim OEffenen der Datei ist ein Fehler Nr.', &
19          io_error,' aufgetreten'
20  end if
21  close(unit=20)
22
23  end program lesen
```

Das Programm liefert folgende Bildschirmausgabe:

```
1 -te Zeile: 1 1.000000
2 -te Zeile: 2 2.000000
3 -te Zeile: 3 3.000000
4 -te Zeile: 4 4.000000
5 -te Zeile: 5 5.000000
```

Empfehlenswert ist es, zusätzlich auf Fehler zu prüfen, die während des Lesens eines Datensatzes auftreten können.

7.4 Fehlererkennung und Behandlung über `iostat` in der `read`-Anweisung

Zunächst soll anhand eines Beispiels gezeigt werden, wie mit `iostat` in einer `read`-Anweisung Fehlersituationen, die bei der Ausführung des Programms auftreten können, abfangen kann. Typischerweise würde ein Laufzeitfehler (*run time error*) auftreten, wenn mit `read` ein `real`-Wert eingelesen werden soll, aber bei der Ausführung der `read`-Anweisung ein `character`-Zeichen an die `real`-Variable übergeben wird.
Anwendungsbeispiel:
Mit dem einfachen Programm `real_wert_einlesen` wird die obige Situation nachgebildet.
Beispielprogramm:

```
1  program real_wert_einlesen
2
3  implicit none
4  real :: wert
5
6  write(*,*) 'Geben Sie einen Wert vom Datentyp real ein: '
7  read(*,*) wert
8  write(*,*) 'Ihre Eingabe : ', wert
9
10 end program real_wert_einlesen
```

Im Regelfall zeigt dieses Programm das gewünschte Verhalten:

```
Geben Sie einen Wert vom Datentyp real ein:
2.0
Ihre Eingabe : 2.000000
```

Gibt ein Anwender versehentlich statt einer Zahl einen Buchstaben ein, kommt es - wie beschrieben - während der Programmausführung zu einem Laufzeitfehler.

```
Geben Sie einen Wert vom Datentyp real ein:
a
forrtl: severe (59): list-directed I/O syntax error, unit -4, file /dev/pts/0
0: __FINI_00_remove_gp_range [0x3ff81a6c374]
1: __FINI_00_remove_gp_range [0x3ff81a6c8f4]
2: __FINI_00_remove_gp_range [0x3ff81a94e68]
3: __FINI_00_remove_gp_range [0x3ff81a94538]
4: werteinlesen_ [real_wert_einlesen.f90: 6, 0x120001ca4]
5: main [for_main.c: 203, 0x120001bcc]
6: __start [0x120001b48]
```

Durch eine Erweiterung der read-Anweisung mit Fehlerabfang-Routinen über iostat lässt sich der Quellcode erheblich verbessern. Beispielprogramm:

```
1  program real_wert_einlesen_mod
2
3  implicit none
4  real    :: wert
5  integer :: io_err
6
7  write(*,*) 'Geben Sie einen Wert vom Datentyp real ein: '
8  read(*,*,iostat=io_err) wert
9
10 write(*,*) 'Rueckgabewert von iostat:  io_err = ', io_err
11 if ( io_err == 0) then
12         write(*,*) 'Ihre Eingabe : ', wert
13 else
14         stop '=> Fehler in der Eingabe! Programmabbruch.'
15 end if
16
17 end program real_wert_einlesen_mod
```

In diesem Fall erhält man bei einem „Data type mismatch “ statt des Laufzeitfehlers folgende Bildschirmausgabe:

```
Geben Sie einen Wert vom Datentyp real ein:
a
Rueckgabewert von iostat:  io_err = 59
=> Fehler in der Eingabe! Programmabbruch.
```

Im Fehlerfall wird eine positive ganze Zahl zurückgegeben. Der genaue Zahlenwert ist betriebssystem- und compilerabhängig und lässt sich in der Dokumentation zum Compiler nachlesen. Nach Fortran-Standard ist nur festgelegt, dass die Zahl größer als 0 sein muss. Wenn Sie also Programme nach Fortran-Standard schreiben möchten (was sich zur besseren Portierbarkeit der Programme sehr empfiehlt), reicht zum „error handling “ die Abfrage, ob iostat größer als 0 sei, vollkommen aus.
Will man bei einer fehlerhaften Eingabe den Anwender erneut einen Wert eingeben lassen, so kann man dies mit einer zusätzlichen do-Schleife realisieren.
Beispielprogramm:

```
program real_wert_einlesen_cycle

implicit none
real    :: wert
integer :: io_err

do
        write(*,*) 'Geben Sie einen Wert vom Datentyp real ein: '
        read(*,*,iostat=io_err) wert
        if ( io_err == 0) then
                write(*,*) 'Ihre Eingabe : ', wert
                exit
        else
                cycle
        end if
end do

end program real_wert_einlesen_cycle
```

In diesem Fall erhält man als exemplarische Bildschirmausgabe

```
Geben Sie einen Wert vom Datentyp real ein:
a
Geben Sie einen Wert vom Datentyp real ein:
2.0
Ihre Eingabe : 2.000000
```

7.5 `iostat` in Zusammenhang mit dem Einlesen von Werten aus Dateien

Fehlerabfangroutinen lassen sich über `iostat` sowohl beim Öffnen einer Datei (`open`) als auch beim Einlesen von Werten aus Dateien (`read`) einsetzen. Insbesondere lässt sich über `iostat` bei `read` feststellen, wann das Dateiende erreicht wird.
Beispielprogramm:

```
program einlesen

! Beispiel zum Einlesen von Datensaetzen aus Dateien
! mit Fehlererkennung

implicit none
character(len=20) :: filename  ! Name des Files
integer :: anzahl = 0          ! Anzahl der eingelesenen Werte,
                               ! hier gleichzeitig Nummer des Datenstatzes
integer :: status_open         ! Rueckgabewert aus iostat bei open
integer :: status_read         ! Rueckgabewert aus iostat beim
                               ! Einlesen der Daten mit read
real :: wert                   ! eingeleser Wert

! Interaktive Eingabe des Filenamens

```

```
17 write(*,*) ' Bitte geben Sie den Namen der zu lesenden Datei'
18 write(*,'(A$)') ' (nicht in Anfuehrungszeichen eingeschlossen) an: '
19 read(*,'(A)') filename
20 write(*,*) ' Als Name der Datei wurde eingelesen: ', filename
21 write(*,*)
22
23
24 ! OEffnen der Datei mit Abfangen von I/O-Fehlern
25 open(unit=25, file=filename, status='OLD', action='READ', iostat=status_open)
26
27 oeffnen: if ( status_open == 0 ) then
28 ! Beim OEffnen der Datei sind keine Fehler aufgetreten
29 ! es geht weiter mit dem Einlesen der Werte
30
31   einlese_schleife: do
32     read (25,*,iostat=status_read) wert ! Einlesen des Wertes
33     if ( status_read /= 0 ) exit        ! Programm wird am Ende
34                                         ! der einlese_schleife fortgesetzt,
35                                         ! wenn beim Einlesen des Wertes
36                                         ! ein Fehler auftritt oder das
37                                         ! Dateiende erreicht wurde
38     anzahl = anzahl + 1                 ! Anzahl der eingelesenen Werte
39                                         ! hier gleich Zeilennummer
40     !Bildschirmausgabe
41     write(*,*) ' Zeilennummer = ', anzahl, ' Wert = ', wert
42   end do einlese_schleife
43
44 ! hier geht's weiter, wenn status_read /= 0 war, deshalb:
45 ! Behandlung der 2 moeglichen Faelle mit status_read <> 0
46   readif: if ( status_read > 0 ) then
47   write(*,*) 'Beim Lesen von Zeile ', anzahl+1, &
48   ' ist ein Fehler aufgetreten'
49   else ! status_read < 0
50        ! das Dateiende (end of file = EOF) wurde erreicht
51        ! der Benutzer wird darueber explizit informiert und
52        ! die Gesamtanzahl der Datensaetze
53        ! wird nochmals ausgegeben
54
55     write(*,*)
56     write(*,*) ' Hinweis: das Dateiende wurde erreicht'
57     write(*,*) ' => In der Datei sind insgesamt ', &
58     anzahl, ' Werte'
59   end if readif
60
61 ! die noch ausstehende Behandlung des Fehlerfalls
62 ! beim OEffnen der Datei
63 ! status_open <> 0 wird hier behandelt
64
65 else oeffnen
66   write(*,*) 'Beim OEffnen der Datei trat &
67 &  Systemfehler Nr. ', status_open,' auf'
68 end if oeffnen
69
70 close( unit=25 )  !Datei schliessen
71
```

```
72 end program einlesen
```

Beispieldateien zum Ausprobieren:
Bei `datei1.dat` mit folgendem Inhalt:

```
-12.0
30.0001
1.0
.15E-10
-3.141953
```

liefert das Programm `einlesen` folgende Bildschirmausgabe:

```
Bitte geben Sie den Namen der zu lesenden Datei
(nicht in Anfuehrungszeichen eingeschlossen) an: datei1.dat
Als Name der Datei wurde eingelesen: datei1.dat

Zeilennummer = 1 Wert = -12.00000
Zeilennummer = 2 Wert = 30.00010
Zeilennummer = 3 Wert = 1.000000
Zeilennummer = 4 Wert = 1.5000000E-11
Zeilennummer = 5 Wert = -3.141953

Hinweis: das Dateiende wurde erreicht
=> In der Datei sind insgesamt 5 Werte
```

Bei `datei2.dat` mit folgendem Inhalt:

```
-12.0
30.0001
abc
.15E-10
-3.141953
```

liefert das Programm `einlesen` folgende Bildschirmausgabe:

```
Bitte geben Sie den Namen der zu lesenden Datei
(nicht in Anfuehrungszeichen eingeschlossen) an: datei2.dat
Als Name der Datei wurde eingelesen: datei2.dat

Zeilennummer = 1 Wert = -12.00000
Zeilennummer = 2 Wert = 30.00010
Beim Lesen von Zeile 3 ist ein Fehler aufgetreten
```

Bei Eingabe des Namens einer nicht vorhandenen Datei liefert das Programm `einlesen` folgende Bildschirmausgabe:

```
Bitte geben Sie den Namen der zu lesenden Datei
(nicht in Anfuehrungszeichen eingeschlossen) an: datei3.erg
Als Name der Datei wurde eingelesen: datei3.erg

Beim OEffnen der Datei trat Systemfehler Nr. 29 auf
```

7.6 Positionierung innerhalb einer geöffneten Datei

Falls mit `open` eine Verknüpfung zwischen einer *unit number* und einer Datei hergestellt wurde, kann man mit

```
backspace(unit=< unit number >)
```

einen Datensatz (d.h. eine Zeile) „zurückgespult “ werden.
An den Dateianfang gelangt man mit

```
rewind(unit=< unit number >)
```

Die beiden obigen Kommandos werden in der Regel nur in Zusammenhang mit `scratch`-Dateien benötigt.

7.7 Anhängen von Werten an eine bereits bestehende Datei

Ergänzt man die `open`-Anweisung durch den Zusatz `position = 'append'` kann an eine bereits bestehende Datei weitere Daten angehängt werden, z.B. könnte eine `open`-Anweisung zum Änhängen an die bereits bestehende Datei `'wertetabelle.txt'` lauten (alles in einer Befehlszeile):

```
open(unit=< unit number >, file='wertetabelle.txt', status='old',
action='write',position='append',iostat=io_err)
```

7.8 Die `close`-Anweisung

Die `close`-Anweisung wurde bereits in jedem der obigen Programmbeispiele verwendet.

```
close(unit=< unit number >)
```

oder kompakter

```
close(< unit number >)
```

dient dazu, um die vorher durch die entsprechende `open`-Anweisung zwischen einer *unit number* und einer Datei auf der Festplatte geschaffene Zuordnung wieder aufzuheben. Bei Erreichen des Programmendes (`end program ...`) würde ohnehin die Zuordnungen aus `open` wieder aufgehoben. Jedoch sollte man jede Zuordnung zwischen *unit number* und der Datei wieder aufheben, sobald diese nicht mehr benötigt wird Einerseits aus Gründen der Sorgfalt und Übersichtlichkeit, so dass weniger Fehler passieren können, andererseits hat jedes System hat einen Maximalwert an gleichzeitig geöffneten Dateien. Durch Schließen nicht mehr benötigter Dateien, kann man vermeiden, die Anzahl der maximal gleichzeitig geöffneten Dateien zu überschreiten.
Eine Modifikation der `close`-Anweisung erlaubt, Dateien mit dem „Schließen “ gleichzeitig zu löschen. Dazu wird wird `status='delete'` eingefügt.

```
close(unit=< unit number >,status='delete')
```

Um mögliche Fehler bei `close`-Anweisungen abfangen zu können, gibt es analog zu `open` bei `close` die Methodik mit `iostat`. Wird `status='delete'` nicht angegeben, geht der Compiler davon aus, dass `status='keep'` gilt und die Datei bleibt natürlich erhalten.
Achtung: Erst wenn eine mit `open` geöffnete Datei mit `close` oder durch Erreichen des Programmendes wieder geschlossen wurde, sollte von anderen Prozessen des Betriebssystems aus (z.B. mit einem Editor) auf diese Datei zugegriffen werden.

7.9 Die `inquire`-Anweisung

Mit `inquire` lassen sich z.B. Informationen über den Zustand einer Datei gewinnen, bevor diese geöffnet wird. Will man z.B. feststellen, ob eine Datei des Namens `wertetabelle.txt` vorhanden ist, so geht dies z.B. mit

```
inquire(file='wertetabelle.txt',exist=vorhanden)
```

Die Variable `vorhanden` muss dazu vorher als vom Datentyp `logical` deklariert worden sein. Existiert die Datei im aktuellen Verzeichnis, so ist der Wert von `vorhanden .true.`, falls die Datei nicht vorhanden wäre, so würde `vorhanden` der Wert `.false.` zugewiesen. Anhand des Wertes der Variable `vorhanden` lässt sich bei Bedarf das Programm sinnvoll verzweigen. Soll z.B. verhindert werden, dass eine bereits vorhandene Datei überschrieben wird, so kann der Anwender gefragt werden, ob

- er/sie einen anderen Dateinamen eingeben möchte
- er/sie die Daten an die bereits bestehende Datei anhängen möchte
  (dazu könnte der Befehl `open` wird mit dem Zusatz `position='append'` ausgeführt werden)
- er/sie die bereits vorhandene Datei wirklich überschreiben lassen möchte
  (in diesem Fall kann der Befehl `open` mit der Option `status='replace'` ausgeführt werden)

Kapitel 8

Formatbeschreiber

Nicht immer genügt das listengesteuerte Ausgabeformat den Anforderungen des Anwenders nach einem angemessenen Ausgabeformat.

```
write(*,*) <Variablenliste>
```

Das Sternchen an der 2. Stelle steht für die Formatangabe „listengesteuert“ bzw. „systembestimmt“ der nachfolgenden Variablenwerte. Zum Beispiel sollen Geldbeträge in Euro und Cent in der Regel mit 2 Nachkommastellen dargestellt werden. So soll z.B. eine Preisangabe von 9.98 Euro auf dem Bildschirm als `9.98` und nicht als `9.980000` erscheinen. Beispielprogramm:

```
1  program euro
2
3  implicit none
4  real :: preis = 9.98
5
6  write(*,*) 'listengesteuertes␣Ausgabeformat:'
7  write(*,*) preis
8  write(*,*)
9  write(*,*) 'formatgesteuertes␣Ausgabeformat:'
10 write(*,'(F8.2)') preis
11
12 end program euro
```

Das Programm liefert folgende Bildschirmausgabe:

```
listengesteuertes Ausgabeformat:
9.980000

formatgesteuertes Ausgabeformat:
9.98
```

Bei der formatgesteuerten Ausgabe wurde das zweite Sternchen durch die Formatangabe `'(F8.2)'` ersetzt. Dadurch wird der Wert der `real`-Variablen `preis` mit zwei Nachkommastellen in einem insgesamt 8 Stellen breiten Feld rechtsbündig ausgegeben. Vor der Zahl `9.98` befinden sich somit noch 4 Leerstellen.
Durch die Formatbeschreiber lassen sich in Fortran die Ausgabeformate den Erfordernissen des Anwenders anpassen.

8.1 Drei Möglichkeiten, Formate festzulegen

Es sei vereinbart:

integer :: i = 123456
real :: x = 3.141593

Möglichkeit 1: direkt die Formatbeschreiberkette angeben

```
write(*,'(1X,I6,F10.2)') i, x
```

Möglichkeit 2: die „Anweisungsnummer“ (engl. *statement label*) einer zugehörigen `format`-Anweisung angeben

```
write(*,100) i, x
100 format(1X,I6,F10.2)
```

Möglichkeit 3: den Namen einer Zeichenkette, die den Formatbeschreiber enthält, als Formatangabe eintragen

```
character(len=16) :: string

string = '(1X,I6,F10.2)'
write(*,string) i, x
```

Beispielprogramm:

```
1  program format_beispiel
2
3  implicit none
4  integer   :: i = 123456
5  real      :: x = 3.141592
6  character(len=16) :: string = '(1X,I6,F10.2)'   ! Formatangabe
7
8  ! Vorstellung der 3 Moeglichkeiten, Formatbeschreiber anzugeben
9
10 ! 1. Moeglichkeit: direkte Angabe der Formatbeschreiberkette
11 write(*,'(1X,I6,F10.2)') i, x
12
13 ! 2. Moeglichkeit: Angabe einer Anweisungsnummer, die die zugehoerige
14 !                  Format-Anweisung enthaelt
15 write(*,100) i, x
16 100    format(1X,I6,F10.2)
17
18 ! 3. Moeglichkeit: Angabe einer Zeichenkette, die die
19 !                   Formatbeschreiberkette enthaelt
20 write(*,string) i, x
21
22 end program format_beispiel
```

Das Programm liefert folgende Bildschirmausgabe:

```
123456 3.14
123456 3.14
123456 3.14
```

Jede der 3 angegebenen Möglichkeiten, nacheinander den Wert von `i` und `x` auszugeben, ist in der Wirkung gleich. Die Formatangabe `'(1X,I6,F10.2')` bewirkt in der `write`-Anweisung, dass bei der Ausgabe der Werte von `i` und `x` zunächst eine Leerstelle ausgegeben wird `(1X)`, die folgenden 6 Felder Breite werden zur Ausgabe des Wertes von `i` verwendet. Da die Zahl 123456 genau 6 Stellen breit ist, werden diese 6 Felder vollständig von der Zahl ausgefüllt. Die folgenden 10 Felder sind zur Ausgabe der real|-Zahl, der der Variablen `x` zugewiesen wurde, vorgesehen. Die Ausgabefelder werden prinzipiell von rechts her aufgefüllt. `F` ist einer der Möglichkeiten, Werte vom Datentyp `real` formatiert ausgeben zu lassen. Durch `F` wird festgelegt, dass es sich um eine Festpunktzahl handeln soll, deren Anzahl der Nachkommastellen im obigen Beispiel durch die Zahl hinter dem . (Punkt) im Formatbeschreiber, und somit auf 2 Nachkommastellen festgelegt wurde. An der 3. Stelle von rechts seht dann der Dezimalpunkt, an der 4. Stelle steht dann die 3. Deshalb bleiben zwischen der letzten Ziffer ausgegebenen Ziffer von `i` und der ersten ausgegebenen Ziffer von `x` 6 Leerzeichen (Blanks) frei.

8.2 Allgemeine Bemerkungen zu den Möglichkeiten der Formatangabe

Möglichkeit 1 (die direkte Angabe der Formatbescheiberkette) ist sinnvoll, wenn die Formatangabe nur einmal benötigt wird.
Möglichkeit 2 bietet sich an, wenn derselbe Formatbeschreiber an mehreren Stellen der gleichen Programmeinheit Verwendung finden kann. `format`-Anweisungen können an beliebiger Stelle im Anweisungsteil einer Programmeinheit stehen. Es bietet sich an, sie an exponierter Stelle (z.B. unterhalb der Variablendeklarationen) einzufügen, so dass sie leicht angepasst werden können. Bei der `format`-Anweisung handelt es sich um eine nicht ausführbare Anweisung. An das entsprechende *statement label* kann somit nicht mittels `goto` gesprungen werden.
Möglichkeit 3 bietet den Vorteil, sich mittels Zeichenkettenverarbeitung den Formatbeschreiber innerhalb des Programms den Erfordernissen des Ausgabeformats geeignet anzupassen. (z.B. die universelle Ausgabe von n x n Matrizen, wobei n zwischen 2 und 10 liegen kann) in einem gut lesbaren Format.

8.3 Generelles zu den Formatbeschreiberketten

Die einzelnen Formatangaben in der Formatbeschreiberkette werden in der Regel durch Kommata voneinander getrennt.
Ausnahme: bei / für den Zeilenvorschub (*new line*) ist dies nicht notwendig, ebensowenig bei $ zur Unterdrückung des Zeilenvorschubs am Ende einer `write`-Anweisung.
Formatbeschreiberketten lassen sich in ein Klammerpaar einschliessen und vor dieses lässt sich wiederum eine Zahl als Wiederholungfaktor stellen. Zum Beispiel wären

```
10 format(3(1X,F12.2))
```

und

```
20 format(1X,F12.2,1X,F12.2,1X,F12.2)
```

in ihrer Wirkung identisch.
Es ist auch möglich, in die Formatangabe Textbestandteile einzubauen. Soll z.B. eine Zeilennummer und ein Wert hintereinander mit einem Kommentar ausgeben werden, so könnte man z.B. schreiben

```
write(*,30) i, x
30 format(1X,'Zeilennummer =',I7,',',2X,'Wert =',F12.6)
```

ergibt als Output

```
Zeilennummer = 123456,  Wert =    3.141593
```

8.4 Allgemeine Bemerkungen zum Umgang mit Formatbeschreibern

Falls Sie Formatbeschreiber einsetzen, bitte stets auf die Übereinstimmung von Formatbeschreiber und Datentyp der Variaben achten! Ein falscher Formatbeschreiber kann unter Umständen zu *run time errors* (Laufzeitfehlern) führen oder zur Ausgabe einer Sequenz an Sternchen.
Durch die Angabe eines Formatbeschreiber kann man die Genauigkeit in der Zahlendarstellung für den Datentyp `real` reduzieren, z.B. werden mit `F11.3` nur noch 3 signifikante Nachkommastellen und mit `E11.3` nur noch insgesamt 3 signifikante Stellen einer Zahl dargestellt. **Dabei rundet Fortran mathematisch korrekt auf oder ab.**
Generell ist beim Umgang mit Formatbeschreibern darauf zu achten, dass es niemals ungewollt zu einem Verlust an Genauigkeit kommen darf. Sollen z.B. numerisch gewonnnene Daten auf Dateien geschrieben und später weiterverarbeitet werden, so ist z.B. fast immer die Ausgabe im listengesteuerten Format sinnvoll, weil dadurch die maximale Darstellungsgenauigkeit der Zahl erhalten bleibt.
Im Einzelfall kann es allerdings auch Sinn machen, die Darstellungsgenauigkeit der Zahlen bei der Abspeicherung anzupassen, beispielsweise, wenn es sich um Messwerte mit einer vorgegeben Genauigkeit handelt. Liegen z.B. Messwerte mit 3 Stellen Genauigkeit vor, so würde es keinen Sinn machen, die Zahlen listengesteuert auf 7 Stellen genau abzuspeichern. In diesem Fall würde nur unnötig Speicherplatz verbraucht werden.

8.5 Formatbeschreiber für den Datentyp `integer`

Dieser kann z.B. lauten `Iw` oder `rIw` oder `rIw.m`.
Hier und im folgenden bedeuten

- w = Feldbreite
- r = Wiederholungsfaktor
- m = Mindestanzahl der darzustellenden Zeichen

Im Beispiel oben wurde vereinbart

```
integer   :: i = 123456
```

Damit hat i einen sechstelligen Wert erhalten. Mit dem Formatbeschreiber '(I6)' wird vereinbart, dass für die Ausgabe von i 6 Stellen vorgesehen werden. In diesem Beispiel stimmen Feldbreite und Anzahl der Ziffern überein. Mit der Anweisung

```
write(*,'(I6)') i
```

wird der Wert von i auf dem Bildschirm rechtsbündig in ein Feld mit 6 Stellen Breite geschrieben als

```
123456
```

Die Ausgabe beginnt also direkt am linken Rand. Durch die Anweisung

```
write(*,'(I8)') i
```

wird die Feldbreite auf 8 Stellen erhöht. Der Wert von i (im Beispiel ist dies 123456) wird rechtsbündig in das 8 Stellen breite Feld eingefügt und damit beginnt die Bildschirmausgabe mit 2 Leerstellen

```
  123456
```

Die Zahlen werden rechtsbündig in die angegebene Ausgabefeldbreite eingefügt. Solange w grösser ist als die Anzahl der Ziffern l, werden w-l Leerzeichen der Zahl vorangestellt. Müssen mehr Ziffern ausgegeben werden, als der Wert von w angibt, passt die Zahl nicht mehr in die dafür vorgesehene Feldbreite. In diesem Fall werden w Sternchen „gedruckt“. Konkret: würde man im obigen Beispiel

```
write(*,'(I5)') i
```

verlangen, dass die sechstellige Ziffer 1234567 in ein Feld der Breite 5 ausgeben wird, so kann dies nicht funktionieren. Die Fortran-Compiler verhalten sich nach dem Standard so, dass in diesem Fall 5 (im allgemeinen Fall w) Sternchen ausgeben werden

```
*****
```

so dass, wenn Werte nicht in das programmierte Ausgabeformat passen, der Wert der Feldbreite w als Sternchen auf dem Bildschirm erscheinen. In diesem Fall ist der Programmierer aufgefordert, diesen Fehler im Ausgabeformat zu beheben. Beispielprogramm:

```
1  program format_integer
2  ! Beispiele zum Formatbeschreiber fuer den Datentyp integer
3
4  !      Iw  oder rIw oder rIw.m oder Iw.m
5
6  !      r : Wiederholungsfaktor
7  !      w : Feldgroesse
8  !      m : Mindestfeldbreite
9
10
11 implicit none
```

```
integer :: i = 123, j = -123 , k = 123456 , l = -123456, m = 12345678

write(*,*) 'definiert wurde:'
write(*,*) &
'integer ::i =123,j= -123,k= 123456,l= -123456,m= 12345678 '
write(*,*) 'i =123'
write(*,*) 'Listengesteuerte Ausgabe einzeln:'
write(*,*) i
write(*,*) j
write(*,*) k
write(*,*) l
write(*,*) m
write(*,*)
write(*,*) 'Listengesteuerte Ausgabe der Variablenliste'
write(*,*) i, j, k, l, m
write(*,*)
write(*,*) 'Formatgesteuerte Ausgabe ''(5I9)'':'
write(*,'(5I9)') i, j, k, l, m
write(*,*)
write(*,*) 'Formatgesteuerte Ausgabe ''(4I10)'' und 5 Werte:'
write(*,'(4I10)') i, j, k, l, m
write(*,*)
write(*,*) 'Formatgesteuerte Ausgabe ''(I5)'' einzeln:'
write(*,*) 'Achtung: zu kleine Feldbreite bei k,l,m'
write(*,'(I5)') i
write(*,'(I5)') j
write(*,'(I5)') k
write(*,'(I5)') l
write(*,'(I5)') m

end program format_integer
```

Wird eine Mindestbreite `m` im `integer`-Formatbeschreiber vorgeben, und sei die Zahl insgesamt `l` Ziffern lang, so werden insgesamt `m-l` Nullen der Zahl vorangestellt. Dies könnte man nutzen wenn man eine Art Zähler programmieren will.

```
write(*,'(I8.7)') i
```

bedeutet, dass der Wert von `i` rechtbündig in ein Feld der Breite 8 und mit mindestens 7 Stellen ausgeben werden soll. Da im obigen Beispiel `i` nur 6 Stellen besitzt (`l=6`) werden der Zahl `m-l = 7-6 = 1` Null vorangestellt. Vor der ausgegebenen Zahl befindet sich aufgrund der gewählten Beispielzahlen natürlich noch eine Leerstelle.

```
0123456
```

Beispielprogramm:

```
program format_integer_feldbreite

implicit none
integer :: i = 123

write(*,'(I5.4)') i

end program format_integer_feldbreite
```

8.6 Binär-, Octal- und Hexadezimaldarstellung von Zahlen des Datentyps integer

Zahlen vom Datentyp integer lassen sich auch ausgeben mit dem Formatbeschreiber

- Bw
- Ow
- Zw

Dabei entspricht

- B = Binärdarstellung (Zahlensystem-Basis 2)
- O = Octaldarstellung (Zahlensystem-Basis 8)
- Z = Hexadezimaldarstellung (Zahlensystem-Basis 16)

- w = Feldbreite

Beispielprogramm:

```
1  program formate_integer
2
3  implicit none
4  integer :: i = 73
5
6  write(*,*) 'Umgewandelt wird:'
7  write(*,'(I12)') i
8  write(*,*)
9
10 write(*,*) 'In eine Binaerzahl (Zahlen-Basis 2):'
11 ! Umwandlung in eine Binaerzahl (Basis 2)
12 write(*,'(B12)') i
13
14 write(*,*)
15 write(*,*) 'In eine Octal (Zahlen-Basis 8):'
16 ! Umwandlung in eine Octalzahl (Basis 8)
17 write(*,'(O12)') i
18
19 write(*,*)
20 write(*,*) 'In eine Hexadezimalzahl(Zahlen-Basis 16):'
21 ! Umwandlung in eine Hexadezimalzahl (Basis 16)
22 write(*,'(Z12)') i
23
24 end program formate_integer
```

Formatbeschreiber		Bedingung
`Fw.d`	Fixpunktdarstellung	`w>=d+2`
`Ew.d`	wissenschaftliche Darstellung	`w>=d+7` vor dem Dezimalpunkt steht bei den meisten Compilern als Ziffer eine 0, manchmal auch nichts
`ESw.d`	„echte“ wissenschaftliche Darstellung (Fortran90/95 Erweiterung)	`w>=d+7` vor dem Dezimalpunkt steht eine Ziffer zwischen 1 und 9
`ENw.d`	„Ingenieur-Darstellung“(Fortran90/95 Erweiterung)	`w>=d+9` die Zahl vor dem Dezimalpunkt liegt zwischen 1 und 999 bzw. -1 und -999 und der Exponent zur Basis 10 wird stets als ein Vielfaches von 3 bzw. -3 dargestellt wie es den üblichen Benennungen als Kilo, Mega, Giga ... bzw. Milli, Mikro, Nano... entspricht
`Gw.d`	„Gleitpunktzahl“ je nach Größe der Zahl F oder E-Format	`w>=d+7` falls sich die Zahl als Fixpunktzahl (F-Format) darstellen lässt, wird dieses verwendet, wenn nicht, wird die E-Darstellung eingesetzt)

Tabelle 8.1: Bedingungen des Formatbeschreibers für den Datentyp `real`

8.7 Formatbeschreiber für den Datentyp `real`

Für die formatierte Ausgabe einer Zahl des Datentyps real kann man wählen zwischen

- `Fw.d` bzw. `rFw.d` (Fixpunktzahl)
- `Ew.d` bzw. `rEw.d` (Darstellung mit Exponenten zur Basis 10)
- `ESw.d` bzw. `rESw.d` („echte“ wissenschaftliche Darstellung mit Exponenten zur Basis 10, Fortran 90/95)
- `ENw.d` bzw. `rENw.d` („Ingenieur-Darstellung“, der Exponent zur Basis 10 ist ein Vielfaches von 3, Fortran 90/95)
- `Gw.d` bzw. `rGw.d` (Gleitpunktzahl)

- `w` = Feldbreite
- `d` = Anzahl der Nachkommastellen
- `r` = Wiederholungsfaktor

Dabei gelten die in Tabelle 8.1 gezeigten Bedingungen zwischen `w` und `d`. Beispielprogramm:

```
! Beispiele Formatbeschreiber REAL
!
! a) rFw.d   Fixpunktdarstellung
! b) rEw.d   wissenschaftliche Darstellung
! c) rESw.d  "echte" wissenschaftliche Darstellung (Fortran90/95 Erweiterung)
! d) rENw.d  "Ingenieur-Darstellung" (Fortran90/95 Erweiterung)
! e) rGw.d   "Gleitpunktzahl" je nachdem F oder E-Format
!
! r : Wiederholungsfaktor
! w : Feldgroesse
! d : Nachkommastellen: dabei muss gelten:
!
! a) w >= d+2
! b) w >= d+7 falls der Compiler die Null vor
!             dem Punkt schreibt. Falls der Compiler
!             die Ziffer 0 nicht ausschreibt, reicht w >= d+6
! c) w >= d+7 aehnlich b), die Ziffer vor dem Punkt
!             liegt zwischen -9 und 9
! d) w >= d+9 aehnlich b) und c), die Zahl vor dem Punkt liegt zwischen
!             1 und 999 bzw. -1 und -999 und die Exponent zur Basis 10
!             wird stets als ein Vielfaches von 3 bzw. -3 dargestellt
!             wie es den ueblichen Benennungen als Kilo, Mega, Giga ...
!             bzw. Milli, Mikro, Nano... entspricht
!
! e) w >= d+7 Falls sich die Zahl als Fixpunktzahl (F-Format)
!             darstellen laesst, wird dieses verwendet, wenn nicht,
!             wird die E - Darstellung eingesetzt)
!
program format_real

implicit none
real :: x = 1.234567, y = 2.222222E5, z = -3.3333E-5

write(*,*) 'Listengesteuerte Ausgabe einzeln:'
write(*,*) x
write(*,*) y
write(*,*) z
write(*,*)

write(*,*) 'Formatgesteuerte Ausgabe ''(F13.4)'':'
100 format(1X,F13.4)
write(*,100) x
write(*,100) y
write(*,100) z
write(*,*)

write(*,*) 'Formatgesteuerte Ausgabe ''(E13.4)'':'
200 format(1X,E13.4)
write(*,200) x
write(*,200) y
write(*,200) z
write(*,*)

write(*,*) 'Formatgesteuerte Ausgabe ''(ES13.4)'':'
300 format(1X,ES13.4)
```

```
56 write(*,300) x
57 write(*,300) y
58 write(*,300) z
59 write(*,*)
60
61 write(*,*) 'Formatgesteuerte␣Ausgabe␣''(EN13.4)'':'
62 400 format(1X,EN13.4)
63 write(*,400) x
64 write(*,400) y
65 write(*,400) z
66 write(*,*)
67
68 write(*,*) 'Formatgesteuerte␣Ausgabe␣''(G13.4)'':'
69 500 format(1X,G13.4)
70 write(*,500) x
71 write(*,500) y
72 write(*,500) z
73
74 end program format_real
```

8.8 Formatbeschreiber für den Datentyp `logical`

Der Formatbeschreiber zur Ausgabe von Werten des logischen Datentyps lautet

- `Lw` bzw. `rLw`

- `w` = Feldbreite
- `r` = Wiederholungsfaktor

Bezüglich der Anwendung und der Wirkung betrachte man das folgende Beispielprogramm:

```
1  ! Beispiele zum Formatbeschreiber fuer den Datentyp logical
2  !      Lw  oder rLw
3  !      r : Wiederholungsfaktor
4  !      w : Feldgroesse
5
6  program format_logical
7
8  implicit none
9  logical :: wahr = .TRUE., falsch = .FALSE.
10
11 write(*,*) 'Listengesteuerte␣Ausgabe:'
12 write(*,*) wahr, falsch
13 write(*,*)
14 write(*,*) 'Formatgesteuerte␣Ausgabe␣''(2(1X,L5))'':'
15 write(*,'(2(1X,L5))') wahr, falsch
16
17 end program format_logical
```

8.9 Formatbeschreiber für den Datentyp character

Für Zeichenketten existieren als mögliche Formatbeschreiber A bzw. rA und Aw bzw. rAw, die sich ihrer Wirkung nach unterscheiden.

- A bzw. rA oder
- Aw bzw. rAw

- w = Feldbreite
- r = Wiederholungsfaktor

Verwendet man bei der Ausgabe einer Zeichenkette der Länge l als Formatangabe A, so wird die Zeichenkette in ihrer generischen Feldbreite l ausgegeben.
Schreibt man hingegen als Formatangabe Aw, so sind die Fälle w>=l und w<l zu unterscheiden. Ist die Zeichenkettenlänge kleiner als als die zur Ausgabe vorgesehene Feldbreite, so werden w-l Leerzeichen (Blanks) der Zeichenkette vorangestellt. Das Ausgabefeld wird von rechts her aufgefüllt. Passt die Zeichenkette allerdings nicht in die Ausgabefeldbreite hinein, so werden nur die ersten l Zeichen der Zeichenkette ausgegeben.
Zusatzbemerkung: Wurde zur Ausgabe einer Zeichenkette das listengesteuerte Ausgabeformat verwendet, wird bei den meisten Compilern zunächst ein Leerzeichen und dann erst die Zeichenkette ausgegeben.
Beispielprogramm:

```
!
! Beispielprogramm zur formatierten Ausgabe von Zeichenketten
!

program format_character

implicit none
character(5) ::  zeichenkette_1 = 'abcde'
character(10) :: zeichenkette_2 = '1234567890'

write(*,*) 'Listengesteuerte␣Ausgabe␣einzeln:'
write(*,*)  zeichenkette_1
write(*,*)  zeichenkette_2
write(*,*)

write(*,*) 'Listengesteuerte␣Ausgabe␣hintereinander:'
write(*,*)  zeichenkette_1, zeichenkette_2
write(*,*)

write(*,*) 'formatgesteuerte␣Ausgabe␣''(2A)'':'
write(*,'(2A)')  zeichenkette_1, zeichenkette_2
write(*,*)

write(*,*) 'formatgesteuerte␣Ausgabe␣''(1X,A,1X,A)'':'
write(*,'(1X,A,1X,A)')  zeichenkette_1, zeichenkette_2
write(*,*)

```

```
28 write(*,*) 'formatgesteuerte␣Ausgabe␣''(1X,A10,1X,A10)'':'
29 write(*,'(1X,A10,1X,A10)')  zeichenkette_1, zeichenkette_2
30 write(*,*)
31
32 write(*,*) 'formatgesteuerte␣Ausgabe␣''(1X,A5,1X,A5)'':'
33 write(*,'(1X,A5,1X,A5)')  zeichenkette_1, zeichenkette_2
34 write(*,*)
35
36 write(*,*) 'Zeilenumbruch␣im␣Formatbeschreiber:'
37 write(*,*) 'formatgesteuerte␣Ausgabe␣''(1X,A,/,A)'':'
38 write(*,'(1X,A,/,A)')  zeichenkette_1, zeichenkette_2
39 write(*,*)
40
41 write(*,*) 'Setzen␣von␣Tabulatoren:'
42 write(*,*) 'formatgesteuerte␣Ausgabe␣''(T1,A,T12,A)'':'
43 write(*,'(T1,A,T12,A)')  zeichenkette_1, zeichenkette_2
44 write(*,*)
45
46 end program format_character
```

8.10 Formatbeschreiber zur Positionierung

Bei `X` und `nX` in der Formatangabe werden 1 bzw. `n` Leerzeichen bei der Ausgabe eingefügt. Will man eine Ausgabe mit Hilfe eines Formatbeschreibers an eine bestimmte Spalte setzen (und so eine Art Tabulatorfunktion nutzen), kann man dies ab Fortran 90/95 mit `Tc` tun. Die Zahl `c` gibt die Spaltennummer an. Einen Zeilenvorschub kann man mit `/` einbauen. Dementsprechend führt `//` in einer Formatbeschreiberkette dazu, dass eine Leerzeile ausgeben wird. Will man den gegenteiligen Effekt erreichen und am Ende einer `write`-Anweisung den Zeilenvorschub unterdrücken, kann man an das Ende einer Formatbeschreiberkette `$` anhängen. Während die einzelnen Formatbeschreiber in einer Formatbeschreiberkette durch Kommata getrennt werden müssen, kann man bei der Verwendung von `/` und `$` diese auch weglassen.

8.11 Formatgesteuertes Einlesen von Werten

Genauso wie in den `write`-Anweisungen lassen sich auch beim `read`-Befehl Formatbeschreiber angeben. In der Regel ist das listengesteuerte Einlesen von Werten, d.h. die `read`- Anweisung mit der Angabe von `*` als Formatangabe eine gute Wahl. Während sich das listengesteuerte Einlesen von Werten vom Datentyp `real` mit `read(*,*)` extrem gutmütig verhält und alle Formate von `real`-Werten annimmt (sei es nun als Fixpunktzahl oder in der wissenschaftlichen Notation), birgt die Vorgabe eines festen Formatbeschreibers in diesem Fall die Gefahr, dass z.B. mit einem vorgebenen Formatbeschreiber für eine Fixpunktzahl der Anwender den Wert im wissenschaftlicher Notation angibt und dann Fehler auftreten. Haben Sie allerdings formatierte Wertetabellen mit Zahlen auf Dateien geschrieben, könnte es unter Umständen sinnvoll sein, wieder formatgesteuert einzulesen. Hier ist jedoch unbedingt auf eine exakte Gleichheit der Formatbeschreiber beim Wertetabellen-Schreiben mit denen beim Wertetabellen-Lesen zu achten, weil sonst Fehler auftreten könnten.

Formatiert geschriebene Wertetabellen mit Werten des Datentyps `real` können jedoch in Fortran immer listengesteuert eingelesen werden. Das interaktive formatgesteuerte Einlesen von Werten ist meines Erachtens nur für Zeichenketten (engl. *strings*) oder logische Werte sinnvoll.
Hinweis: Insbesondere Zeichenketten, die Leerzeichen enthalten können, sollten formatiert eingelesen werden, sonst würde das 1. Leerzeichen oder Komma als Ende-Markierung der Zeichenkette angesehen werden und es würde nur der Anfangsteil der Zeichenkette als Variablenwert zugewiesen werden. Man liest also Zeichenketten am besten mit einer Formatangabe ein, z.B.

```
read(*,'(A)') zeichenkette
```

Beispielprogramm:

```
1  program zeichenkette_einlesen
2
3  implicit none
4  character(len=30),parameter :: messlatte='....|....1....|....2....|....3'
5  character(len=30) :: zeichenkette1, zeichenkette2
6
7  write(*,*) 'Geben␣Sie␣eine␣Zeichenkette␣mit␣max.␣30␣Zeichen␣ein:'
8  write(*,*) 'zum␣listengesteuerten␣Einlesen'
9  read(*,*) zeichenkette1
10 write(*,*)
11
12 write(*,*) 'Geben␣Sie␣erneut␣eine␣Zeichenkette␣mit␣max.␣30␣Zeichen␣ein:'
13 write(*,*) 'zum␣formatgesteuerten␣Einlesen'
14 read(*,'(A)') zeichenkette2
15 write(*,*)
16 write(*,*) 'die␣beiden␣Zeichenketten␣werden␣nun␣wieder␣ausgegeben'
17 write(*,*)
18
19 write(*,*) 'die␣listengesteuert␣eingelesene␣Zeichenkette:'
20 write(*,'(A)') messlatte
21 write(*,'(A)') zeichenkette1
22 write(*,*)
23 write(*,*) 'die␣formatgesteuert␣eingelesene␣Zeichenkette:'
24 write(*,'(A)') messlatte
25 write(*,'(A)') zeichenkette2
26
27 end program zeichenkette_einlesen
```

Das Programm liefert folgende Bildschirmausgabe:

```
Geben Sie eine Zeichenkette mit max. 30 Zeichen ein:
zum listengesteuerten Einlesen
abcdefg,45 hdf

Geben Sie erneut eine Zeichenkette mit max. 30 Zeichen ein:
zum formatgesteuerten Einlesen
abcdefg,45 hdf

die beiden Zeichenketten werden nun wieder ausgegeben
```

```
die listengesteuert eingelesene Zeichenkette:
....|....1....|....2....|....3
abcdefg

die formatgesteuert eingelesene Zeichenkette:
....|....1....|....2....|....3
abcdefg,45 hdf
```

Selten dürfte sein, dass logische Werte eingelesen werden sollen. Hier macht die Format-Angabe für logische Werte (`L`) insofern Sinn, als dass dadurch vom Anwender explizit ein logischer Wert bei der Eingabe mit `T` oder `F` angegeben werden muss und mit Hilfe von `iostat` in der read|-Anweisung falsche Eingabewerte leichter erkannt und abgefangen werden können. Das unterschiedliche Verhalten beim unformatierten und beim formatierten Lesen logischer Werte - insbesondere wenn man statt `T` oder `F` z.B. `0` oder `B` angibt - kann das Verhalten z.B. mit dem folgenden Programm `read_logical` getestet werden.
Beispielprogramm:

```
program read_logical

implicit none
logical :: l
integer :: status

endlosschleife: do              !Endlosschleife
 write(*,*) 'Einlesen␣eines␣logischen␣Werts␣(listengesteuert)'
 write(*,'(1X,A$)') 'Bitte␣geben␣Sie␣einen␣logischen␣Wert␣ein:␣'
 read(*,*) l
 write(*,*) 'eingelesen␣wurde␣:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣', l
 write(*,*)
 write(*,*) 'Einlesen␣eines␣logischen␣Werts␣''(L)'''
 do
  write(*,'(1X,A$)') 'Bitte␣geben␣Sie␣einen␣logischen␣Wert␣ein:␣'
  read(*,'(L7)',iostat=status) l
  if (status /= 0) then
   write(*,*) 'Bitte␣geben␣Sie␣T␣fuer␣wahr␣und␣F␣fuer␣falsch␣oder␣\&
␣␣␣.true.␣bzw.␣.false.␣ein!'
   cycle
  end if
  write(*,'(1X,A,17X,L1)') 'eingelesen␣wurde␣:␣', l
  write(*,*)
  exit
 end do
end do endlosschleife

end program read_logical
```

8.12 Umwandlung eines `real`-Wertes in eine Zeichenkette und umgekehrt (*internal files*)

In Fortran existiert ein spezieller Mechanismus als Erweiterung des bisher kennengelernten Verfahrens der Ein- und Ausgabe. Im Englischen wird dieser Mechanismus als *internal files* bezeichnet. Statt mit `write` auf eine externe Datei (oder den Bildschirm) zu schreiben, kann man damit stattdessen intern in den Speicherbereich einer Variablen schreiben, um so z.B. einen `real`-Wert in eine Zeichenkette (`character`) zu verwandeln. Der Mechanismus der *internal files* wird im folgenden Beispiel verwendet, um genau dieses zu tun. Zusätzlich wird im Beispiel das Format (der Formatbeschreiber) durch die Größe des zu konvertierenden Wertes bestimmt. Beispielprogramm:

```
! Umwandlung von Werten des Datentyps real in
! eine Zeichenkette (character)
! gemaess eines von der Zahl abhaengigen Formatbeschreibers

! Demonstration des Mechanismus der sogenannten "internal files"

program real_in_zeichenkette

implicit none
real :: x
character(len=9)  :: fb
character(len=12) :: zeichenkette

write(*,'(1X,A$)') 'Geben␣Sie␣bitte␣einen␣Wert␣vom␣Datentyp␣real␣ein:␣'
read(*,*) x
write(*,*) 'Ihre␣Eingabe:␣x␣=␣', x
write(*,*)

! Je nach Groesse der eingelesenen Zahl wird nun der Formatbeschreiber fb
! festgelegt. Die Feldbreite jedes Formats wurde entsprechend der Laenge
! des Strings Zeichenkette gewaehlt

if ( x > 9999999.0) then
 fb = '(ES12.5)'
else if ( x < -999999.0) then
 fb = '(ES12.5)'
else if ( x == 0) then
 fb = '(F12.4)'
else if ( abs(x) < 0.01 ) then
 fb = '(ES12.5)'
else
 fb = '(F12.4)'
end if

! Umwandlung von x aus dem Datentyp real in einen String

write(zeichenkette,fb) x

! Bildschirmausgabe der Zeichenkette

write(*,*) "Es␣wird␣nun␣'x_'//zeichenkette//'_x'␣ausgegeben:␣", \&
```

```
'x_'//zeichenkette//'_x'

end program real_in_zeichenkette
```

Genauso wie mit

```
write(<Variable_2>,<Formatangabe>) <Variable_1>
```

der Inhalt aus dem Speicherbereich von `Variable_1` unter Beachtung der Formatangabe in den Speicherbereich der `Variable_2` geschrieben werden kann, lässt sich die umgekehrte Operation mit `read` realisieren. Beispielprogramm:

```
program internal_file_read

implicit none
character(len=15) :: werte = '1.234␣2.4e-6'
real :: x, y

read(werte,*) x, y

write(*,*) 'intern␣wurden␣zugewiesen:␣'
write(*,*) 'x␣=␣', x
write(*,*) 'y␣=␣', y

end program internal_file_read
```

Das Programm liefert folgende Bildschirmausgabe:

```
intern wurden zugewiesen:
x = 1.234000
y = 2.4000001E-06
```

Zu beachten ist, dass als erstes Argument der *internal file*-`read`- bzw. `write`-Anweisung mit den Namen **einer Variablen** auf den Speicherplatz dieser Variablen referenziert wird und hier nur der Name **einer Variablen** angegeben werden kann, während als Argumente der Name **einer oder mehrerer Variablen** stehen können.

Kapitel 9

Datenfelder (engl. *arrays*) oder indizierte Variablen

Datenfelder (auch engl. *arrays* oder indizierte Variablen genannt) werden benötigt, wenn viele gleichartig strukturierte Daten mit einem Programm verarbeitet werden sollen. Arrays sind ebenso notwendig, falls der Gesamtumfang der Datensätze bei der Entwicklung des Programms noch nicht feststeht.
Denn in solchen Fällen wäre es äußerst umständlich, wenn man sich immer neue Variablen mit leicht modifizierten Namen definieren und im Programm einsetzen müsste. Viel einfacher und viel sinnvoller ist es, wenn man sich die einzelnen Datensätze in durchnummerierten Namen abspeichern und mit Hilfe eines Indizes auf den einzelnen Datensatz zugreifen kann. Zum Beispiel:

```
a(1), a(2), ...., a(100)
```

statt

```
a1, a2, ..., a100
```

Im ungünstigen 2. Fall müsste man 100 verschiedene Variablen deklarieren und auf die einzelne Variable durch Angabe des Namens zugreifen, während man im 1. Fall durch die Indizierung leicht über eine Schleife auf alle Komponenten des Datenfelds zugreifen kann. Will man z.B. von jeder Komponente des Datenfelds den natürlichen Logarithmus bilden, könnte man im 1. Fall schreiben

```
integer :: i

do i = 1, 100
a(i) = log( a(i) )
end do
```

Im ungünstigen 2. Fall müsste man die 100 Variablen einzeln aufführen

```
a1 = log(a1)
a2 = log(a2)
a3 = log(a3)
...
a100 = log(a100)
```

Schon anhand dieses Beispiels lassen sich die immensen Vorteile bei der Verwendung indizierter Variablen (Datenfelder) in Programmen erahnen. Als Weiterentwicklung von Fortran 77 bietet Fortran 90/95 erweiterte Zugriffsmöglichkeiten auf die Komponenten eines Datenfelds als Ganzes, z.B. Wertzuweisungen, Ausgaben, Additionen etc. die die Programmerstellung nochmals erleichtern. In Fortran 90/95 lässt sich sogar schreiben, nachdem a als Array vom Datentyp `real` mit 100 Komponenten deklariert wurde, um an allen Komponenten des Datenfeldes den natürlichen Logarithmus der Komponente im Datenfeld stehen zu haben

```
a = log(a)
```

9.1 Deklaration von Datenfeldern (statisch)

Im Deklarationsteil einer Programmeinheit muss bei einer statischen Deklaration für jedes Datenfeld der Datentyp, die Dimension und der Name festgelegt werden. Beispiel: einen Vektor `v` mit den 3 Komponenten `v(1)`, `v(2)` und `v(3)` deklarieren:

```
real, dimension(3) :: v
```

alternativ ließe sich noch die alte Fortran 77 - Syntax verwenden (hier sei dies nur der Vollständigkeit halber erwähnt, bitte nicht mehr in neuen Fortran 90/95-Programmen einsetzen!)

```
real :: v(3)
```

Beispielprogramm:

```
!
! Beispielprogramm zur Deklaration eines Datenfeldes a
! mit den Komponenten vom Datentyp real a(1), a(2), a(3), a(4)
!
program datenfeld_1

implicit none
real, dimension(4) :: a    ! Deklaration des Datenfeldes a
integer            :: i    ! integer-Variable i. Diese wird
                           ! verwendet, um auf die einzelnen i
                           ! Komponenten des Datenfeldes zuzugreifen
! FORTRAN 77 (90/95)
do i = 1, 4                ! mit dieser Schleife wird jede
 a(i) = real(i*i)          ! Komponente im Datenfeld auf den Wert des
end do                     ! Index**2 gesetzt

! Fortran 90/95
a = a - 1.5                ! von allen Komponenten von a
                           ! wird 1.5 subtrahiert

a(3) = 5.0                 ! die 3. Komponente des Datenfeldes wird
                           ! auf den Wert 5.0 gesetzt

do i = 1, 4                ! mittels einer Schleife werden
 write(*,*) a(i)           ! die aktuellen Inhalte an dem Speicherplatz
end do                     ! des Datenfeldes ausgegeben

end program datenfeld_1
```

9.2 Die implizite do-Schleife bei der Ein- und Ausgabe von Feldern (Fortran 77)

Durch eine implizite do-Schleife lassen sich die Elemente eines Datenfelds sukzessive in einer Zeile ausgeben. Dieses Verfahren entspricht dem alten Fortran 77 - Syntax. In Fortran 90/95 geht es aber noch viel einfacher, wie im nächsten Abschnitt gezeigt wird. Nach dem alten Fortran 77 - Syntax, kann zur Ausgabe alle Komponenten eines Feldes eine implizite do-Schleife einsetzen. Will man z.B. alle Elemente eines dreikomponentigen Vektors v in einer Zeile ausgeben, so kann man dies durch

```
write(*,*) ( v(i), i=1,3 )
```

realisieren.
Beispielprogramm:

```
program implizite_do_schleife

implicit none
real, dimension(3) :: v, z
integer            :: i

! Fortran 77
do i = 1, 3
 v(i) = sqrt(real(i))
end do

! implizite do-Schleife
write(*,*) ' Ausgabe des Vektors ueber write(*,*) ( v(i), i = 1, 3) : '
write(*,*) ( v(i), i = 1, 3)

! Fortran 90/95
z = v*v
write(*,*)
write(*,*) 'Neue Wertzuweisung z=v*v (Fortran 90 - Syntax)'
write(*,*) 'Ausgabe aller Komponenten von z ueber write(*,*) z :'
write(*,*) z

end program implizite_do_schleife
```

Mit einer impliziten do-Schleife lassen sich analog Werte zeilenorientiert von der Standardeingabe oder von Dateien einlesen und sukzessive den einzelnen Komponenten eines Datenfelds zuweisen.

9.3 Ein- und Ausgabe eines eindimensionalen Arrays in Fortran 90/95

In Fortran 90/95 lässt sich die implizite do-Schleife zur Ausgabe von Datenfeldern durch ein einfacheres Konstrukt ersetzen. Zum Beispiel wenn vom obigen Vektor v alle Komponenten nacheinander (in einer Zeile) ausgegeben werden sollen:

```
write(*,*) v
```

bzw., wenn man nur Unterbereiche des Vektors `v`, z.B. hintereinander die 1. und 2. Komponente ausgeben möchte, so kann man dies einfach durch Angabe des Indexbereiches realisieren:

```
write(*,*) v(1:2)
```

Beispielprogramm:

```
program fortran90_io

implicit none
real, dimension(3) :: v
integer            :: i

do i = 1, 3
 v(i) = sqrt(real(i))
end do

! Fortran 90/95
write(*,*) ' Ausgabe des Vektors ueber:   write(*,*) v '
write(*,*) v
write(*,*)
write(*,*) ' Ausgabe der ersten beiden Vektorkomponenten &
&ueber:   write(*,*) v(1:2): '
write(*,*) v(1:2)

end program fortran90_io
```

Analog zum obigen Beispiel kann man in Fortran 90/95 auch zeilenorientiert einlesen.

9.4 Deklaration und Ausgabe zweidimensionaler Datenfelder (statisch)

Bei der statischen Deklaration zweidimensionaler Datenfeldern wird - wie in der Mathematik - die erste Angabe in dem Attribut `dimension` mit der Zeilendimension und der zweite Wert mit der Spaltendimension assoziiert. Zum Beispiel definiert

```
integer, dimension(3,4) :: a
```

ein zweidimensionales Datenfeld `a`, welches 3 Zeilen und 4 Spalten und somit 12 Komponenten aufweist (`a` ist somit eine 3 x 4 - Matrix mit der Zeilendimension 3 und der Spaltendimension 4)

```
a(1,1) a(1,2) a(1,3) a(1,4)
a(2,1) a(2,2) a(2,3) a(2,4)
a(3,1) a(3,2) a(3,3) a(3,4)
```

Im Speicher werden diese Werte nacheinander abgelegt. **Achtung:** Dabei werden im Memory des Rechners die Komponenten des Arrays `a` spaltenweise abgelegt. In den Speicherplätzen stehen somit aufeinanderfolgend

```
a(1,1) a(2,1) a(3,1) a(1,2) a(2,2) a(3,2) a(1,3) a(2,3) a(3,3) a(1,4) ...
```

Bei einem Speicherzugriff auf die (i,j)-te Komponente eines n x m Datenfeldes, also auf a(i,j) wird immer die relative Indexposition zum Beginn der ersten Komponente des Datenfeldes (hier a(1,1)) berechnet. Von dieser Startposition aus wird zur Speicherstelle der (i,j)-ten Komponente gesprungen und danach der dortige Inhalt je nach Befehl ausgelesen oder abgelegt.
Will man die (i,j)-te Komponente eines n x m - Datenfeldes auslesen, so muss sich der Zeiger, der auf den Inhalt des auszulesenden Datenfeldes zeigen soll, relativ zum Beginn des Speicherplatzes des Datenfeldes

((j-1) * Zeilendimension + (i-1)) * Anzahl der Byte pro Datentyp
= ((j-1) * n + (i-1)) * * Anzahl der Byte pro Datentyp

weiterbewegen.
Konkreter anhand des obigen Zahlenbeispiels: Will man z.B. den Wert von a(2,4), die 4. Komponente in der 2. Zeile des Feldes a, auslesen, so wird von dem Beginn des Datenfeldes

((j-1) * Zeilendimension + (i-1)) * Anzahl der Byte pro Datentyp
= ((4-1)*3+(2-1)) * 4 Byte für den Datentyp integer
= (3*3 + 1) * 4 Byte
= 40 Byte

weitergegangen und die folgenden 4 Bytes als Wert der Komponente a(2,4) aus dem Speicher ausgelesen. Als Gegenprobe stellt man fest, dass sich aufgrund der spaltenweisen sequentiellen Datensicherung, die Komponente a(2,4) an 11. Stelle in der Speicherreihenfolge befindet, so dass beim Datentyp integer bei 4 Byte als interne Repräsentation der Zahlen relativ zum Beginn des Speicherbereichs der Matrixkomponenten 40 Byte übersprungen werden müssen, bevor die zu a(2,4) gehörenden 4 Byte ausgelesen und weiterverarbeitet werden können.
Dementsprechend lässt sich als zweites Beispiel z.B. der Wert von a(3,2) (die Matrixkomponente in der 3. Zeile und der 2. Spalte) in den 4 Byte ab dem

((2-1)*3+ (3-1))*4 Byte
= 5*4 Byte

ab dem 20. Byte relativ zum Beginn des Datenfeldes finden.
Werden unmittelbar aufeinanderfolgende Speicherstellen eines Datenfeldes ausgelesen, z.B. a(1,3) nach a(3,2), so muss der Computer diesmal nicht die relative Position berechnen, sondern „weiß “, dass er nur die folgenden (hier 4) Bytes auszulesen braucht und erspart sich den sonst anfallenden Zeitaufwand für die relative Positionsberechnung.
Fazit: Insbesondere bei grossen mehrdimensionalen Datenfeldern ist der Zugriff auf unmittelbar aufeinanderfolgende Speicherplätze sehr viel schneller, wenn der Zugriff spaltenorientiert erfolgt, da die relativen Positionen bezüglich des Anfangspunktes des Arrays in diesem Fall nicht extra berechnet werden müssen.
Merke: Eine laufzeitoptimierte Programmierung im Umgang mit Datenfeldern muss sich an der internen Organisation des Speichers orientieren.
Beispielprogramm:

```
1 !
2 ! Beispielprogramm zur
```

```
! Vereinbarung eines zweidimensionalen Datenfeldes (Matrix)
!
! matrix(i,j) ist die Matrixkomponente in
!             der i-ten Zeile und der j-ten Spalte
!
! Das Programm setzt den Wert der Matrixkomponente
! matrix(i,j) auf den Wert einer Zahl in der Form i.j, so dass sich
! anhand der Zahl der Zeilen- und der Spaltenindex ablesen laesst

program matrix_beispiel
   implicit none

   real, dimension (3,3) :: matrix   ! Name des Datenfeldes
   integer               :: i, j     ! Zeilen- und Spaltenindex

   write(*,*) 'Zur Kontrolle des Schleifendurchlaufs werden jeweils'
   write(*,*) 'der aktuelle Zeilenindex, Spaltenindex und Wert innerhalb'
   write(*,*) 'der Schleife ausgegeben'
   write(*,*)
   do j = 1, 3
    do i = 1, 3
       matrix(i,j) = real(i) + 0.1 * real(j)
       write(*,*) 'Zeile:',i,'   Spalte: ', j,'   Wert: ',  matrix(i,j)
    end do
   end do

! Achtung: Fortran speichert die Komponenten der Matrix sequentiell
!          der Reihe nach _spaltenorientiert_ ab
!          Im Speicher werden also nacheinander
!          matrix(1,1) matrix(2,1) matrix(3,1) matrix(2,1) ... matrix(3,3)
!          abgelegt.
!          Will man die Laufzeit von Programmen optimieren, sollte
!          auf eine speichergerechte Programmierung der Schleifen achten
!     d.h. z.B. bei geschachtelten Schleifen die Komponenten der Matrix
!          spaltenweise zu durchlaufen

write(*,*)
write(*,*) 'Ausgabe der Matrix &
&(write(*,''(3(1X,F5.1))'') ( (matrix(i,j),j=1,3), i=1,3 )'
write(*,'(3(1X,F5.1))') ( (matrix(i,j),j=1,3), i=1,3 )
                            ! implizite geschachtelte
                            ! write-Anweisung, die dafuer
                            ! sorgt, dass die uebliche
                            ! mathematische Reihenfolge
                            ! eingehalten wird
write(*,*)
write(*,*) 'Ausgabe nach Fortran 90/95 - Syntax'
do i = 1, 3
 write(*,'(3(1X,F5.1))') matrix(i,1:3)
end do

write(*,*)
write(*,*) 'Fortran 90/95 bietet die Moeglichkeit, Datenfelder'
write(*,*) 'direkt in der im Speicher vorliegenden (sequentiellen) Form'
write(*,*) 'auszuschreiben ( write(*,*) matrix )'
```

```
58 write(*,*)
59
60 write(*,*) matrix
61
62 write(*,*)
63 write(*,*) 'Bemerkung:'
64 write(*,*) 'Die␣Komponenten␣des␣2-dim.␣Datenfeldes␣wurden␣in␣der'
65 write(*,*) 'speicheraequivalenten␣Reihenfolge␣und␣somit␣'
66 write(*,*) 'spaltenorientiert␣ausgegeben.'
67
68 end program matrix_beispiel
```

Das Programm `zeitverbrauch` ist eine Laufzeitmessung, welche in diesem Fall eine fast 50%-ige Zeitersparnis beim spaltenorientierten Durchlaufen der Matrix im Vergleich zum zeilenorientierten Durchgang zeigt.
Beispielprogramm:

```
1  program zeitverbrauch
2
3  implicit none
4  integer, parameter :: n = 2000    ! n fuer n x n - Matrix
5  real, dimension (n,n) :: a        ! n x n - Matrix
6  integer :: i, j, m                ! Schleifenindizes
7  integer :: c1, c2, c3             ! Systemzeitmessung
8  integer :: cc1, cc2, cc3          ! Systemzeitmessung
9  real    :: time1, time2
10 real :: random                    ! Datentyp-Deklaration fuer die
11                                   ! Fortran-Funktion zur Erzeugung
12                                   ! von (Pseudo-Zufallszahlen)
13
14 100 format(1X,A,I4.4,A,I4.4,A,F7.5) ! Formatbeschreiber zur
15                                     ! Ausgabe der Matrixkomponenten
16                                     ! I4.4 anpassen an max. Index
17                                     ! I4.4 fuer 1000 <= n <= 9999
18 write(*,*) 'Anzahl␣der␣Zyklen:␣', n
19 write(*,*)
20
21 ! Initialisierung des Pseudozufallsgenerators
22 call random_seed()
23
24 call system_clock(c1,c2,c3)       ! Fortran 90/95-Routine
25 write(*,*) 'system_clock:␣', c1, c2, c3
26 call cpu_time(time1)              ! Fortran 95-Routine
27 write(*,*) 'cpu_time␣=␣', time1
28
29 ! a) zeilenorientierte Vorgehensweise
30 !
31 !
32 write(*,*) '===>␣zeilenorientiertes␣Vorgehen'
33 do i = 1, n
34  do j = 1, n
35   a(i,j) = random()    ! Zufallszahl zw. 0 und 1
36  end do
37 end do
38
```

```
do m = 1, n       ! Multiplikationsprozess wird n mal wiederholt
 do i = 1, n
  do j = 1, n
   a(i,j) = a(i,j)*a(i,j)
  end do
 end do
end do

! b) spaltenorientierte Vorgehensweise
! gemaess der internen Speicherreihenfolge (spaltenorientiert)

! write(*,*) '===> spaltenorientiertes, speicherkonformes Vorgehen'
! do j = 1, n
!  do i = 1, n
!   a(i,j) = random()
!  end do
! end do
!
! do m = 1, n
!  do j = 1, n
!   do i = 1, n
!    a(i,j) = a(i,j)*a(i,j)
!   end do
!  end do
! end do

! Bildschirmausgabe (bei Bedarf einkommentieren)
!
! do i = 1, n
!  do j = 1, n
!   write(*,100) 'a(',i,',',j,') = ', a(i,j)
!  end do
! end do

call system_clock(cc1,cc2,cc3)
write(*,*) 'system_clock: ', cc1, cc2, cc3
call cpu_time(time2)
write(*,*) 'cpu_time = ', time2
write(*,*)
write(*,*) 'Zeitverbrauch:'
write(*,*) 'nach system_clock : ', cc1 - c1
write(*,*) 'nach cpu_time      : ', time2 - time1

end program zeitverbrauch
!-------------------------------------------------------------
!!! Auswertung:
!-------------------------------------------------------------
!! Bestimmung der Zeitdifferenz
!! Bildschirmausgabe mit Laptop Intel Pentium PM 2.0 GHz, 1024 MB

!! Bildschirmausgabe, falls der nur der zeilenorientierte Teil
!! a) verwendet wurde

!  system_clock:    162412491         1000   2147483647
!  cpu_time =      31142.3
```

```
!  system_clock:    162594232         1000  2147483647
!  cpu_time =      31323.9

!! Bildschirmausgabe, falls nur der spaltenorientierte Teil
!! b) verwendet wurde

! system_clock:    162097037         1000  2147483647
! cpu_time =      30826.8
! system_clock:    162188659         1000  2147483647
! cpu_time =      30918.3

!! Aus dem ersten system_clock-Werten lassen sich der Zeitbedarf fuer
!! der Anweisungsteil mit den Matrixdurchlauefen berechnen

!!  tz = Laufzeit(system_clock,zeilenweise) : 181741
!!  ts = Laufzeit(system_clock,spaltenweise) : 91622

!! Zeitersparnis bei speichergerechten Durchlauf durch Matrix:
!! nach system_clock:

!! tz / ts = 1.98, d.h. bei dem unguenstigen Durchlauf der
!!                 Matrixkomponenten braucht man
!!                 im Beispiel unnoetigerweise fast
!!                 das Doppelte an Rechenzeit

!! Die analoge Berechnung laesst sich mit den cpu_time-Werten durchfuehren
!! und fuehrt zum gleichen Zahlenverhaeltnis

!! t_z = Laufzeit(cpu_time,zeilenweise): 181.6
!! t_s = Laufzeit(cpu_time,spalten): 91.5
!! t_z / t_s = 1.98

!! Fazit: je nach Struktur des Programms kann man, falls haeufige
!! Durchlaeufe in mehrdimensionalen grossen Datenfeldern
!! notwendig werden, durch eine speicherangepasste Gestaltung
!! des Programms erheblich Rechenzeit einsparen.
```

9.5 Verschiebung der Indexgrenzen in Datenfeldern bei der statischen Deklaration

Möchte man das erste Element eines Datenfeldes nicht mit dem Index 1 beginnen lassen, so kann man sich die (statische) Felddeklaration entsprechend anpassen:

```
real, dimension(0:2) :: x
```

würde das Feld x mit den Komponenten x(0), x(1) und x(2) definieren.
Beispielprogramm:

```
program feldgrenzen

implicit none
real, dimension(0:2) :: x
```

```
5
6  x(0) = -1.0
7  x(1) = 7.0
8  x(2) = x(0) * x(1)
9
10 write(*,*) 'x(0) = ', x(0)
11 write(*,*) 'x(1) = ', x(1)
12 write(*,*) 'x(2) = x(0) * x(1) = ', x(2)
13
14 end program feldgrenzen
```

Es lassen sich auch negative ganze Zahlen als Indexgrenzen angeben.
Bei der Angabe der Indexgrenzen muss stets der als untere Grenze angegebene Wert kleiner als die Zahl für die Obergrenze sein.
Beispielprogramm:

```
1  program feldgrenzen
2
3  implicit none
4  real, dimension(0:2)    :: x
5  real, dimension(-3:-1) :: y
6  real, dimension(-1:1)   :: z
7  integer :: i
8
9  x(0) = -1.0
10 x(1) = 7.0
11 x(2) = x(0) * x(1)
12
13 write(*,*) 'x(0) = ', x(0)
14 write(*,*) 'x(1) = ', x(1)
15 write(*,*) 'x(2) = x(0) * x(1) = ', x(2)
16
17 y = 3.0       ! alle Komponenten von y auf den Wert 3.0 setzen
18 write(*,*)
19 write(*,*) 'y(-3) = ', y(-3)
20 write(*,*) 'y(-2) = ', y(-2)
21 write(*,*) 'y(-1) = ', y(-1)
22
23 do i = -1, 1
24  z(i) = real(i)
25 end do
26 write(*,*)
27 write(*,*) 'z(-1) = ', z(-1)
28 write(*, *) 'z( 0) = ', z(0)
29 write(*,*) 'z(-1) = ', z(-1)
30
31 end program feldgrenzen
```

9.6 Initialisierung von Datenfeldern (engl. *arrays*) mit Werten (statisch)

Bei der statischen Deklaration von Datenfeldern ist es in Fortran 90/95 möglich, durch eine direkte Zuweisung alle Komponenten mit dem angegebenen Wert vorzubelegen. Beispielsweise wird mit

```
real, dimension(10) :: v = 0.0
```

jede der 10 Komponenten des eindimensionalen Datenfelds (Vektor) `v(0) v(1),..., v(10)` auf den Wert `0.0` gesetzt.
Will man ein Datenfeld bei der Initialisierung mit verschiedenen Werten vorbelegen, kann man diese durch einen Datenfeld-Konstruktor angeben. Beispiel:

```
real, dimension(3) :: w = (/ 1.1, 2.2, 3.3 /)
```

setzt `w(1)` auf den Wert `1.1`, `w(2)` auf `2.2` und `w(3)` auf `3.3`.
Dieses Beispiel der Wertzuweisung mittels eines Array-Konstruktors lässt sich noch um den Fall mit impliziten Berechnungsvorschriften bei der Deklaration ergänzen. Innerhalb des Konstruktors kann man verschachtelte Schleifen einbauen und so raffinierte Vorbelegungen eines Datenfeldes in Abhängigkeit von den Indizes kreieren.
Angenommen, es soll die Sequenz

```
1.  2.  6.  4.  5.  12.  7. 8. 18. 10. 11. 24.
```

dem Datenfeld `z(1), z(2), ... z(12)` zugeordnet werden, so kann man dies in der Form

```
integer :: i, j
real, dimension(12) :: z = (/ ( (1.0*i+3.0*(j-1),i=1,2), &
6.0*j,j=1,4) /)
```

tun.
Beispielprogramm:

```
1  program array_constructor
2
3  implicit none
4  integer :: i, j
5  real, dimension(12) :: z =  (/ ( (1.0*i+3.0*(j-1),i=1,2), &
6  6.0*j,j=1,4) /)
7
8  do i = 1,12
9   write(*,*) i, z(i)
10 end do
11
12 end program array_constructor
```

Array-Konstruktoren lassen sich bei der Datentyp-Deklaration auch einsetzen, um **mehrdimensionale Felder** mit Werten vorzubelegen. In diesem Zusammenhang muss man sich in Erinnerung rufen, wie die Werte mehrdimensionaler Datenfelder im Speicher abgelegt werden. Bei einem zweidimensionalen Datenfeld geschieht dies spaltenorientiert. In dieser

speicheräquivalenten Reihenfolge müssen die zu initialisierenden Wertepaare angegeben werden - zusätzlich ist die Funktion `reshape` und Angabe der gewünschten mehrdimensionalen Form des Datenfeldes notwendig. Angenommen, eine Matrix `matrix_1` mit 4 Zeilen und 3 Spalten soll die Vorbelegung

```
1.0 2.0 3.0
1.0 2.0 3.0
1.0 2.0 3.0
1.0 2.0 3.0
```

bekommen, so lautet die zugehörige Fortran 90/95-Zeile zur Initialisierung

```
real, dimension(4,3) :: matrix_1 = &
reshape( (/1.,1.,1.,1.,2.,2.,2.,2.,3.,3.,3.,3./), (/4,3/) )
```

Beispielprogramm:

```
!
! Beispiele zur Initialisierung von 2-dimensionalen Feldern
!
program matrix_reshape

implicit none

! matrix_1 ist 4x3-Matrix (4 Zeilen und 3 Spalten)
! und wird bei der Variablendeklaration
! gleich mit Werten belegt
!
! In der Matrix soll stehen
!
!   1.0 2.0 3.0
!   1.0 2.0 3.0
!   1.0 2.0 3.0
!   1.0 2.0 3.0

real, dimension(4,3) :: matrix_1 = &
reshape( (/1.,1.,1.,1.,2.,2.,2.,2.,3.,3.,3.,3./), (/4,3/) )

integer :: i, j  ! Hilfsvariablen fuer die implizite do-Schleife
                 ! bei der Ausgabe der Matrixelemente

! Formatbeschreiber fuer eine Zeile der Matrix
10 format(3(2X,F3.1))   ! immer 3 Matrixelemente in einer Zeile
                        ! mit jeweils 2 Blanks, dann die
                        ! Festpunktzahl 3 Felder breit
                        ! 1 Nachkommastelle und 1 Vorkommastelle

write(*,*) 'Ausgabe der Matrix ueber eine implizite do-Schleife: '
write(*,*)
write(*,*) '10 format(3(2X,F3.1))'
write(*,*) 'write(*,10)  ( (matrix_1(i,j), j=1,3), i=1,4 :'
write(*,*)
! Fortran 77
write(*,10)  ( (matrix_1(i,j), j=1,3), i=1,4)

```

```
39 ! Fortran 90
40 write(*,*)
41 write(*,*) 'Ausgabe der Matrix mit'
42 write(*,*) 'write(*,10) matrix(i,1:3) :'
43 write(*,*)
44 do i = 1, 4
45  write(*,10) matrix_1(i,1:3)
46 end do
47
48 end program matrix_reshape
```

Die auf die obigen Arten bei der Deklaration der Datenfelder zugewiesenen Werte lassen sich jederzeit innerhalb des Anweisungsteils der Programms wieder verändern. Man kann auf einzelne Komponenten eines Arrays zuweisen oder man kann z.B. mit Hilfe von `do`-Schleifen und Schleifenindizes das gesamte Datenfeld oder Teilbereiche durchlaufen, um die dort gespeicherten Werte auszugeben, weiter zuverarbeiten oder zu verändern.
`do`-Schleifen lassen sich auch nutzen, um Datenfelder erst nach dem Deklarationsteil im Anweisungsteil mit Werten vorzubelegen, wie dies z.B. im Programm `matrix_beispiel` zu sehen ist.
Beispielprogramm:

```
1  !
2  ! Beispielprogramm zur
3  ! Vereinbarung eines zweidimensionalen Datenfeldes (Matrix)
4  !
5  ! matrix(i,j) ist die Matrixkomponente in
6  !              der i-ten Zeile und der j-ten Spalte
7  !
8  ! Das Programm setzt den Wert der Matrixkomponente
9  ! matrix(i,j) auf den Wert einer Zahl in der Form i.j, so dass sich
10 ! anhand der Zahl der Zeilen- und der Spaltenindex ablesen laesst
11
12 program matrix_beispiel
13
14 implicit none
15 real, dimension (3,3) :: matrix   ! Name des Datenfeldes
16 integer               :: i, j     ! Zeilen- und Spaltenindex
17
18 write(*,*) 'Zur Kontrolle des Schleifendurchlaufs werden jeweils'
19 write(*,*) 'der aktuelle Zeilenindex, Spaltenindex und Wert innerhalb'
20 write(*,*) 'der Schleife ausgegeben'
21 write(*,*)
22 do j = 1, 3
23  do i = 1, 3
24   matrix(i,j) = real(i) + 0.1 * real(j)
25   write(*,*) 'Zeile:',i,'   Spalte: ', j,'   Wert: ',  matrix(i,j)
26  end do
27 end do
28
29 ! Achtung: Fortran speichert die Komponenten der Matrix sequentiell
30 !          der Reihe nach _spaltenorientiert_ ab
31 !          Im Speicher werden also nacheinander
32 !          matrix(1,1) matrix(2,1) matrix(3,1) matrix(2,1) ... matrix(3,3)
33 !          abgelegt.
```

```
!           Will man die Laufzeit von Programmen optimieren, sollte
!           auf eine speichergerechte Programmierung der Schleifen achten
!            d.h. z.B. bei geschachtelten Schleifen die Komponenten der Matrix
!           spaltenweise zu durchlaufen

write(*,*)
write(*,*) 'Ausgabe der Matrix &
&(write(*,''(3(1X,F5.1))'') ( (matrix(i,j),j=1,3), i=1,3 )'
write(*,'(3(1X,F5.1))') ( (matrix(i,j),j=1,3), i=1,3 )
                        ! implizite geschachtelte
                        ! write-Anweisung, die dafuer
                        ! sorgt, dass die uebliche
                        ! mathematische Reihenfolge
                        ! eingehalten wird
write(*,*)
write(*,*) 'Ausgabe nach Fortran 90/95 - Syntax'
do i = 1, 3
 write(*,'(3(1X,F5.1))') matrix(i,1:3)
end do

write(*,*)
write(*,*) 'Fortran 90/95 bietet die Moeglichkeit, Datenfelder'
write(*,*) 'direkt in der im Speicher vorliegenden (sequentiellen) Form'
write(*,*) 'auszuschreiben ( write(*,*) matrix )'
write(*,*)

write(*,*) matrix

write(*,*)
write(*,*) 'Bemerkung:'
write(*,*) 'Die Komponenten des 2-dim. Datenfeldes wurden in der'
write(*,*) 'speicheraequivalenten Reihenfolge und somit '
write(*,*) 'spaltenorientiert ausgegeben.'

end program matrix_beispiel
```

9.7 Zugriff auf einzelne Komponenten eines Datenfeldes

Wie wir schon mehrmals gesehen haben, lässt sich auf eine einzelne Komponente eines Datenfeldes durch die Angabe des Namens und des entsprechenden Indizes innerhalb des Anweisungsteils jederzeit zugreifen.
Eine Gefahr bei dem Zugriff auf Datenfelder besteht darin, dass manche Compiler nicht selbstständig auf den Indexbereich des Datenfeldes achten. Auch unser **f90**-Compiler tut dies erst, wenn man explizit statt mit `f90` mit `f90 -C` das Programm übersetzt. Erst mit dem zusätzlichen Compiler-Schalter `-C` wird bei der Übersetzung zusätzlicher Code generiert, der bei der Programmausführung bei der Überschreitung des Indexbereichs zu einer Fehlermeldung führt.
Beispielprogramm:

```
program bound_check

implicit none
```

```
real, dimension(3) :: a = (/1.0,2.0,3.0/)
real, dimension(3) :: b = (/4.0,5.0,6.0/)

write(*,*) a(4)

end program bound_check
```

9.8 Zugriff auf Datenfelder als Ganzes (Fortran 90/95)

Will man nicht nur einen einzelnen Wert in einem Datenfeld verändern, sondern mathematische Operationen nach einem vorgegebenen Schema an allen Komponenten des Datenfeldes vornehmen, so geht dies jederzeit über do-Schleifen.
Das folgende ausführlichere Beispielprogramm zeigt diese Möglichkeiten der älteren Fortran 77-Syntax auf und ergänzt diese mit einigen neueren Möglichkeiten der Verarbeitung von Datenfeldern als Ganzem aus Fortran 90/95.
Beispielprogramm:

```
!
! Grundlegendes  zu Feldern (engl. Arrays)
!

program array_definition

implicit none
real, dimension(10) :: x       ! Fortran 90/95 - Syntax
real, dimension(5)  :: summe
real                :: y(5)    ! ginge noch (alter Fortran 77 - Syntax)
                               ! in neuen Programmen nicht mehr verwenden

real, dimension(5)  :: z = (/2.0,3.0,4.0,5.0,6.0/)  ! Wertvorbelegung
                                                    ! in Fortran 90/95
integer :: i, j

! Die Arrays werden durch eine do-Schleife mit Werten belegt
! (eine Moeglichkeit)

do i = 1, 10
 x(i) = real(i)
end do

do j = 1, 5
 y(j) = -real(j)
end do

! Werte ausgeben
!
! 1. Moeglichkeit
! Die Werte in einem Array werden ueber eine do-Schleife wieder ausgegeben

write(*,*)  'Ausgabe des ersten Feldes x:'
do i = 1, 10
 write(*,*) x(i)
```

```
end do

write(*,*)  'Ausgabe␣des␣zweiten␣Feldes␣y␣:'
do j = 1, 5
 write(*,*) y(j)
end do

write(*,*)  'Ausgabe␣des␣dritten␣Feldes␣z␣:'
do j = 1, 5
 write(*,*) z(j)
end do

! 2. Moeglichkeit
! implicite do-Schleife fuer die Ausgabe

write(*,*)
write(*,*)
write(*,*) 'Ausgabe␣ueber␣implizite␣do-Schleife'
write(*,*)
10 format(1X,F12.5)
write(*,*)  'Ausgabe␣des␣ersten␣Feldes␣x␣:'
write(*,10)  (x(i),i=1,10)
write(*,*)  'Ausgabe␣des␣zweiten␣Feldes␣y␣:'
write(*,10)  (y(j),j=1,5)
write(*,*)  'Ausgabe␣des␣dritten␣Feldes␣z␣:'
write(*,10)  (z(j),j=1,5)

write(*,*)
write(*,*) 'Ausgabe␣ueber␣implizite␣do-Schleife'
write(*,*) 'immer␣3␣Werte␣in␣eine␣Zeile␣schreiben'
write(*,*)
20 format(3(1X,F12.5))
write(*,*)  'Ausgabe␣des␣ersten␣Feldes␣x␣:'
write(*,20)  (x(i),i=1,10)
write(*,*)  'Ausgabe␣des␣zweiten␣Feldes␣y␣:'
write(*,20)  (y(j),j=1,5)
write(*,*)  'Ausgabe␣des␣dritten␣Feldes␣z␣:'
write(*,20)  (z(j),j=1,5)

! das Array summe mit Werten vorbelegen

write(*,*)
write(*,*) 'das␣Array␣summe␣wurde␣mit␣Werten␣vorbelegt'
write(*,*)

! Fortran 77

do j = 1, 5
 summe(j) = 1.1
end do
write(*,*)
write(*,*) 'Nach␣Fortran␣77␣-␣Syntax␣(Wert␣jeweils␣1.1)'
write(*,*) (summe(j),j=1,5)

! Fortran 90/95
```

```
91  summe = 0.0
92  write(*,*)
93  write(*,*) 'Nach Fortran 90/95 - Syntax (Wert jeweils 0.0)'
94  write(*,*) (summe(j),j=1,5)
95
96  ! Addition von Arrays (die allgemeinste Form)
97
98  do j = 1, 5
99   summe(j) = y(j) + z(j)
100 end do
101 write(*,*)
102 write(*,*) 'summe(j) = y(j) + z(j)'
103 write(*,*) (summe(j),j=1,5)
104
105 ! Fortran 90/95 - Syntax fuer Arrays der gleichen Laenge
106
107 summe = y - z
108 write(*,*)
109 write(*,*) 'summe = y - z'
110 write(*,*) summe
111
112 end program array_definition
```

Datenfelder der gleichen Ausdehnungen (gleiche Dimensionen, unabhängig vom Indexbereich) lassen sich in Fortran 90/95 paarweise addieren, subtrahieren, multiplizieren und dividieren.
Es lassen sich auf ein Datenfeld als Ganzes die in Fortran 90/95 intrinsisch eingebauten Funktionen anwenden. Diese mathematischen Funktionen lassen sich nur durch die Angabe des Namens des Datenfeldes auf alle Komponenten anwenden - ohne, dass wie dies noch in Fortran 77 notwendig war, `do`-Schleifen zum Durchlaufen der Felder über Indizes aussen herum programmiert zu werden brauchen.
Beispielprogramm:

```
1  program gesamt_zugriff
2
3  implicit none
4  real, dimension (4,3) :: x
5  integer :: i, j
6
7  do j = 1, 3
8   do i = 1, 4
9    x(i,j) = real(i)+0.1*real(j)
10  end do
11 end do
12
13 10 format(3(1X,F4.1))    ! Formatbeschreiber fuer die Matrix
14                          ! immer 3 Werte in einer Zeile
15
16 write(*,*) 'urspruengliche Matrix:'
17 write(*,10) ( (x(i,j), j=1,3), i=1,4 )
18 write(*,*)
19
20 ! von jeder Komponente der Matrix wird 1.0 abgezogen
21 ! Fortran 77 - Syntax
```

```
do j = 1, 3
 do i = 1, 4
  x(i,j) = x(i,j) - 1.0
 end do
end do
write(*,*) 'von␣jeder␣Komponente␣wurde␣1.0␣abgezogen:'
write(*,10) ( (x(i,j), j=1,3), i=1,4 )
write(*,*)

! von jeder Komponente der Matrix wird nochmals 1.0 abgezogen
! Fortran 90/95 - Syntax
x = x - 1.0
write(*,*) 'von␣jeder␣Komponente␣wurde␣nochmals␣1.0␣abgezogen:'
write(*,10) ( (x(i,j), j=1,3), i=1,4 )
write(*,*)

! von jeder Komponente wird der Betrag gebildet
! Fortran 90/95 - Syntax
x = abs(x)
write(*,*) 'von␣jeder␣Komponente␣wurde␣jetzt␣der␣Betrag␣gebildet:'
write(*,10) ( x(i,1:3), i=1,4 )
write(*,*)

end program gesamt_zugriff
```

9.9 Zugriff auf Untermengen mehrdimensionaler Datenfelder und in Fortran 90/95 enthaltene Matrixoperationen

In Fortran 90/95 sind viele Erweiterungen zu Fortran 77 enthalten. Insbesondere bei der Umsetzung von Fragestellungen der numerischen Mathematik in Lösungswege, welche sich von mathematischen Hintergrund her mittels Matrizen und Vektoren formulieren lassen, bietet Fortran 90/95 eine Vielzahl von zusätzlichen nützlichen Funktionen im Vergleich zu Fortran 77 und anderen algorithmisch orientierten Programmiersprachen wie z.B. C und Pascal.
Unter anderem werden folgende Fortran 90/95 spezifischen Features angeboten und lassen sich in der Praxis hervorragend einsetzen. Z.B Zugriff auf Untermatrizen in zwei- oder mehrdimensionalen Datenfeldern, Herausgreifen von Spalten- und Zeilenvektoren (Programm `unterfelder`), oder Matrixoperationen (Programm `matrixoperationen`), sowie typische Matrix-Vektor-Operationen aus der (numerischen) Linearen Algebra im Programm `matrixoperationen2`.
Beispielprogramm:

```
program unterfelder

implicit none
integer, dimension(5,5) :: a
integer                 :: i, j
integer, dimension(3,4) :: b
integer, dimension(2,3) :: c
integer, dimension(5)   :: zeile3, spalte1
```

```
9
10 10 format(5(1X,I3))   ! Formatbeschreiber 5 Werte in einer Zeile
11 20 format(4(1X,I3))   ! Formatbeschreiber 4 Werte in einer Zeile
12 30 format(1X,I3)      ! Formatbeschreiber 1 Wert  in einer Zeile
13 40 format(3(1X,I3))   ! Formatbeschreiber 3 Werte in einer Zeile
14
15 do j = 1, 5
16  do i = 1, 5
17   a(i,j) = (i-1)* 5 + (j-1)
18  end do
19 end do
20
21 write(*,*)
22 write(*,*) 'Matrix a:'
23 !  write(*,10) ( (a(i,j), j=1,5), i=1,5)  !Fortran 77
24 write(*,*) ( a(i,1:5), i=1,5 )
25 write(*,*)
26
27 b = a(1:3,2:5)  ! Fortran 90/95 - Syntax
28
29 write(*,*)
30 write(*,*) 'Untermatrix b = a(1..3,2..5) :'
31 write(*,20) ( (b(i,j), j=1,4), i=1,3)
32 write(*,*)
33
34 c = a(2:3,3:)  ! Fortran 90/95 - Syntax
35
36 write(*,*)
37 write(*,*) 'Untermatrix c = a(2:3,3:) :'
38 write(*,40) ( (c(i,j), j=1,3), i=1,2)
39 write(*,*)
40
41 zeile3 = a(3,:)   ! Fortran 90/95 -Syntax
42 write(*,*)
43 write(*,*) 'Zeilenvektor der 3. Zeile :'
44 write(*,10) zeile3
45
46 spalte1 = a(:,1)
47 write(*,*)
48 write(*,*) 'Spaltenvektor der 1. Spalte :'
49 write(*,30) spalte1
50
51 write(*,*)
52 write(*,*) 'Testfunktionen fuer Arrays'
53 write(*,*) 'shape(a)    = ', shape(a)
54 write(*,*) 'size(a)     = ', size(a)
55 write(*,*) 'lbound(a)   = ', lbound(a)
56 write(*,*) 'ubound(a)   = ', ubound(a)
57 write(*,*)
58 write(*,*) 'c deklariert als real,dimension(2,3) :: c. Zugewiesen&
59 & wurde: c = a(2:3,3:)'
60 write(*,*) 'shape(c)    = ', shape(c)
61 write(*,*) 'size(c)     = ', size(c)
62 write(*,*) 'lbound(c)   = ', lbound(c)
63 write(*,*) 'ubound(c)   = ', ubound(c)
```

```
64
65 end program unterfelder
```

Beispielprogramm:

```
1  program matrixoperationen
2
3  implicit none
4  real, dimension (3,3) :: m1, m2 = 0.0 , m3
5  integer :: i, j
6
7  10 format(3(1X,F4.1))  ! 3 Werte in einer Zeile
8  do j = 1, 3
9   do i = 1, 3
10   m1(i,j) = real(i) + 0.1 * real(j)
11  end do
12 end do
13
14 write(*,*)
15 write(*,*) 'm1 :'
16 write(*,10) ( m1(i,1:3), i=1,3)
17
18 m2(1,3) = 1.0
19 m2(2,2) = 1.0
20 m2(3,1) = 1.0
21
22 write(*,*)
23 write(*,*) 'm2 :'
24 write(*,10) ( m2(i,1:3), i=1,3)
25
26 m3 = m1*m2
27
28 write(*,*)
29 write(*,*) 'm3 = m1*m2 :'
30 write(*,10) ( m3(i,1:3), i=1,3)
31
32 m3 = matmul(m1,m2)
33 write(*,*)
34 write(*,*) 'm3 = matmul(m1,m2) :'
35 write(*,10) ( m3(i,1:3), i=1,3)
36
37 end program matrixoperationen
```

Beispielprogramm:

```
1  program matrixoperationen2
2
3  implicit none
4  real, dimension(3,3)     :: m1, m2 = 0.0 , &
5    m3 = reshape((/ ((1.0/(i+j-1.0),i=1,3),j=1,3) /),(/3,3/))
6  logical, dimension(3,3) :: mask = .false.
7  real, dimension (3)      :: v1 = (/1.0, 2.0,-1.0/), v2 = (/1.0, 0.0, -1.0/)
8  integer :: i, j
9
10 10 format(3(1X,F4.1))  ! 3 Werte in einer Zeile
11 20 format(3(1X,F4.1/)) ! 3-komponentiger Vektor als Spaltenvektor
12
```

```
do j = 1, 3
 do i = 1, 3
  m1(i,j) = real(i) + 0.1 * real(j)
 end do
end do

write(*,*) 'm1:'
write(*,10) (m1(i,1:3),i=1,3)
write(*,*) 'm2:'
write(*,10) (m2(i,1:3),i=1,3)
write(*,*) 'm3:'
write(*,10) (m3(i,1:3),i=1,3)

write(*,*) 'm2 = transpose(m1):'
m2 = transpose(m1)
write(*,10) (m2(i,1:3),i=1,3)

write(*,*) 'Die 3. Zeile von m2 mit -1.0 multiplizieren'
m2(3,:) = -1.0*m2(3,:)
write(*,*)
write(*,*) 'm2:'
write(*,10) (m2(i,1:3),i=1,3)

write(*,*) 'Groesster Wert von m2:'
write(*,*) maxval(m2)
write(*,*) 'bei Index: ', maxloc(m2)
write(*,*) 'Groesster Wert von m2 entlang der Hauptdiagonalen:'
! Die Werte von mask entlang der Hauptdiagonalen werden auf .true.
! gesetzt
do i = 1, 3
 mask(i,i) = .true.
end do
write(*,*) maxval(m2,mask)
write(*,*) 'bei Index: ', maxloc(m2,mask)

write(*,*)
write(*,*) 'm2:'
write(*,10) (m2(i,1:3),i=1,3)

write(*,*) 'Spaltenmaxima von m2:'
write(*,10) maxval(m2,1)

write(*,*) 'Zeilenmaxima von m2:'
write(*,20) maxval(m2,2)

write(*,*) 'm3 = cshift(m2,2,1)'
m3 = cshift(m2,2,1)
write(*,10) (m3(i,1:3),i=1,3)

write(*,*)
write(*,*) 'Vektoren:'
write(*,*) 'v1 als Zeilenvektor:'
write(*,10) v1
write(*,*) 'v2 als Spaltenvektor:'
write(*,20) v2
```

```
68 write(*,*) 'Skalarprodukt␣v1.v2:'
69 write(*,*) dot_product(v1,v2)
70
71 write(*,*) 'Betrag␣des␣Vektors␣v1␣(Euklidsche␣Norm):'
72 write(*,*) sqrt(dot_product(v1,v1))
73
74 end program matrixoperationen2
```

9.10 Übersichtstabelle über Operationen auf Datenfeldern als Ganzem (Fortran 90/95)

- grundsätzlich lassen sich durch eine einfache Wertzuweisung alle Komponenten eines Datenfeldes auf den gleichen Wert setzen
  - z.B. `matrix = 0.0`
- der Zugriff auf Teilbereiche von Datenfeldern erfolgt durch Angabe von Indizes
  - z.B. `matrix(1,1:3)` entspricht den Komponenten in der 1. Zeile in den Spalten 1-3, falls z.B. die Matrix als `real, dimension(3,3) :: matrix` deklariert wurde
- alle in Fortran 90/95 intrinisch enthaltenen mathematischen Funktionen lassen sich auch auf Datenfelder anwenden
- **Skalarprodukt** zweier Vektoren (`vektor1` und `vektor2` seien 1-dim. Datenfelder gleicher Länge) berechnen
  - `dot_product(vektor1,vektor2)`
- **Matrix-Matrix-Multiplikation** zweier Matrizen oder **Matrix-Vektor-Multiplikation** (n x m - Matrix mit n-komponentigem Vektor `vektor1`)
  - `matmul(matrix1,matrix2)`
  - `matmul(matrix1,vektor1)`
- die **Transponierte** einer (n x m) - Matrix berechnen
  - `transpose(matrix)`
- finden der Indizes der größten Komponente eines Datenfeldes
  - `maxloc(vektor1)` oder
  - `maxloc(matrix1)` oder
  - `maxloc(mehrdimensionales_datenfeld)`

Beispielprogramm:

```
program test_maxloc

implicit none
real, dimension(3,4)   :: matrix
integer                :: i, j
integer, dimension(2) :: indices_groesster_Wert

! Die Matrixkomponenten werden mit einem numerischen Wert
! der dem Zeilenindex als Vorkommastelle und dem
! Spaltenindex als Nachkommastelle belegt

do j = 1, 4
 do i = 1, 3
  matrix(i,j) = real(i) + 0.1*real(j)
 end do
end do

matrix(2,3) = 10.0

! Bildschirmausgabe

write(*,*) 'Die Matrix matrix sieht nun so aus:'
write(*,*)
do i = 1, 3
 write(*,*) (matrix(i,j), j=1,4)
end do

! maxloc Anwendung und ausgeben
write(*,*)
write(*,*) 'Rueckgabewerte von maxloc(matrix) sind die Indices, an denen'
write(*,*) 'die groesste Komponente sitzt:'
write(*,*)
write(*,*) maxloc(matrix)

! um mit den Rueckgabewerten von maxloc(matrix) weiterarbeiten zu
! koennen, braucht man ein Datenfeld, dessen Anzahl der Komponenten
! der Dimension von matrix entspricht

indices_groesster_Wert = maxloc(matrix)
write(*,*)
write(*,*) 'Zeilenindex der groessten Komponente von matrix:  ', &
indices_groesster_Wert(1)
write(*,*) 'Spaltenindex der groessten Komponente von matrix: ', &
indices_groesster_Wert(2)

end program test_maxloc
```

- finden der Indizes eines mehrdimensionalen Arrays, an dem sich die kleinste Komponente befindet
  - `minloc(vektor1)` oder
  - `minloc(matrix1)` oder
  - `minloc(mehrdimensionales_datenfeld)`

- finden des größten Wertes in einem Datenfeld
  - `maxval(vektor1)` oder
  - `maxval(matrix1)` oder
  - `maxval(mehrdimensionales_datenfeld)`
  - **Spaltenmaxima** berechnen: `maxval(matrix1,1)`
  - **Zeilenmaxima** berechnen: `maxval(matrix1,2)`
- finden des kleinsten Wertes in einem Datenfeld
  - `minval(vektor1)` oder
  - `minval(matrix1)` oder
  - `minval(mehrdimensionales_datenfeld)`
- Summe aller Komponenten in einem Datenfeld berechnen
  - `sum(vektor1)` oder
  - `sum(matrix1)` oder
  - `sum(mehrdimensionales_datenfeld)`
- Produkt aller Komponenten in einem Datenfeld berechnen
  - `product(vektor1)` oder
  - `product(matrix1)` oder
  - `product(mehrdimensionales_datenfeld)`
- **Zyklische Rotation** von Zeilen oder Spalten einer Matrix
  - `cshift(matrix,anz,dim)`
  - `anz`: Anzahl der Plätze, um die verschoben werden soll
  - `dim`: Dimension in der verschoben werden soll

Die Befehle `maxloc`, `minloc`, `maxval`, `minval`, `sum` und `product` erlauben es, hinter dem Datenfeldnamen mit einem Komma getrennt zusätzlich ein zweites Datenfeld mit exakt gleicher Dimensionierung anzugeben, dessen Komponenten vom Datentyp `logical` sein müssen. Man spricht von einer sogenannten **Maske**, welche vor dem Ausführen der gewünschten Operation über das Datenfeld gelegt wird, so dass nur noch diejenigen Komponenten berücksichtigt werden, bei denen sich an der korrespondierenden Stelle der Maske der Wert `.TRUE.` befindet.
Beispielprogramm:

```
1 program matrix_demos
2
3 implicit none
4 integer, parameter       :: n = 3
5 integer, dimension(n,n) :: matrix, matrix_neu
6 logical, dimension(n,n) :: mask
```

```
integer :: i, j

do j = 1, n
 do i = 1, n
  matrix(i,j) = (3*i**2 - 3*i+ 2*j) / 2
 end do
end do

write(*,*) 'Matrix:'
do i = 1, n
 write(*,*) matrix(i,1:n)
end do

write(*,*)
write(*,*) 'Matrixkomponenten in Speicherreihenfolge:'
write(*,*) matrix

do j = 1, n
 do i = 1, n
  if ( mod(i+j,2) /= 0 ) then
   mask(i,j) = .true.
  else
   mask(i,j) = .false.
  end if
 end do
end do

write(*,*)
write(*,*) 'Logische mask-Matrix:'
do i = 1, n
 write(*,*) mask(i,1:n)
end do

write(*,*)
write(*,*) 'maxval(matrix,mask) = ', maxval(matrix,mask)
write(*,*)
write(*,*) 'maxval(matrix)       = ', maxval(matrix)
write(*,*)
write(*,*) 'maxval(matrix,1)     = ', maxval(matrix,1)
write(*,*) 'Maximalwerte entlang der 1. Dimension => Spaltenmaxima'
write(*,*)
write(*,*) 'maxval(matrix,2)     = ', maxval(matrix,2)
write(*,*) 'Maximalwerte entlang der 2. Dimension => Zeilenmaxima'
write(*,*)
write(*,*) 'Matrix:'
do i = 1, n
 write(*,*) matrix(i,1:n)
end do

write(*,*)
write(*,*) 'cshift(matrix,1,2), d.h. um einen Index in der 2. Dimension verschieben:'
matrix_neu = cshift(matrix,1,2)
do i = 1, n
 write(*,*) matrix_neu(i,1:n)
end do
```

```
62
63 write(*,*)
64 write(*,*) 'Matrix:'
65 do i = 1, n
66  write(*,*) matrix(i,1:n)
67 end do
68
69 end program matrix_demos
```

9.11 Formatierte Ausgabe von Datenfeldern

Will man Datenfelder formatiert und nicht listengesteuert ausgeben und ist die Anzahl der pro Zeile auszugebenden Komponenten bekannt, so kann man den entsprechenden Formatbeschreiber direkt angeben.
Beispielprogramm:

```
1  program vektor
2
3  implicit none
4  integer, parameter :: n = 9     ! bei Bedarf anpassen
5  real, dimension(n) :: v         ! Vektor v
6  integer :: i
7  character (len=20) :: fb = '(9(2X,F5.3))' ! Bei Bedarf anpassen
8                                            ! vor der 2. runden Klammer
9                                            ! muss der Zahlenwert von n stehen
10
11 write(*,*) fb
12
13 do i = 1, n
14  v(i) = 1.0/real(i)
15 end do
16
17 ! Bildschirmausgabe des Vektors
18 write(*,fb) v
19
20 end program vektor
```

Kann sich jedoch die pro Zeile auszugebende Anzahl an Werten ändern, so empfiehlt es sich, die Formatbeschreiberkette mittels Zeichenkettenverarbeitung zusammenzubauen. Sind pro Zeile z.B. `n` Werte, wobei `n` zwischen 1 und maximal 9 liegen darf, formatiert zu schreiben, kann man mittels der Funktionen `iachar` und `achar` die Zahl `n` in ihr korrespondierendes `character`-Zeichen umwandeln. Beispielprogramm:

```
1  program vektor
2
3  implicit none
4  integer, parameter :: n = 9     ! bei Bedarf anpassen
5  real, dimension(n) :: v         ! Vektor v
6  integer :: i
7
8  ! Teile des Formatbeschreibers, keine haendische Anpassung notwendig
9  character (len=20) :: fb2 = '(2X,F5.3))'
10 character          :: fb1
```

```
character (len=40) :: fb

if (n <= 0 .or. n > 9) stop "n␣kann␣nur␣Werte␣von␣1␣bis␣9␣annehmen"

! Zusammensetzung des Formatbeschreibers
! funktioniert nur fuer n = 1 .. 9
fb1 = achar(iachar('0')+n)
fb  ="("//fb1//fb2

write(*,*) fb

do i = 1, n
 v(i) = 1.0/real(i)
end do

! Bildschirmausgabe des Vektors
write(*,fb) v

end program vektor
```

Sollen auch größere Werte von n zugelassen werden, z.B. n zwischen 1 und 999999, so kann man sich auch hier mittels Zeichenkettenverarbeitung einen passenden Formatbeschreiber konstruieren.
Beispielprogramm:

```
program vektor

implicit none
integer, parameter :: n = 55    ! bei Bedarf anpassen, Zahlen zw. 1 und 999999
real, dimension(n) :: v         ! Vektor v
integer :: i

! Teile des Formatbeschreibers, keine haendische Anpassung mehr noetig
character (len=20) :: fb2 = '(2X,F5.3))'
character (len=10) :: fb1
character (len=40) :: fb

if ( n < 1 .or. n > 999999 ) stop "n␣muss␣im␣Bereich␣zw.␣1␣und␣999999␣liegen!"
! Zusammensetzung des Formatbeschreibers
! funktioniert nur fuer n von 1 .. 999999 wg. I6 (6 Stellen)
write(fb1,'(I6)') n

fb  ="("//trim(adjustl(fb1))//fb2
write(*,*) fb

do i = 1, n
 v(i) = 1.0/real(i)
end do

! Bildschirmausgabe des Vektors
write(*,fb) v

end program vektor
```

Von zentraler Bedeutung bei der Erstellung des Formatbeschreibers für diesen allgemeineren Fall ist, dass mittels eines `internal writes` der Zahlenwert von `n` in die entsprechende Zeichenkette `fb1` umgewandelt wird

```
write(fb1,'(I6)') n
```

Um evtl. noch vorhandene Leerzeichen aus der Zeichenkette `fb1` zu entfernen, wird `trim(adjustl(fb1)` eingesetzt, bevor dieser Teilstring mit den restlichen Teilstrings zur Formatbeschreiberkette zusammengesetzt wird.

9.12 Das `where`-Konstrukt (Fortran 90/95)

Soll auf ein Datenfeld als Ganzem zugegriffen werden, aber dabei bestimmte Umformungen oder mathematische Funktionen nur durchgeführt werden, wenn bestimmte Vorbedingungen erfüllt sind, müsste man in Fortran 77 neben den `do`-Schleifen zum Durchlaufen der Indexbereiche der Datenfelder zusätzlich `if`-Abfragen einbauen. Fortran 90/95 bietet hier die komfortable Lösung über ein `where`-Konstrukt.
Das where-Konstrukt in Fortran 90 hat die Syntax

```
where ( < Zugriffsbedingung auf ein Datenfeld > )

    < Name eines Datenfeldes > = < arithmetischer Ausdruecke mit dem
    Datenfeldnamen >

else where

    Alternativen, wenn die Zugriffsmaskenbedingung bei
    einzelnen Komponenten nicht erfuellt ist

end where
```

Das `where`-Konstrukt lässt sich analog zu den `if`-Konstrukten und den `do`-Schleifen mit einem Namen versehen (benanntes `where`-Konstrukt).

```
<Name des where-Konstrukts >: where (...)

    ...
    else where <Name des where-Konstrukts>
    ...

end where <Name des where-Konstrukts>
```

Ebenfalls lässt sich ähnlich der `if`-Konstruktion bei einer nicht notwendigen Alternative und einer einzigen notwendigen Anweisen die Konstruktion in einen Einzeiler verkürzen.

```
where ( < Zugriffsmaske auf ein Datenfeld > ) Anweisung
```

Fortran 95 bietet die erweiterte Möglichkeit in Analogie zu dem `if then - else if - else if ... - else - end if`-Konstrukt, weitere `else where` mit anderen Bedingungen vor dem obigen `else where` dazwischenzuschalten, so dass immer eine der angegebenen Anweisungsblöcke auf die Komponenten des Datenfeldes angewandt werden, je nachdem, welche der Bedingungen wahr ist.
Beispielprogramm:

```
program where_konstrukt

implicit none
real, parameter :: nan_wert = -99999.9    ! Der Wert der statt nan
                                          ! (not a number) verwendet wird
real, dimension (4,3) :: x, y, z
integer :: i, j

do j = 1, 3                              ! Initialisierung des Datenfeldes
 do i = 1, 4
  x(i,j) = real(i)+0.1*real(j)
 end do
end do

10 format(3(3X,ES14.5))      ! Formatbeschreiber fuer die Matrix
                             ! immer 3 Werte in einer Zeile
5 format(3(3X,F4.1))

write(*,*) 'urspruengliche Matrix:'
write(*,5) ( (x(i,j), j=1,3), i=1,4 )
write(*,*)

! von jeder Komponente der Matrix wird 3.0 abgezogen
! Fortran 90/95 - Syntax
x = x - 3.0
write(*,*) 'von jeder Komponente wurde 3.0 abgezogen:'
write(*,5) ( (x(i,j), j=1,3), i=1,4 )
write(*,*)

! von jeder Komponente wird der natuerliche Logarithmus gebildet
! sofern mathematisch zulaessig, andernfalls wird die Komponente
! auf den Wert nan_wert gesetzt

! Fortran 77 - Syntax

do j = 1, 3
 do i = 1, 4
  if ( x(i,j) .GT. 0.0 ) then
   y(i,j) = log(x(i,j))
  else
   y(i,j) = nan_wert
  end if
 end do
end do
write(*,*) 'Bedingte logarithmische Umformung (Fortran 77)'
write(*,10) ( (y(i,j), j=1,3), i=1,4 )
write(*,*)

```

```
49 ! Fortran 90/95 - Syntax
50
51 where ( x > 0.)
52   z = log(x)
53 else where
54  z = nan_wert
55 end where
56
57 write(*,*) 'Bedingte␣logarithmische␣Umformung␣(where-Konstrukt␣Fortran␣90)'
58 write(*,10) ( (z(i,j), j=1,3), i=1,4 )
59 write(*,*)
60
61 end program where_konstrukt
```

9.13 Ein wichtiger Hinweis zum Umgang mit Datenfeldern in Fortran 90/95

Was beim Betrachten der Beispielprogramme auffällt, ist, dass die Fortran 90/95 - Erweiterungen im Umgang mit mehrdimensionalen Datenfeldern auf die explizite Programmierung von Schleifen zum Durchlaufen der Indexbereiche, wie sie in FORTRAN 77 notwendig sind, nicht mehr benötigen.
Als Weiterentwicklung von FORTRAN 77 sind die Befehle von Fortran 90/95 im Umgang mit gesamten Datenfeldern so konstruiert, dass sie dem Compiler auf Mehr - Prozessor - Maschinen und Vektorrechnern die Möglichkeit geben, Programmteile zu parallelisieren und zu vektorisieren und damit Ihrem Programmcode zu optimieren. Verwenden Sie deshalb in Ihren Programmen, immer, wenn dies möglich ist, die Fortran 90/95-Befehle, die es erlauben, am Stück gesamte Datenfeldern oder Unterdatenfelder mit einem Befehl zu verarbeiten (z.B. kann man mathematische Funktionen und Ausdrücke auf ein gesamtes Datenfeld oder Unterdatenfeld anwenden und die speziellen Befehle für Datenfelder wie z.B. `matmul`, `dot_product`, `transpose` einsetzen). Dadurch erlauben Sie dem Compiler eine an die jeweilige Rechnerarchitektur angepasste Optimierung. Zusätzlich gibt es natürlich rechner- und compilerspezifische Optimierungsmöglichkeiten, die sie mit Hilfe der Manuals evtl. weiter ausschöpfen können, wenn Sie Ihren Code auf einem bestimmten Rechner laufen lassen wollen.

9.14 Untersuchungsfunktionen (engl. *inquiry functions*) für Datenfelder

Beispielprogramm:

```
1 program test_array
2
3 implicit none
4 real, dimension (-5:4,3) :: a = 0.0
5 integer, dimension(2) :: lindex, uindex
6
7 write(*,*) 'shape(a)␣␣␣=␣', shape(a)
```

```
write(*,*)
write(*,*) 'size(a)␣␣␣␣=␣', size(a)
write(*,*)
write(*,*) 'lbound(a)␣␣␣=␣', lbound(a)
write(*,*)
write(*,*) 'ubound(a)␣␣␣=␣', ubound(a)
write(*,*)
lindex = lbound(a)
write(*,*) 'Der␣kleinste␣Zeilenindex␣ist:␣', lindex(1)

end program test_array
```

9.15 Dynamische Speicherallokierung für Datenfelder (Fortran 90/95)

Falls bei Umsetzung einer Programmieraufgabe nicht bekannt ist, wieviel Elemente ein Datenfeld umfassen wird, so bietet Fortran 90/95 (noch nicht Fortran 77) die Möglichkeit, den für ein Datenfeld benötigten Speicher dynamisch zu verwalten, d.h. den Speicher erst im Anweisungsteil für die Datenfelder zu reservieren, sobald dieser benötigt wird (dynamische Speicherallokierung) und den dynamisch belegten Speicher wieder freizugeben (deallokieren) sobald dieser nicht mehr benötigt wird. Auf diese Art und Weise können Sie den Speicher Ihres Rechners optimal nutzen, es treten keine Lücken im Speicher auf wie dies evtl. bei einer statischen (im Deklarationsteil) überdimensionierten Speicherreservierung der Fall sein würde.
Mit der dynamischen Speicherallokierung haben Sie die Möglichkeit bei Bedarf mit der Anzahl der zu verarbeitenden Komponenten flexibel bis an die Grenzen des Hauptspeichervolumens zu gehen.
Beispielprogramm:

```
! Beispiel zur dynamische Deklaration von Datenfeldern

program dynamische_deklaration

implicit none
real, allocatable, dimension (:,:) :: df    ! dynamisches 2-dim. Datenfeld
integer                            :: n, m ! die spaeter festzulegende
                                           ! Zeilen und Spaltenanzahl
integer                            :: alloc_status, dealloc_status
                                           ! Variablen zur Fehlerbehandlung
integer                            :: i,j   ! Schleifenvariablen

! Information des Anwenders, Einlesen von n, m

write(*,*)  'Demo-Programm␣zur␣dynamischen␣Speicherallokierung'
write(*,*)  'Betrachtet␣wird␣eine␣Matrix␣(2-dim.␣Datenfeld)'
write(*,*)
write(*,*)  'Bitte␣geben␣Sie␣die␣gewuenschten␣Werte␣ein'
write(*,10) 'Anzahl␣an␣Zeilen:␣␣␣'; read(*,*) n
write(*,10) 'Anzahl␣an␣Spalten:␣'; read(*,*) m
write(*,*)
```

```
10 format(1X,A$)

! Es muss nun der benoetigte Speicherplatz fuer das Datenfeld
! allokiert werden

allocate(df(n,m),stat=alloc_status)
write(*,*)
write(*,*) 'Rueckgabewert von allocate:', alloc_status
write(*,*)

! Die Matrixkomponenten werden mit einem numerischen Wert
! der dem Zeilenindex als Vorkommastelle und dem
! Spaltenindex als Nachkommastelle belegt

do j = 1, m
 do i = 1, n
  df(i,j) = real(i) + 0.1*real(j)
 end do
end do

! Bildschirmausgabe zur Kontrolle

write(*,*) 'Die Matrix sieht nun so aus:'
write(*,*)
do i = 1, n
 write(*,*) (df(i,j), j=1,m)
end do

! Falls das Datenfeld nicht mehr benoetigt wird, kann der Speicherplatz
! nun deallokiert werden

deallocate(df,stat=dealloc_status)
write(*,*)
write(*,*) 'Rueckgabewert von deallocate:', dealloc_status
write(*,*)

end program dynamische_deklaration
```

Kapitel 10

Unterprogramme

Steht man vor einer komplexeren Programmieraufgabe, so geht man am besten nach dem logischen Prinzip des Top-Down-Entwurfs vor. Ein wesentlicher Schritt besteht darin, das generalisierte Problem in einzelne Teilschritte zu zerlegen. Diese Teilaufgaben können wiederum sukzessive in einzelne weitere Unteraufgaben verfeinert werden.
Fortran besitzt eine spezielle Struktur, um diese logische Untergliederung als Programmcode realisieren zu können. So kann jeder einzelne, in sich logisch geschlossene Teilschritt als eigenes Unterprogramm entwickelt und unabhängig vom restlichen Programmcode getestet werden.
Die einzelnen Unterprogramme bilden in sich geschlossene separate Programmeinheiten. Der Informationsaustausch zwischen den einzelnen Programmteilen eines Fortran - Programms, zu denen auch das Hauptprogramm gehört und z.B. einem Unterprogramm ist über Schnittstellen geregelt.
Die Übersichtlichkeit modular erstellter Programme wissen all diejenigen zu schätzen, die fremdentwickelte Programme betreuen und erweitern müssen.
An der Uni kommt es z.B. recht häufig vor, dass bereits vorhandene Programme von Doktoranden und Diplomanden verwendet, ausgebaut und umgebaut werden sollen. Falls Sie hier ein schlecht dokumentiertes Spaghetti-Code-Programm erben sollten, werden Sie mit Sicherheit wenig Freude haben und einen Großteil Ihrer Zeit darauf verwenden müssen, herauszufinden, was das ererbte Programm eigentlich tut. Der nächste Schritt, Änderungen einzubauen und zu prüfen, ob die Algorithmen wirklich korrekt arbeiten, ist nahezu immer ein schier hoffnungsloses Unterfangen.
Ein persönlicher Tip: Falls nach Ansicht Ihres Betreuers die Ihnen angebotene Diplomarbeit oder Dissertation zu einem grossen Teil daraus bestehen sollte, ein „bewährtes“ Spaghetti-Code-Programm eines ihrer Vorgänger umzubauen - gehen Sie lieber - solange noch Zeit ist - an einen anderen Lehrstuhl, der Ihnen die Möglichkeit zu einer echten Forschungsarbeit bietet.
Wenn Sie Programme erstellen, können Sie sich Ihre Arbeit durch den Einsatz von Unterprogrammen erheblich erleichtern! Prinzipiell kennt Fortran zwei Arten von Unterprogrammen:

```
subroutine function
```

10.1 Vorteile von Unterprogrammen

1. **Unabhängiges Entwickeln und Testen der Teilaufgaben**

   Jede einzelne Teilaufgabe kann in Code umgesetzt und als unabhängiger Teil compiliert werden. Die Tests der einzelnen Programmeinheiten können unabhängig von anderen Programmeinheiten stattfinden. Dadurch wird die Wahrscheinlichkeit für das Auftreten unerwünschter Interferenzen zwischen einzelnen Programmteilen reduziert.
   Die Programmentwicklung - und das Programm - werden übersichtlicher, klarer strukturiert und weniger fehleranfällig.

2. **Wiederverwendbarer Code**

   Die in sich logisch abgeschlossenen Unterprogrammeinheiten lassen in der Regel im aktuellen Programm oder in anderen Programmen orginalgetreu oder leicht modifiziert wieder einsetzen.
   Die Entwicklungs- und Testzeiten sowie die Programme werden kürzer.

3. **Klare Struktur**

   Zwischen den einzelnen Programmeinheiten ist die Werteübergabe klar geregelt. Lokale Variablen sind nur in der Programmeinheit gültig, in der sie deklariert wurden (sogenanntes data hiding). Die lokalen Variablen können auch nur innerhalb ihrer Programmeinheit verwendet und modifiziert werden. Die Wahrscheinlichkeit, dass in Folge von Programmänderungen an einer Stelle des Codes unbeabsichtigte Folgen an anderen Stellen des Programms auftreten, wird geringer.
   Die Gefahr von „Programmierunfällen“ wird vermindert.
   Die Fehlersuche wird erleichtert.

10.2 `subroutine` - Unterprogramme

Diese werden von einer anderen Programmeinheit über

`call` <Name der Subroutine >

bzw. über

`call` <Name der Subroutine > (Liste der aktuellen Parameter)

aufgerufen. Die zweite Version wird verwendet, wenn zwischen einzelnen Programmeinheiten (z.B. zwischen dem Hauptprogramm und dem Unterprogramm) Informationen ausgetauscht werden sollen.
Die **subroutine** bildet eine eigene, in sich geschlossene Programmeinheit mit folgender Struktur:

```
subroutine <Name der Subroutine> (Liste der formalen Parameter)
```

```
        implicit none
        ! Datentypdeklaration der formalen Parameter
        ...
        ! Datentypdeklaration der lokalen Variablen
        ...
        ! Anweisungsteil der Subroutine
        ...
        ...

    return end subroutine <Name der Subroutine>
```

In der `Liste der formalen Parameter` werden als Platzhalter Namen von Variablen aufgeführt. Unmittelbar nach der Kopfzeile des Unterprogramms müssen die formalen Parameter in gewohnter Weise mit Namen und zugehörigem Datentyp deklariert werden.
Beim Aufruf der Subroutine aus einer anderen Programmeinheit heraus muss die Reihenfolge der an das Unterprogramm übergebenen Werte bzw. Variablen (d.h. die Liste der aktuellen Parameter) vom Datentyp und der Zuweisungsreihenfolge her genau mit der Liste der formalen Parameter übereinstimmen.

10.3 Ein einfaches Anwendungsbeispiel mit einer `subroutine`

Beispielprogramm:

```
1  program unterprogramm_demo1
2
3  implicit none
4  real :: a, b, c
5
6  write(*,*) 'Berechnet die Hypotenusenlaenge eines rechtwinkligen Dreiecks'
7  write(*,*) 'Bitte die Laenge der Seiten eingeben:'
8  read(*,*)  a, b
9
10 call hypotenuse(a,b,c)  ! a, b wurden interaktiv eingelesen
11         ! c, die Hypotenusenlaenge wird berechnet
12 write(*,*) 'Die Laenge der Hypotenuse betraegt: ', c
13
14 call hypotenuse(3.0,4.0,c)  ! Statt Variablen, zu denen
15         ! Werte einlesen wurden,
16         ! koennen auch fuer die beiden
17         ! ersten formalen Parameter in der
18         ! Subroutine direkt Werte uebergeben
19         ! werden
20 write(*,*)
21 write(*,*) 'Aufruf der Subroutine ueber call hypotonuse(3.0,4.0,c)'
22 write(*,*) 'Die Laenge der Hypotenuse bei a=3.0, b= 4.0 betraegt: ', c
23
24 end program unterprogramm_demo1
25
26
27 subroutine hypotenuse(x, y, z) ! Ein Unterprogramm des Typs subroutine
28 ! Deklaration der formalen Parameter
29 implicit none
```

```
30 real, intent(in)  :: x, y
31 real, intent(out) :: z
32
33 ! Deklaration der lokalen Variablen
34 real :: wert
35
36 ! Anweisungsteil der Subroutine
37 wert = x*x + y*y
38 z = sqrt(wert)
39 return
40 end subroutine hypotenuse
```

Die Subroutine hypotenuse wird im Hauptprogramm erstmals über die Anweisung

```
call hypotenuse(a,b,c)
```

aufgerufen. Zuvor wurden den Variablen a und b durch die Anweisung

```
read(*,*)  a, b
```

interaktiv durch den Anwender Werte zugewiesen. Beim Aufruf der Subroutine werden diese Werte an die formalen Parameter `x` und `y` übergeben. In Fortran 90 und 95 wird bei der Deklaration der formalen Parameter im Unterprogramm durch das Attribut `intent(in)` die Richtung der Informationsübergabe gekennzeichnet

```
real, intent(in)  :: x, y
```

Das Attribut `intent(in)` kennzeichnet, dass die Information von der aufrufenden Programmeinheit an das Unterprogramm übergeben wird und damit die Richtung des Informationsflusses. Demententsprechend kennzeichnet das Attribut

```
real, intent(out) :: z
```

dass die Information im Unterprogramm berechnet und an die aufrufende Programmeinheit (in diesem Fall das Hauptprogramm) zurücküberreicht wird. Der Rücksprung zur aufrufenden Programmeinheit wird eingeleitet, sobald der Programmablaufzeiger ein `return` (oder die `end`-Anweisung eines Unterprogramms erreicht). In einem Unterprogramm können mehr als ein `return` stehen. Prinzipiell sollte jedoch zur Sicherheit vor der `end`-Anweisung eines Unterprogramms ein `return` eingefügt werden, weil es ab und zu Compiler gab (gibt), die unter Umständen ohne `return` falsche Rückgabewerte abgeliefert haben (abliefern würden?).

10.4 Das „pass by reference scheme“

Oder: was passiert beim Aufruf von Unterprogrammen?
Beim Aufruf von Unterprogrammen werden zur Informationsübermittlung Zeiger (engl. *pointer*) auf die Speicherplätze der Variablen im Hauptspeicher übergeben, an denen die in der Liste der aktuellen Parameter angelegten Speicherplätze der Variablen zu finden sind. Zum Beispiel sind im obigen Programm `unterprogramm_demo1` in der Anweisung

```
call hypotenuse(a,b,c)
```

als aktuelle Parameter `a`, `b` und `c` angegeben. Diese Variablen wurden im Deklarationsteil des Hauptprogramms als dem Datentyp `real` zugehörig vereinbart.
Die Kopfzeile des Unterprogramms beginnt mit

```
subroutine hypotenuse(x, y, z)
! Deklaration der formalen Parameter
implicit none
real, intent(in) :: x, y
real, intent(out) :: z
```

Im Unterprogramm erscheinen die formalen Parameter als `x`, `y`, `z`. Sowohl von der Reihenfolge als auch von den Datentypen her muss die Liste der aktuellen Parameter mit der Liste der formalen Parameter übereinstimmen.
Der Mechanismus des Informationsaustauschs zwischen aufrufender Programmeinheit und Unterprogramm wird als „**pass by reference scheme**“ bezeichnet. Das „pass by reference scheme“ stellt sich wie in Abbildung 10.1 gezeigt dar. Beim Aufruf eines Unterprogramms

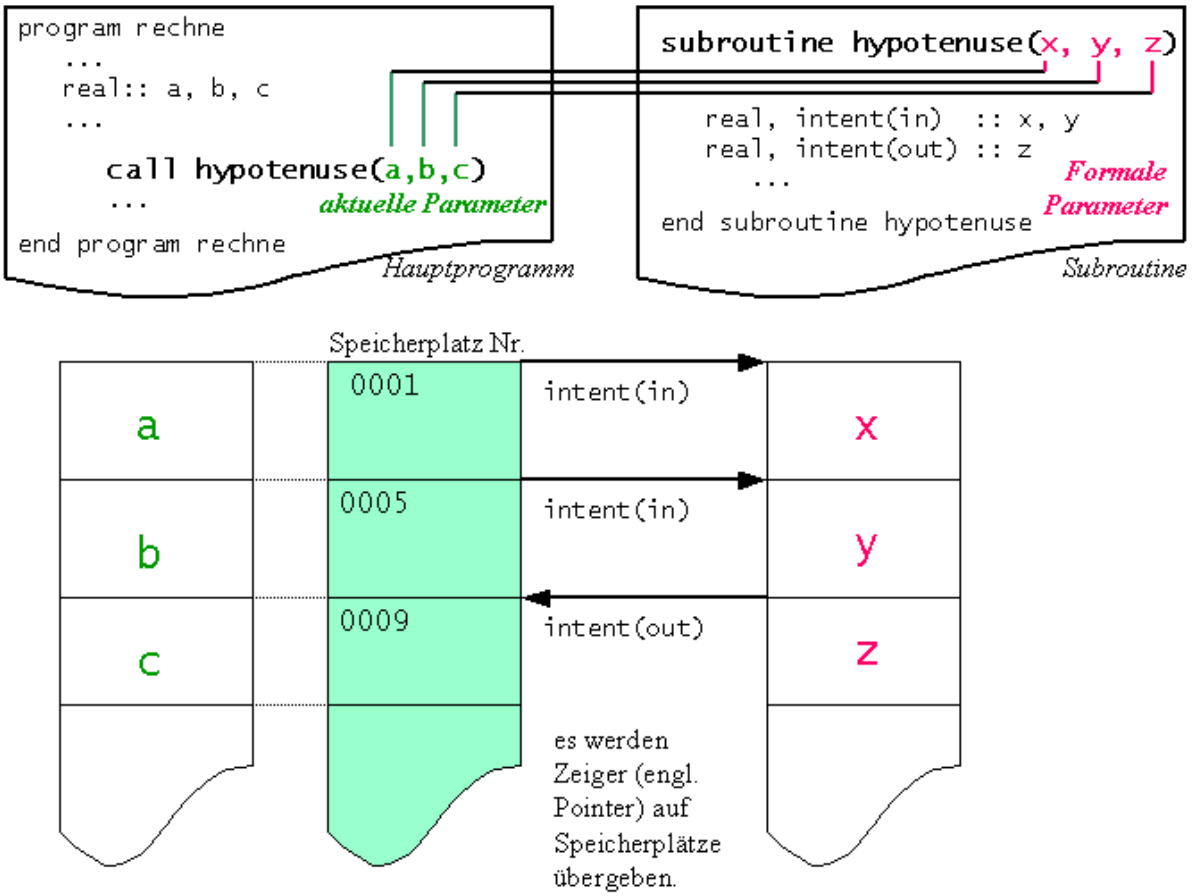

Abbildung 10.1: pass by reference scheme

werden also nicht die aktuellen Werte, sondern nur Zeiger (Pointer) auf die Speicherplätze übergeben, in denen die jeweiligen Werte abgelegt sind. Da Werte vom Datentyp `real`

im allgemeinen 4 Bytes im Memory belegen, wird für den formalen Parameter `x` der Subroutine beispielsweise ein Zeiger auf den Speicherplatz 0001 übergeben. Die Speicherplätze 0001, 0002, 0003 und 0004 werden benötigt, um den `real`-Wert für die Variable `a` abzulegen, demzufolge wird an den formalen Parameter `y` beim Aufruf der Subroutine ein Zeiger überreicht, der dem Speicherplatz 0005 zugeordnet ist und an dem der Speicherbereich für den aktuellen Wert `b` beginnt. Analog erfolgt die Zuordung zwischen dem formalen Parameter `z` und dem aktuellen Parameter `c`.
Im Diagramm bezeichnet die Richtung des Pfeils nach rechts, dass vom dem Unterprogramm Information vom einem Speicherplatz gelesen werden sollen (`intent(in)`). Wird ein formaler Parameter mit dem Attribut (`intent(out)`) deklariert, wird von der Subroutine Information an dem Speicherplätzen abgelegt, auf den der entsprechende Pointer zeigt.
Soll durch den Aufruf eines Unterprogramms der Wert einer übergebenen Variablen durch den Unterprogrammaufruf verändert werden, so muss dementsprechend als Attribut `intent(inout)` gesetzt werden.
Ein Beispiel:
In einer Firma soll bei der Rechnungserstellung für treue Kunden noch 3 Prozent Rabatt von der Rechnungsendsumme abgezogen werden. Nach der Prüfung der Voraussetzungen, ob ein zusätzlicher Rabatt gewährt werden soll, erfolgt von einer Programmeinheit aus der Aufruf des Unterprogramms.

```
...
pruefe_rabattvorausssetzung: if (...) then

    call gewaehre_3_prozent_rabatt(betrag)
    write(*,'(1X,A)') 'Als treuen Kunden gewaehren wir Ihnen &
    &3 Prozent Rabatt.'
    else

        write(*,*)

    end if pruefe_rabattvoraussetzung

write(*,'(1X,A)') 'Vielen Dank fuer Ihren Einkauf!'
write(*,'(//1X,A,F12.2,1X,A5)') 'Bitte ueberweisen Sie ', &
betrag, 'Euro.'
...
```

Dementsprechend könnte die Subroutine lauten:

```
subroutine gewaehre_3_prozent_rabatt(summe)
implicit none
real, intent(inout) :: summe
summe = summe*0.97
return
subroutine gewaehre_3_prozent_rabatt(summe)
```

Hier ist es notwendig, dass als Attribut `intent(inout)` bei der Deklaration des formalen Paramters summe steht. Beim Aufruf des Unterprogramms

```
call gewaehre_3_prozent_rabatt(betrag)
```

wird ein Zeiger auf den Speicherplatz von `betrag` übergeben („pass by reference scheme“). Von dort liest das Unterprogramm den Wert aus und verarbeitet diesen als Wert der Variablen `summe` weiter. Erreicht der Programmablaufzeiger im Unterprogramm die `return`-Anweisung, wird der aktuelle Wert der Variablen `summe` in den Speicherplatz von `betrag` geschrieben. Daraufhin wird das Programms mit dem sich an die `call`-Anweisung anschließenden Befehl fortgesetzt.
Steht als Attribut eines formalen Parameters `intent(in)`, ist es dem Unterprogramm untersagt, den dort gespeicherten Wert zu verändern. Von diesem Speicherplatz dürfen vom Unterprogramm ausschließlich Information gelesen werden.
Wird ein formaler Parameter mit dem Attribut `intent(out)` deklariert, darf dementsprechend das Unterprogramm am Speicherplatz dieser Variablen nur Information ablegen und dadurch Informationen an die aufrufende Programmeinheit weitergeben. Das Unterprogramm darf keinesfalls versuchen von einem als `intent(out)` deklarierten formalen Parameter Information zu erhalten.
Bei Widersprüchen zwischen der Deklaration der formalen Parameter und dem Unterprogrammcode geben Fortran 90/95-Compiler oft nur bei Aktivierung bestimmter Compileroptionen eine Fehlermeldung aus und der Compiliervorgang wird abgebrochen. Ohne diese strengen Compileroptionen zum Datentypabgleich kann es passieren, dass ein Programm scheinbar fehlerfrei compiliert wird. Startet man das Executable, treten in diesem Fall jedoch Fehler im Programmablauf auf, deren Ursache in Typ-Mismatches zwischen den aktuellen Parametern beim Unterprogrammaufruf und der Datentypdeklaration bei den formalen Parametern des Unterprogramms liegt.
Fazit: Durch die Implementierung definierter Schnittstellen zum Informationsaustausch zwischen Hauptprogramm und Unterprogramm und durch die Verwendung nur lokal im Unterprogramm gültiger Variablen lassen sich unerwünschte Seiteneffekte durch ein versehentliches Verändern von Variablenwerten vermeiden. Ein **Unterprogramm** stellt eine Art in sich gekapselte Untereinheit dar („**black box**“ oder „**information hiding**“), deren Interaktion mit dem Hauptprogramm bzw. der aufrufenden Programmeinheit allein über die definierte Schnittstelle im Unterprogrammkopf (die Liste der formalen Parameter) geregelt ist.

10.5 Die FORTRAN 77 - Syntax

Fortran 90/95 stellt eine Fortentwicklung von FORTRAN 77 dar. Das Konzept mit dem `intent`-Attribut bei der Deklaration der formalen Parameter eines Unterprogramms war in FORTRAN 77 noch nicht vorhanden. Die erweiterte Fortran 90/95 - Syntax steigert die Strukturqualität und die Übersichtlichkeit der Programme und reduziert die Fehleranfälligkeit.
Ein Vorab-Hinweis: Eine weitere Verbesserung der Qualität beim Datenaustausch stellen in Fortran 90/95 die Module als Weiterentwicklung der veralteten `COMMON`-Blöcke dar.

10.6 `function` - Unterprogramme

Anders als eine `subroutine` besitzt eine `function` genau einen Rückgabewert. Man kann sich dies ähnlich einer mathematischen Funktion vorstellen, bei der genau ein Funktions-

wert aus dem/den Variablenwert/en berechnet wird. Vor einer `function` muss der Datentyp des Rückgabewertes deklariert werden.

```
<Datentyp d. Rueckgabewerts> function <Name der Function> (Liste
der formalen Parameter)
implicit none
! Datentypdeklaration der formalen Parameter
..., intent(in) :: ... ! ausschliesslich intent(in)
..., intent(in) :: ... ! fuer die formalen Parameter
...
! Datentypdeklaration der lokalen Variablen
...
! Anweisungsteil der Function
...
...
<Name der Function> = ... ! abschliessende Wertzuweisung
! des Rueckgabewertes
return
end function <Name der Function>
```

Will man eine `function` von einer anderen Programmeinheit aus aufrufen, so gibt man nur den Namen der `function` an. Jedoch muss bei der Variablen-Deklaration in der aufrufenden Programmeinheit der Datentyp der Funktion mit aufgelistet werden, damit der Datenaustausch zwischen zwischen aufrufender Programmeinheit und `function`-Unterprogramm reibungslos vonstatten gehen kann.
In der aufrufenden Programmeinheit muss expliziert im Deklarationsteil (am besten an dessen Ende) die Funktion zusammen mit dem dazugehörigen Datentyp deklariert werden.

```
! In der aufrufenden Programmeinheit muss
! (am besten am Ende des Deklarationsteils) stehen:

<Datentyp der function> :: <Name der function>


....
! innerhalb der aufrufenden Programmeinheit kann nun die
! function ueber Ihrem Namen bei entsprechender Angabe der
! aktuellen Parameter aufgerufen werden
```

Um diesen noch recht abstrakten Sachverhalt zu veranschaulichen, soll die Berechnung der Länge der Hypotenuse eines rechtwinkligen Dreieck diesmal nicht mit einer `subroutine`, sondern mit einer `function` realisiert werden.
Da bei der Berechnung der Hypotenusenlänge der Wert der den rechten Winkel einschließenden Dreiecksseiten nicht verändert wird und sich die Länge der Hypotenuse aus der Wurzel der Summe der Quadrate der beiden Seitenlägen ergibt (mathematisch gesehen eine eindeutige Zuordnung), ist es sinnvoll mit einer `function` zu arbeiten. Das obige Programm wurde hierfür einfach umgeschrieben.
Beispielprogramm:

```
program unterprogram_demo2
implicit none

real :: hypotenusen_laenge    ! der Datentyp des Rueckgabewertes
                              ! einer Function muss in der aufrufenden
                              ! Programmeinheit deklariert werden
real :: a, b, c

write(*,*) 'Berechnet die Hypotenusenlaenge eines rechtwinkligen Dreiecks'
write(*,*) 'Bitte die Laenge der Seiten eingeben:'
read(*,*)  a, b               ! a, b wurden interaktiv eingelesen

! c, die Hypotenusenlaenge wird berechnet
c = hypotenusen_laenge(a,b)

write(*,*) 'Die Laenge der Hypotenuse betraegt: ', c

c = hypotenusen_laenge(3.0,4.0)  ! Statt Variablen, zu denen Werte einlesen
                          ! wurden, koennen auch fuer die beiden
                          ! ersten formalen Parameter der
                          ! Funktion direkt Werte uebergeben werden
write(*,*) 'Die Laenge der Hypotenuse bei a=3.0, b= 4.0 betraegt: ', c
end program unterprogram_demo2


real function hypotenusen_laenge(x, y)
! Deklaration der formalen Parameter
implicit none
real, intent(in)  :: x, y
! Deklaration der lokalen Variablen
real :: wert

wert = x*x + y*y
hypotenusen_laenge = sqrt(wert)
return
end function hypotenusen_laenge
```

Wie wir sehen, stellt die letzte Zeile der `function` vor dem `return` (der Rücksprunganweisung) zur aufrufenden Programmeinheit eine Wertzuweisung dar. Hier wird der Wert von `sqrt(wert)` an den Speicherplatz der `real`-Variablen `hypotenusen_laenge` geschrieben und beim Rücksprung zur aufrufenden Programmeinheit der Zeiger (engl. *pointer*) auf diesen Speicherplatz übermittelt. Diese Art der Informationsübertragung zwischen Unterprogramm und aufrufender Programmeinheit wird wiederum „pass by reference scheme“ bezeichnet.
Ein weiteres einfaches Beispiel für den Einsatz einer `function`:

```
program parabel
implicit none
integer :: i
real    :: a, b, c
real    :: x
real    :: f

write(*,*) 'Das Programm berechnet Werte zu'
```

```
9  write(*,*) ' f(x) = a*x**2 + b*x + c'
10 write(*,*) 'Bitte geben Sie die Werte fuer a, b und c ein:'
11 read(*,*) a, b, c
12
13 write(*,*) 'Wertetabelle fuer f(x) fuer x von 1 .. 10:'
14 write(*,*) '   x          f(x)'
15 write(*,*) '-------------------'
16
17 do i = 0, 10
18   x = real(i)
19   write(*,'(1X,F5.2,3X,F6.2)') x, f(x,a,b,c)
20 end do
21 write(*,*)
22 write(*,*) 'Aufruf der Funktion f(x,a,b,c) mit direkter Werteuebergabe'
23 write(*,*) 'z.B.  f(2.0,1.0,-1.0,-1.0) = ', f(2.0,1.0,-1.0,-1.0)
24 end program parabel
25
26
27 real function f (x,a,b,c)
28 implicit none
29 real, intent(in) :: x, a, b, c
30 f = a * x * x + b * x + c
31 return
32 end function f
```

In diesem Beispiel treten sowohl im Hauptprogramm (beim Funktionsaufruf) als auch in der `function` die Variablen (`x,a,b,c`) sowohl als aktuelle (im Hauptprogramm) als auch als formale Parameter (im Unterprogramm-Kopf) auf. Dies ist durchaus erlaubt und sinnvoll, wenn dadurch die Arbeitsweise des Gesamtprogramms transparenter wird. Genausogut wäre es möglich, den formalen Parametern im Unterprogramm ganz andere Namen als den Variablen im Hauptprogramm zu geben.
Entscheidend für eine fehlerfreie Compilierung ist allein, dass sowohl Reihenfolge als auch Datentypen beim Unterprogrammaufruf mit den formalen Parametern im Unterprogrammkopf 1:1 übereinstimmen. Dann können Sie sicher sein, dass Ihr Programm fehlerfrei compiliert wird.
Achtung: Natürlich gilt auch im Zusammenhang mit Unterprogrammen, dass ein Compiler nur Fehler auf Syntax-Ebene und nicht auf der logischen Algorithmus-Ebene finden kann. Es bleibt wie immer Ihre Aufgabe sicherzustellen, dass Ihr Programm keine logischen Fehler enthält. Beispiel für einen logischen Fehler in einem Unterprogramm, den nur Sie als Mensch finden und eliminieren können:
Hätten Sie z.B. wie oben den Wert einer quadratischen Parabel mit

```
a = 1.0, b = -1.0, c = -3.0
```

also den Wert von `x**2 - x - 3` an der Stelle `x = 2.0` berechnen wollen und versehentlich in einer falschen Reihenfolge die Liste der aktuellen Parameter statt mit `f(x,a,b,c)` - wie es aufgrund der Deklaration des Unterprogramms richtig gewesen wäre - fälschlicherweise `f(a,b,c,x)` in den Programmcode geschrieben, so stimmen zwar die Anzahl, Reihenfolge und Datentypen der formalen und aktuellen Parameter exakt überein, jedoch liegt ein schwerwiegender logischer Fehler vor, denn aufgrund des ***pass by reference schemes*** wird im Zahlenbeispiel statt (wie es mathematisch korrekt wäre)

```
f(2.0,1.0,-1.0,-3.0) =
1.0 * 2.0 * 2.0 + (-1.0) * 2.0 + (-3.0)
= -1.0
```

nun aufgrund der falsch gewählten Reihenfolge in der Liste der aktuellen Parameter

```
f(1.0,-1.0,-3.0,2.0)
(-1.0) * 1.0 * 1.0 + (-3.0) * 1.0 + 2.0
= -2.0
```

und damit ein falscher Zahlenwert berechnet.
Auf Compilerebene liegt kein syntaktischer Fehler vor, dennoch ist das Programm solange unbrauchbar, bis der logische Fehler in der Reihenfolge der aktuellen Parameter gefunden und beseitigt worden ist. Hier handelt es sich um einen typischen logischen Fehler, der nur von einem Menschen durch den Vergleich der Programmresultate mit unabhängig ermittelten Ergebnissen (je nach Anforderungsgrad: z.B. durch Kopfrechnen, Papier und Bleistift, Computeralgebrasystem, wissenschaftliche Veröffentlichungen der Konkurrenz) gefunden und eliminiert werden kann.
Merke: Das Beispiel zeigt, dass beim Aufruf von Unterprogrammen mit mehr als einem formalen Parameter eine sorgfätige Prüfung der Liste der aktuellen Parameter auf die logisch richtige Reihenfolge angeraten ist, um evtl. Rechenfehler aufgrund einer logisch falschen Zuordnung in der Reihung von formalen und aktuellen Parametern auszuschließen.

10.7 Übergabe von Datenfeldern (engl. *arrays*) an Unterprogramme

Datenfelder (auch engl. *arrays* oder indizierte Variablen genannt) werden eingesetzt, wenn viele gleichartig strukturierte Daten mit einem Programm verarbeitet werden sollen.
Wie wir gesehen haben, wird beim Unterprogrammaufruf ein Zeiger (engl. *pointer*) auf den Beginn des Speicherplatzes übergeben, an denen der erste Wert des Feldes beginnt. Da in Fortran die Werte eines Arrays sequentiell (der Reihe nach) und nach der im Standard vereinbarten Indexreihenfolge (z.B. 2-dimensionale Datenfelder spaltenweise nacheinander) abgespeichert werden, muss an das Unterprogramm neben dem Namen des Feldes (dem aktuellen Parameter) zusätzlich die Information übermittelt werden, welche Gestalt das Datenfeld aufweist.
Zum Beispiel braucht bei einem eindimensionalem Array (einem Vektor) das Unterprogramm eine Mitteilung darüber, bis zu welchem maximalen Komponentenindex das Unterprogramm gehen darf. Dies ist notwendig, um zu vermeiden, dass aus dem Speicher Werte ausgelesen werden, die nicht mehr zu dem eindimensionalen Datenfeld gehören.
In der Regel wird an des Unterprogramm neben dem Namen des eindimensionalen Feldes zusätzlich die Anzahl der relevanten Komponenten eines Vektors übergeben.
Ein einfaches Beispiel, das noch das Risiko einer Bereichsüberschreitung bei der Ausgabe des Datenfeldes in sich birgt, ist z.B. `vektor_uebergabe`:

```
1 program vektor_uebergabe
2 integer, parameter    :: nmax = 3
3 real, dimension(nmax) :: vektor
```

```
integer                 :: i

do i = 1, nmax
        vektor(i) = real(i)
end do

call ausgabe(nmax,vektor)   ! Ausgabe der Komponenten des Vektors
call ausgabe(2,vektor)      ! Ausgabe der beiden ersten Komponenten
call ausgabe(4,vektor)      ! Achtung: Fehler
                            ! Bereichsueberschreitung: 4 > nmax
end program vektor_uebergabe


subroutine ausgabe(n,v)

! Achtung: Subroutine noch unzulaenglich programmiert
! Bereichsueberschreitung des Datenfeldes bei der Ausgabe moeglich

implicit none
! Deklaration der formalen Parameter
integer, intent(in)                 :: n
real, dimension(n), intent(in)      :: v
! Deklaration der lokalen Variablen
integer :: i

write(*,*)
write(*,*) 'Es werden die',n,'ersten Komponenten des Vektors ausgegeben.'
write(*,*)
do i = 1, n
        write(*,*) v(i)
end do
write(*,*)
return
end subroutine ausgabe
```

Im obigen Programm wurde die `subroutine ausgabe` dreimal nacheinander aufgerufen:

```
call ausgabe(nmax,vektor)
call ausgabe(2,vektor)
call ausgabe(4,vektor)
```

Beim ersten Aufruf wird die Anzahl der auszugebenden Komponenten `nmax` (die Gesamtanzahl der Feldkomponenten) übergeben. Beim folgenden Unterprogrammaufruf werden nur 2 der nmax(=3)-Komponenten angefordert und als Felddimension in der Unterroutine vereinbart. Dies ist durchaus zulässig, weil beim Unterprogrammaufruf nur ein Zeiger (pointer) auf den Beginn des Datenfeldes im Hauptspeicher übergeben wird und weil gleichzeitig der gewählte Wert 2 leiner als die Zahl der Datenfeldelemente bleibt.
Problematisch ist der 3. Aufruf.

```
call ausgabe(4,vektor)
```

Hier werden als aktuelle Parameter (`4,vektor`) den formalen Parametern (`n,v`) der `subroutine ausgabe` zugeordnet. Und damit wird das Unterprogramm aufgefordert, 4 Komponenten eines Datenfeldes auszugeben, welches nur 3 Komponenten umfasst.

Beim Aufruf des Unterprogramms wird der Zeiger auf den Beginn des Datenfeldes übergeben, im dem der Vektor v gespeichert ist. Was beim Unterprogrammaufruf nicht übergeben wird, ist die Information, wieviele Komponenten das Datenfeld umfasst. Dies führt dazu, dass ein falscher, nicht mehr zum Datenfeld gehöhrender, vierter Wert ausgeben wird. Der Compiler selbst bietet aufgrund des **pass by reference**-Mechanismus, (nämlich dass beim Unterprogrammaufruf nur ein Zeiger auf den Beginn eines Datenfeldes übergeben und nur noch auf die Gleichheit der Datentypen von aktuellem und formalen Parameter geprüft wird), im Unterprogramm evtl. Bereichsüberschreitungen festzustellen. Dies bleibt Aufgabe des Programmierers. Deshalb könnte eine korrigierte Version des obigen Programms wie folgt aussehen.
Beispielprogramm:

```
1  program vektor_uebergabe
2  integer, parameter     :: nmax = 3
3  real, dimension(nmax) :: vektor
4  integer                :: i
5
6  do i = 1, nmax
7          vektor(i) = real(i)
8  end do
9
10 call ausgabe(nmax,vektor,nmax)  ! Ausgabe der Komponenten des Vektors
11 call ausgabe(nmax,vektor,2)     ! Ausgabe der beiden ersten Komponenten
12 call ausgabe(nmax,vektor,4)     ! Achtung: Fehler
13                                 ! Bereichsueberschreitung: 4 > nmax
14 end program vektor_uebergabe
15
16
17 subroutine ausgabe(n,v,m)
18 ! Bereichsueberschreitung des Datenfeldes bei der Ausgabe
19 ! wird vermieden durch eine zusaeztliche Pruefung
20
21 implicit none
22 ! Deklaration der formalen Parameter
23 integer, intent(in)                :: n   ! Dimension des Datenfeldes
24 real, dimension(n), intent(in)    :: v   ! Name des Datenfeldes
25 integer, intent(in)                :: m   ! Anzahl der auszugebenden
26                                           ! Komponenten
27 ! Deklaration der lokalen Variablen
28 integer ::  m_local, i
29
30 m_local = m
31
32 if ( m_local > n ) then
33    write(*,*) '-----------------------------------------------------&
34 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣&-----------------'
35    write(*,*) 'Achtung:'
36    write(*,*) 'Anzahl␣der␣Komponenten␣im␣Datenfeld:␣', n
37    write(*,*) 'gewuenschte␣Anzahl:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣', m_local
38    write(*,*)
39    write(*,*) 'Die␣automatische␣Begrenzung␣der␣Anzahl␣der␣auszugebenden␣&
40 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣&Komponenten'
41    write(*,*) 'tritt␣in␣Kraft'
42    write(*,*) '-----------------------------------------------------&
```

```
43 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣&-----------------'
44     write(*,*)
45     m_local = n
46 end if
47
48 write(*,*)
49 write(*,*) 'Es␣werden␣die',m_local, &
50     'ersten␣Komponenten␣des␣Vektors␣ausgegeben.'
51 write(*,*)
52 do i = 1, m_local
53     write(*,*) v(i)
54 end do
55 write(*,*)
56 return
57 end subroutine ausgabe
```

Sollen **mehrdimensionale** Arrays zwischen einzelnen Programmeinheiten übergeben werden, muss man zunächst rekapitulieren, wie in Fortran 90/95 im Speicher Datenfelder ablegt werden. Mehrdimensionale Datenfelder werden intern dergestalt abgespeichert, dass zuerst der erste Index hochgezählt wird, dann erst der zweite und dann der dritte und so fort ... Bei einem zweidimensionalen Datenfeld (z.B. einer Matrix) entspricht dies einer **spaltenweisen** sequentiellen Sicherung der einzelnen Komponenten.
Liegt zum Beispiel eine n x m Matrix a mit n Zeilen und m Spalten vor, deren Komponenten mathematisch als

```
a(1,1) a(1,2) a(1,3) ... a(1,m)

a(2,1) a(2,2) a(2,3) ... a(2,m)

a(3,1) a(3,2) ...

...

a(n,1) a(n,2) ... a(n,m)
```

bezeichnet werden, so legt Fortran die Komponenten spaltenweise als

```
a(1,1) a(2,1) a(3,1) ... a(n,1) a(1,2) a(2,2) .... a(1,m) ... a(n,m)
```

im Speicher ab. Durch den Namen eines Datenfelds wird ein Zeiger (Pointer) auf den Speicherplatz definiert, an dem die sequentielle Speicherung der Komponenten beginnt. Bei der statischen Deklaration einer Matrix wird aufgrund des Datentyps der pro Komponente benötigten Bytes berechnet. Durch das `dimension`-Attribut wird die Dimension und die Anzahl der Kompontenten des Datenfeldes festgelegt.
Bei der **statischen** Deklaration eines Datenfeldes müssen im Vereinbarungsteil feste Zahlenwerte eingetragen werden, damit der Compiler die Anzahl der insgesamt benötigten Speicherplätze festlegen kann, z.B.

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

Hiermit werden 2 x 3 x 4 Byte = 24 Byte Speicherplatz für die Matrix `a` reserviert.
Wichtig: ist zu Beginn die genaue Anzahl an Zeilen und Spalten noch nicht bekannt, muss

man beim **statischen** Deklarationsverfahren über die Festlegung eines Maximalwertes für n und m (z.B. nmax=10 und mmax=10) einen geschätzten maximalen Speicherbedarf reservieren.

```
integer, parameter :: nmax = 10, mmax = 10
real,dimension(nmax,mmax) :: a = 0.0
```

Für die Matrix a sind nun im Speicher 10x10x4 Byte = 400 Byte fest reserviert worden. Werden erst später im Programmcode die Werte von n und m eingelesen und zugewiesen, so bleiben im statischen Fall ungenutzte Lücken im reservierten Speicherplatz frei.
Angenommen, es stellt sich im Laufe des Programms heraus, dass die Anzahl der Zeilen mit 3 (n=3) und die der Spalten mit 2 (m=2) festgelegt wird. Dann wird zum einen aufgrund der Schätzung eigentlich unnötig viel Speicherplatz für die aktuellen Komponenten der Matrix a reserviert. Dies ist der grundlegende Nachteil der **statischen Speicherallokierung**, dass wegen der universelleren Einsatzbarkeit der Programme im einzelnen Anwendungsfall mehr Speicher reserviert wurde, als im konkreten Fall gerade notwendig ist. Ab Fortran 90 bietet die `dynamische Speicherallokierung` die Möglichkeit, universelle Programme für den Umgang mit Datenfeldern zu entwickeln, die genau soviel Speicher reservieren und verwenden, wie dies im konkreten Einsatz gerade notwendig ist.
Den Fall der statischen Speicherallokierung in Zusammenhang mit der Übergabe von Datenfeldern an Unterprogramme ist jedoch von prinzipiellem Interesse (evtl. noch in Bibliotheksfunktionen verwendet bzw. in FORTRAN 77) und soll deshalb weiter genauer betrachtet werden.
Zurück zu dem konkreten Zahlenbeispiel:
Da die Komponenten einer Matrix spaltenorientiert abgespeichert werden, sieht die interne Speicherbelegung für die Matrix a wie folgt aus:

```
a(1,1) a(2,1) a(3,1) [7x4 Byte frei] a(2,1) a(2,2) a(2,3) [7x4 Byte frei]
[80x4 Byte]
```

Das Unterprogramm braucht, um mit der Matrix richtig umgehen zu können, also mehr Informationen als nur den Datentyp der Komponenten und den Zeiger auf den Beginn des Datenfeldes:
Merke:
Wenn Unterprogramme **statisch deklarierte, mehrdimensionale Datenfelder** als Ganzes verarbeiten sollen, muss an Information an das Unterprogramm mindestens übermittelt werden:

- Name des Datenfeldes (dadurch wird ein Zeiger auf den Beginn des Speicherbereiches übergeben)
- den Datentyp der Komponenten
- die bei der statischen Deklaration im Deklarationsteil vereinbarte Anzahl der Komponenten in mindestens allen Dimensionen bis einschließlich der vorletzten Dimension
- die tatsächlich im Programmcode verwendete Anzahl der Komponenten in jeder Dimension bis einschließlich in der vorletzten Dimension

- Wurde nicht die Anzahl der vereinbarten Komponenten in der letzten Dimension angegeben und stattdessen * verwendet, so ist es notwendig, stattdessen die exakte Anzahl der vereinbarten Komponenten in der vorletzten Dimension anzugeben.

- Um eine Überschreitung des Indexbereichs in der letzten Komponente zu vermeiden, ist es besser, auch für die letzte Dimension die in der statischen Deklaration festgelegte Anzahl an Komponenten an das Unterprogramm zu übergeben und analog zu dem Beispiel `vektor_uebergabe_korr` explizit auf eine mögliche Indexbereichsüberschreitung zu testen.

Die obigen Forderungen resultieren daher, dass innerhalb des Unterprogramms die Zugriffe auf ein Datenfeld sich ausschließlich innerhalb der gültigen Indexgrenzen bewegen darf. Dazu muss dem Unterprogramm mitgeteilt werden, wie das übermittelte Datenfeld strukturiert ist.
Im obigen Beispiel muss an das Unterprogramm dementsprechend mindestens die maximale Zeilenanzahl `nmax` und die tatsächliche Zeilenanzahl `n` sowie in der letzten Dimension mindestens die tatsächliche Anzahl an Spalten `m` angegeben werden, um die durch die Matrixkomponenten belegten Speicherbereiche exakt angeben zu können und jede belegte Matrixkomponente im Hauptspeicher aufsuchen zu können.
Die relative Position zu Beginn des Datenfeldes z.B. von `a(i,j)` berechnet sich aus ((j-1)*nmax + (i-1)) * 4 Byte, konkret findet sich im obigen Beispiel der Beginn des Speicherbereichs für `a(3,2)` (2-1)*10 + (3-1) = 12 Speicherplätze oder 12*4 = 48 Byte vom Beginn des Datenfeldes entfernt.
Beispielprogramm:

```
program array_uebergabe
implicit none
integer, parameter :: nmax=9, mmax=9      ! Achtung: beide Zahlen muessen
                                          ! <= 9 sein, sonst funktioniert
                                          ! Ausgabeformatstringberechung nicht
real, dimension(nmax,mmax) :: matrix      ! Zweidimensionales Datenfeld
                                          ! mit nmax Zeilen und mmax Spalten
integer :: n, m                           ! die vom Anwender gewuenschte
                                          ! Anzahl an Zeilen und Spalten
integer :: i, j                           ! Schleifenvariablen
logical :: mache                          ! Hilfsvariable
character(len=20) :: formatstring         ! Formatstring zur Ausgabe
                                          ! der Matrix(zeilen)

 5 format(1X,A$)      ! Format fuer Eingabezeile
10 format(1X,A,I2,/) ! Formatbeschreiber zur Ausgabe der Zeilen- bzw.
                      ! Spaltenzahl

! Einlesen der Zeilenanzahl n

mache = .TRUE.
do while (mache)
  write(*,5) 'Geben Sie bitte die Anzahl der Zeilen ein (Zeilenanzahl <= 9): '
  read(*,*) n
  write(*,10) '=> eingegeben wurde: ', n
  if ( n > 0 .AND. n <= nmax ) mache = .FALSE.
end do
```

```
28
29 ! Einlesen der Spaltenanzahl m
30
31 mache = .TRUE.
32 do while (mache)
33   write(*,5) 'Geben␣Sie␣bitte␣die␣Anzahl␣der␣Spalten␣ein&
34 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣&␣(Spaltenanzahl<=␣9):␣'
35   read(*,*) m
36   write(*,10) '=>␣eingegeben␣wurde:␣', m
37   if ( m > 0 .AND. m <= mmax  ) mache = .FALSE.
38 end do
39
40 ! Formatstring fuer die Ausgabe der Matrix
41 ! (wird aus Zeichenketten zusammengesetzt)
42 formatstring = '(1X,'//char(ichar('0')+m)//'(F3.1,1X))'
43 write(*,*) 'Formatstring␣fuer␣die␣Ausgabe␣der␣Matrixzeilen:␣', &
44             formatstring
45
46 do j = 1, m
47   do i = 1, n
48      matrix(i,j) = real(i) + 0.1*real(j)
49   end do
50 end do
51
52 ! Ausgabe der Matrix
53 write(*,*)
54 write(*,*) 'Ausgabe␣der␣Matrix:␣'
55 write(*,*)
56 do i = 1, n
57    write(*,formatstring)  (matrix(i,j), j=1,m)
58 end do
59 write(*,*)
60
61 ! Aufruf des Unterprogramms zur Modifikation der Matrix
62 call matrix_modifikation(matrix,nmax,mmax,n,m)
63
64 ! Ausgabe der Matrix
65 write(*,*)
66 write(*,*) 'Ausgabe␣der␣modifizierten␣Matrix:␣'
67 write(*,*)
68 do i = 1, n
69    write(*,formatstring)  (matrix(i,j), j=1,m)
70 end do
71 write(*,*)
72
73 end program array_uebergabe
74
75
76 subroutine matrix_modifikation (mat,n_max,m_max,n_,m_)
77
78 ! die Komponenten in ungerader Zeilen- bzw. Spaltenanzahl
79 ! werden durch 0.0 ersetzt
80
81 implicit none
82 integer, intent(in) :: n_max, m_max, n_, m_
```

```
real, dimension(n_max,m_max), intent(inout) :: mat
! Deklaration der lokalen Variablen
integer :: i, j

do j = 1, m_
  do i = 1, n_
    if ( mod(i,2) == 1 .OR. mod(j,2) == 1) then
       mat(i,j) = 0.0
    end if
  end do
end do

return
end subroutine matrix_modifikation
```

Beispielprogramm:

```
program array_uebergabe
implicit none
integer, parameter :: nmax=9, mmax=9
real, dimension(nmax,mmax) :: matrix      ! Zweidimensionales Datenfeld
                                          ! mit nmax Zeilen und mmax Spalten
integer :: n, m                           ! die vom Anwender gewuenschte
                                          ! Anzahl an Zeilen und Spalten
integer :: i, j                           ! Schleifenvariablen
logical :: mache                          ! Hilfsvariable
character(len=20) :: formatstring         ! Formatstring zur Ausgabe
                                          ! der Matrix(zeilen)

 5 format(1X,A$)      ! Format fuer Eingabezeile
10 format(1X,A,I2,/) ! Formatbeschreiber zur Ausgabe der Zeilen- bzw.
                      ! Spaltenzahl

! Einlesen der Zeilenanzahl n

mache = .TRUE.
do while (mache)
  write(*,5) 'Geben␣Sie␣bitte␣die␣Anzahl␣der␣Zeilen␣ein␣(Zeilenanzahl<=␣9):␣'
  read(*,*) n
  write(*,10) '=>␣eingegeben␣wurde:␣', n
  if ( n > 0 .AND. n <= nmax ) mache = .FALSE.
end do

! Einlesen der Spaltenanzahl m

mache = .TRUE.
do while (mache)
  write(*,5) 'Geben␣Sie␣bitte␣die␣Anzahl␣der␣Spalten␣ein&
␣␣␣␣␣␣␣␣␣␣␣␣␣␣&␣(Spaltenanzahl<=␣9):␣'
  read(*,*) m
  write(*,10) '=>␣eingegeben␣wurde:␣', m
  if ( m > 0 .AND. m <= mmax  ) mache = .FALSE.
end do

! Formatstring fuer die Ausgabe der Matrix
! (wird aus Zeichenketten zusammengesetzt)
```

```
formatstring = '(1X,'//char(ichar('0')+m)//'(F3.1,1X))'
write(*,*) 'Formatstring␣fuer␣die␣Ausgabe␣der␣Matrixzeilen:␣', &
            formatstring

do j = 1, m
  do i = 1, n
     matrix(i,j) = real(i) + 0.1*real(j)
  end do
end do

! Ausgabe der Matrix
write(*,*)
write(*,*) 'Ausgabe␣der␣Matrix:␣'
write(*,*)
do i = 1, n
   write(*,formatstring)  (matrix(i,j), j=1,m)
end do
write(*,*)

! Aufruf des Unterprogramms zur Modifikation der Matrix
call matrix_modifikation(matrix,nmax,n,m)

! Ausgabe der Matrix
write(*,*)
write(*,*) 'Ausgabe␣der␣modifizierten␣Matrix:␣'
write(*,*)
do i = 1, n
write(*,formatstring)  (matrix(i,j), j=1,m)
end do
write(*,*)

end program array_uebergabe


subroutine matrix_modifikation (mat,n_max,n_,m_)

! die Komponenten in ungerader Zeilen- bzw. Spaltenanzahl
! werden durch 0.0 ersetzt

implicit none
integer, intent(in) :: n_max, n_, m_
real, dimension(n_max,*), intent(inout) :: mat
! Bemerkung: analog dem alten Fortran-Syntax:
! Datenfelddeklaration in der Subroutine in der letzten
! Dimensionsangabe mit * wuerde auch funktionieren, nicht
! mehr verwenden, da moegliche Fehlerquelle
! Deklaration der lokalen Variablen
integer :: i, j

do j = 1, m_
  do i = 1, n_
    if ( mod(i,2) == 1 .OR. mod(j,2) == 1) then
      mat(i,j) = 0.0
    end if
  end do
```

```
95 end do
96
97 return
98 end subroutine matrix_modifikation
```

Ergänzende Bemerkungen Bei der dynamischen Allokierung von Datenfeldern fallen natürlich die bei der statischen Allokierung benötigten Maximalwerte für die im Speicher zu reservierende jeweilige maximale Anzahl an Komponenten weg. An das Unterprogramm brauchen deshalb nur die jeweilige tatsächliche Anzahl an Komponenten übergeben zu werden.
Das Folgende gilt sowohl für dynamisch als auch für statisch allokierte Datenfelder:
Für Arrays, die nicht wie üblich mit dem Index 1 beginnend deklariert werden, müssen bei Bedarf evtl. zusätzlich an das Unterprogramm die Informationen über die unteren Indexgrenzen übermittelt werden. Ob dies tatsächlich notwendig sein wird, hängt im Einzelfall von der Art der mathematischen Algorithmen ab, die Sie im Unterprogramm implementieren möchten. Manchmal reicht vielleicht im Programm evt. der relative Bezug des bzgl. des ersten Indices aus, manchmal vielleicht nicht.

10.8 Das `save`-Attribut in Unterprogrammen

Normalerweise sind die lokalen Variablen eines Unterprogramms nur solange gültig, bis das Unterprogramm wieder verlassen wurde. Nach Verlassen eines Unterprogramms sind die Werte der dort eingesetzten Variablen undefiniert.
Will man nun sicher erreichen, dass in einem Unterprogramm ermittelte Informationen beim nächsten Aufruf des Unterprogramms wieder zur Verfügung stehen, kann man die entsprechenden Variablen mit dem `save`-Attribut versehen, so dass diese Informationen gesichert werden (Fortran 90/95-Standard).
Das `save`-Attribut wird z.B. gebraucht, wenn man beim 2. Aufruf eines Unterprogramms Werte lokaler Variablen wieder verwenden möchte, die beim ersten Aufruf berechnet wurden. Dementsprechend falls man beim (n+1)-ten Aufruf eines Unterprogramms die beim n-ten Aufruf im Unterprogramm ermittelten Werte weiterverarbeiten möchte.
Hierzu müssen die lokalen Variablen, deren Wert beim nächsten Unterprogrammaufruf wieder zur Verfügung stehen soll, bei der Deklaration mit dem Attribut `save` versehen werden.

```
<Datentyp>, save :: <Variablenname>
```

Beispielprogramm:

```
1  program save_demo
2  implicit none
3  real :: x
4
5  do
6     write(*,'(A$)') 'Geben␣Sie␣einen␣Wert␣fuer␣x␣ein␣(0␣fuer␣Programmende):␣'
7     read(*,*)  x
8     if ( x == 0.0) exit
9     call ausgabe(x)
10 end do
```

```
11
12 end program save_demo
13
14 subroutine ausgabe(x)
15 implicit none
16 real           :: x
17 integer, save :: i
18
19 write(*,*)
20 write(*,*) 'Anzahl der bisherigen Aufrufe dieser subroutine:', i
21 write(*,*) 'eingegeben wurde: x = ', x
22 write(*,*)
23 i = i +1
24 end subroutine ausgabe
```

10.9 Rekursive Unterprogramm-Aufrufe

Seit Fortran 90 ist es möglich, dass ein Unterprogramm sich wieder selbst aufruft. Man spricht dann von einem rekursiven Unterprogrammaufruf. Die rekursiven Unterprogrammaufrufe kann z.B. bestimmte Sortieralgorithmen realisieren oder die Fakultät einer Zahl berechnen.
So ist z.B. 5! = 5·4·3·2·1 und 0! = 1. Damit lässt sich die Berechnung von z.B. 5! zerlegen in

```
5! = 5 · 4!
4! = 4 · 3!
3! = 3 · 2!
2! = 1 · 1!
1! = 1 · 0!
0! = 1
```

Der Berechnungs-Algorithmus für die Fakultät lässt sich sukzessive beschreiben als

```
Zerlege n! sukzessive in n! = n · (n-1)!
solange bis als letzte Zerlegung 0!=1 erscheint
dann werte den resultierenden Gesamtausdruck aus
```

Beispiel einer `recursive subroutine` zur Berechnung der Fakultät einer Zahl:

```
recursive subroutine factorial (n, ergebnis)
implicit none
integer, intent(in) :: n
integer, intent(out) :: ergebnis

integer :: z ! lokale Variable

if ( n >= 1) then

    call factorial(n-1,z)
    ergebnis = n * z
```

```
    else

        ergebnis = 1

    end if

    end subroutine factorial
```

Beispiel einer `recursive function` zur Berechnung der Fakultät einer Zahl:

```
    recursive function fakultaet(n) result(produkt)
    implicit none

    integer,intent(in) :: n
    integer :: produkt

    if ( n >= 1) then

        produkt = n * fakultaet(n-1)

    else

        produkt = 1

    end if
    end function fakultaet
```

Beispielprogramm:

```
1  program recursive_demo
2  implicit none
3  integer :: fakultaet  ! Datentyp zum Rueckgabewert der
4                        ! recursive function facultaet
5  integer :: wert
6
7  write(*,*) '----------------------------------------'
8  write(*,*) ' Beispiel zum rekursiven Programmaufruf '
9  write(*,*) '----------------------------------------'
10 write(*,*)
11 write(*,*) 'recursive fuction fakultaet(n) result(product):'
12 write(*,*)
13 write(*,*) 'fakultaet(0) = ', fakultaet(0)
14 write(*,*) 'fakultaet(1) = ', fakultaet(1)
15 write(*,*) 'fakultaet(2) = ', fakultaet(2)
16 write(*,*) 'fakultaet(3) = ', fakultaet(3)
17 write(*,*) 'fakultaet(4) = ', fakultaet(4)
18 write(*,*) 'fakultaet(5) = ', fakultaet(5)
19 write(*,*) 'fakultaet(6) = ', fakultaet(6)
20 write(*,*) 'fakultaet(7) = ', fakultaet(7)
21 write(*,*)
22
23 write(*,*)
24 write(*,*) 'recursive subroutine factorial(n,ergebnis)'
25 write(*,*)
```

```
26
27  call factorial(4,wert)
28
29  write(*,*) 'call␣factorial(4,wert),␣wert␣=␣', wert
30  write(*,*)
31
32  end program recursive_demo
33
34
35  recursive function fakultaet(n) result(produkt)
36  implicit none
37
38  integer,intent(in) :: n
39  integer            :: produkt
40
41  if ( n >= 1) then
42    produkt = n * fakultaet(n-1)
43  else
44    produkt = 1
45  end if
46
47  end function fakultaet
48
49
50  recursive subroutine factorial (n, ergebnis)
51  implicit none
52  integer, intent(in)  :: n
53  integer, intent(out) :: ergebnis
54
55  integer :: z  ! lokale Variable
56
57  if ( n >= 1) then
58    call factorial(n-1,z)
59    ergebnis = n * z
60  else
61    ergebnis = 1
62  end if
63
64  end subroutine factorial
```

Ein wichtiges Einsatzgebiet von sich selbst aufrufenden Unterprogrammen sind z.B. bestimmte Sortier-Algorithmen.

Kapitel 11

Module als Weiterentwicklung der veralteten COMMON-Blöcke von FORTRAN 77

Fortran 90/95 bietet noch eine weitere Möglichkeit zum bisher kennengelernten „**pass by reference scheme**“ um Informationen zwischen einzelnen Programmeinheiten (Hauptprogramm und Unterprogrammen) zu teilen und auszutauschen.
Diese zweite Möglichkeit, Werte zwischen einzelnen Programmteilen auszutauschen, besteht in der Verwendung von **Modulen**. In einem `module`-Programmblock können diejenigen Variablen deklariert werden, deren Werte zwischen den einzelnen Programmeinheiten ausgetauscht werden sollen. Innerhalb der Programmeinheiten, in denen das jeweilige Modul verwendet wird, kann bei Bedarf der Wert der im Modul deklarierten Variablen verändert werden. Sollen in einem Modul Konstanten vereinbart werden, deren Werte in den Programmeinheiten zwar verwendet, aber nicht verändert werden sollen, so werden diese wie üblich mit dem Attribut `parameter` versehen.
Der Aufbau (Syntax) eines einfachen Moduls zum Austausch von Werten bzw. Einbinden von Konstanten:

```
module <Name des Moduls>
implicit none
save ! stellt sicher, dass der Inhalt in den Speicherplaetzen
! zwischen den einzelnen Einbindevorgaengen in den
! einzelnen Programmeinheiten unveraendert bleibt
! Deklaration der Konstanten
<Datentyp>, parameter :: <Name der Konstante> = <Wert>
...
! Deklaration der Variablen
<Datentyp> :: <Name der Variablen>
...
end module <Name des Moduls>
```

Natürlich können in Modulen auch Datenfelder vereinbart werden.
Module werden vor Beginn des Hauptprogramms deklariert, während Unterprogramme hinter dem Hauptprogramm stehen sollten.

Soll in einer Programmeinheit der Inhalt eines vorher deklarierten Modules verwendet werden, so wird das Modul mit

```
use <Name des Moduls>
```

unmittelbar hinter der `program`- bzw. der `subroutine`- oder der `function`-Anweisung eingebunden (d.h. noch bevor das übliche

```
implicit none
```

folgt).
Wird man z.B. der Wert der Kreiszahl `pi` (eine Konstante) in mehr als einer Programmeinheit gebraucht, so kann man `pi` innerhalb eines Moduls deklarieren und dieses Modul in die entsprechenden Programmeinheiten einbinden.
Beispielprogramm:

```
! Beispielprogramm zur Deklaration eines Datenfeldes a
! mit den Komponenten vom Datentyp real a(1), a(2), a(3), a(4)

module kreiszahl
implicit none
save
real, parameter :: pi = 3.141593
end module kreiszahl

program kreis
implicit none
real :: radius, umfang
! Deklaration der Function
real :: area

write(*,*) 'Berechnung von Umfang und Flaeche eines Kreises'
write(*,*) 'Geben Sie den Radius ein:'
read(*,*) radius
call kreisumfang (radius,umfang)
write(*,*) 'Umfang : ', umfang
write(*,*) 'Flaeche: ', area(radius)

end program kreis


subroutine kreisumfang(r,u)
use kreiszahl
implicit none
real, intent(in)  :: r
real, intent(out) :: u

u = 2.0 * pi * r
return

end subroutine kreisumfang


real function area (r)
use kreiszahl
```

```
implicit none
real, intent(in) :: r

area = pi * r * r
return

end function area
```

Im folgenden Beispiel `parabel_module` wird ein Modul verwendet, um die Koeffizienten einer quadratischen Funktion zwischen dem Hauptprogramm und der `real function f(xarg)` auszutauschen.

```
module koeffizienten
implicit none
save                    ! stellt sicher, dass die (an den Speicherplaetzen
                        ! der in dem modul deklarierten Variablen)
                        ! abgelegten Werte zwischen den einzelnen
                        ! Programmeinheiten, in denen das Modul
                        ! eingebunden wird, erhalten bleiben
real :: a, b, c
end module koeffizienten


program parabel
use koeffizienten       ! damit sind die in dem Module koeffizienten
                        ! enthaltenen Definitionen bekannt
implicit none
integer :: i
real    :: x
real    :: f

10 format(6X,A$)
20 format(6X,A3,G12.5,A5,G12.5)
write(*,*) 'Bitte geben Sie zu f(x) = a*x**2 + b*x + c die Koeffizienten ein'
write(*,10) 'a = '; read(*,*) a
write(*,10) 'b = '; read(*,*) b
write(*,10) 'c = '; read(*,*) c
write(*,*) 'Zu welchem Wert x soll der Funktionswert f(x) berechnet werden?'
write(*,10) 'x = '; read(*,*) x
write(*,20) 'f(',x,') = ', f(x)

end program parabel


real function f(xarg)
use koeffizienten       ! damit sind die in dem Module koeffizienten
                        ! enthaltenen Definitionen einschliesslich
                        ! der im Hauptprogramm zugewiesenen Werte bekannt
implicit none
real, intent(in) :: xarg
f = a * xarg * xarg + b * xarg + c
return
end function f
```

Im obigen Beispiel wird das Modul `koeffizienten` genutzt, um Speicherplatz für die Koeffizienten `a`, `b` und `c` anzulegen. Im Hauptprogramm werden für Variablen `a`, `b` und `c` vom

Anwender Zahlenwerte einzulesen in den zugehörigen Speicherplätzen abgelegt.

```
real function f(xarg)
```

stehen aufgrund der Anweisung

```
use koeffizienten
```

die Datentypvereinbarungen von `a`, `b` und `c` sowie die in den Speicherplätzen abgelegten Koeffizientenwerte zur Verfügung.

11.1 Eigenschaften von Modulen

- Durch eine Modul-Definition wird der Speicherbereich für die in dem Modul deklarierten Konstanten und Variablen angelegt.
- Durch die `use`-Anweisung wird dieser Speicherbereich für andere Programmeinheiten nutzbar gemacht.
- Innerhalb der Programmeinheiten, in denen das Module über die `use`-Anweisung eingebunden wurde, kann aus diesen Speicherplätzen Information gelesen und geschrieben werden, es sei denn, dass in dem Modul Konstanten (Variablen mit dem Attribut `parameter`) vereinbart wurden, deren Speicherplätze sind schreibgeschützt.
- Werden aus einem Modul nur ein oder wenige deklarierte Konstanten oder Variablen benötigt, so bindet man diese in die Programmeinheit mit dem Attribut `only`, gefolgt von einem einfachen Doppelpunkt ein:

  `use` < **Name des Moduls** > ,`only` : < **Name(n) der Variablen/Konstanten** >

11.2 Module bei der Konstanten-Deklaration

Hat man in einem Programm sehr viele Konstanten zu deklarieren, deren Werte in den einzelnen Programmeinheiten gebraucht werden, ist das `module`-Konzept von Fortran 90/95 unschlagbar, denn es hat folgende Vorteile zu bieten:

- Der `module`-Deklarationsblock mit den für alle Programmeinheiten wichtigen Konstanten ist klar identifizierbar.
  Evtl. notwendige Änderungen und Anpassungen können an einer zentralen Stelle leicht vorgenommen werden.
- Konstanten lassen sich auch in Modulen durch das Attribut parameter vor unbeabsichtigten Modifikationen schützen.
- Das Einbinden der in Modulen enthaltenen Informationen ist eindeutig und klar:

  `use` < **Name des Moduls** >

- Werden aus einem Modul nur ein oder wenige deklarierte Konstanten benötigt, so bindet man diese in die Programmeinheit mit dem Attribut `only`, gefolgt von einem einfachen Doppelpunkt ein:

  `use` < **Name des Moduls** > ,`only` : < **Name(n) der Konstanten** >

Aus diesem Grunde sollten in größeren, aus einzelnen Programmeinheiten bestehenden Programmen zum Austausch von Konstantenwerten stets Module eingesetzt werden. Dies ist besonders sinnvoll, wenn viele Konstantenwerte gleichzeitig in verschiedenen Programmeinheiten verwendet werden sollen, zum Beispiel, wenn man ein Programmpaket erstellen möchte, um einfache Probleme aus der Elektrodynamik zu lösen, kann man alle häufig verwendeten physikalischen Konstanten in einem Modul zu vereinbaren und dieses Modul in den einzelnen Programmeinheiten einbinden.
Beispielprogramm:

```
1  module physikalische_Konstanten
2  implicit none
3  save
4  real, parameter ::  pi   = 3.141593     ! Kreiszahl pi
5  real, parameter ::  e    = 1.6022e-19   ! Elementarladung [C]
6  real, parameter ::  me0  = 9.1095e-31   ! Ruhemasse des Elektrons [kg]
7  real, parameter ::  c    = 2.99792e8    ! Vakuumlichtgeschwindigkeit [m/s]
8  real, parameter ::  eps0 = 8,8542e-12   ! el. Feldkonstante [C/Vm]
9  real, parameter ::  mu0  = 4.*pi*1.e-7  ! Magnetische Feldkonstante [Vs/Am]
10 end module physikalische_Konstanten
11
12
13 programm einfache_Elektrodynamik
14 use physikalische_Konstanten
15 implicit none
16   ...
17   ...
18 end programm einfache_Elektrodynamik
19
20
21 real function Feldstaerke_Zylinderkondensator(...,...)
22 use physikalische_Konstanten
23 implicit none
24   ...
25 end function Feldstaerke_Zylinderkondensator
26
27
28 real function Feldstaerke_Plattenkondensator(...,...)
29 use physikalische_Konstanten
30 implicit none
31   ...
32 end function Feldstaerke_Plattenkondensator
33
34
35 real function Kapazitaet_Plattenkondensator(...,...)
36 use physikalische_Konstanten
37 implicit none
38   ...
39 end function Kapazitaet_Plattenkondensator
```

```
40
41
42 real function Kapazitaet_Zylinderkondensator(...,...)
43 use physikalische_Konstanten
44 implicit none
45   ...
46 end function Kapazitaet_Zylinderkondensator
47
48
49 real function induzierter_Strom_in_linearem_Leiter(...,...)
50 use physikalische_Konstanten
51 implicit none
52   ...
53 end function induzierter_Strom_in_linearem_Leiter
54
55 ! plus weitere Unterprogramme
```

11.3 Einsatzmöglichkeiten von Modulen

Module kann man auch dazu verwenden, um nicht nur Konstanten, sondern auch die Werte von Variablen zwischen den Programmeinheiten auszutauschen. Dabei können die Unterprogramme auch dazu eingesetzt werden, um die an den entsprechenden Speicherplätzen abgelegten Werte willentlich zu verändern. Die veränderten Variablenwerte stehen dann der nächsten Programmeinheit, in der das Modul verwendet wird, zur Verfügung.
Um die Informationen aus den Speicherplätzen der in dem Modul deklarierten Variablen (und Konstanten) entnehmen zu können, reicht es, das Modul einzubinden. Sobald ein Modul eingebunden wurde, können die Werte aller Variablen (nicht der Konstanten) innerhalb der Programmeinheit verändert werden. Wird der einer in dem Modul zugewiesenen Variablen ein neuer Wert zugewiesen, so wird dieser sogleich an dem entsprechenden Speicherplatz der Variablen eingetragen.
Eine gewisse Gefahr besteht nun darin, dass in eingebundenen Modulen enthaltene Variablen versehentlich modifiziert werden. Im Vergleich zum Unterprogrammaufrufen mit dem zugehörigen *„pass by reference scheme“*, bei dem über die korrespondierende Liste an aktuellen und formalen Parametern exakt die Schnittstelle definiert ist, über die zwischen aufrufender Programmeinheit und dem Unterprogramm Informationen ausgetauscht werden, ist die Informationsübertragung über Module weniger restriktiv.

11.4 Module, die Unterprogramme enthalten (engl. *module procedures*)

Neben der Deklaration von Konstanten und Variablen lassen sich ganze Unterprogramme in Module einfügen. Sobald in einer anderen Programmeinheit das Modul durch eine `use`-Anweisung eingebunden wird, steht (stehen) in dieser Programmeinheit die im Modul enthaltene Unterprogrammroutine(n) zur Verfügung.
Die Syntax eines Moduls, welches ein Unterprogramm enthält:

```
module <Name des Moduls>
implicit none
save ! stellt sicher, dass der Inhalt in den Speicherplaetzen
! zwischen den einzelnen Einbindevorgaengen in den
! einzelnen Programmeinheiten unveraendert bleibt
! Deklaration der Konstanten und Variablen
...
contains
! normaler Unterprogrammcode
...
end module <Name des Moduls>
```

Als Beispiel wird in einem einfachen Programm zur Berechnung von Fläche und Umfang eines Kreises die Funktion `area_kreis` als *module procedure* eingesetzt. Beispielprogramm:

```
1  module kreisflaeche
2  implicit none
3  save
4  real, parameter :: pi = 3.141593
5  contains
6
7  real function area_kreis (r)
8  implicit none
9  real, intent(in) :: r
10
11 area_kreis = pi * r * r
12 return
13 end function area_kreis
14
15 end module kreisflaeche
16
17
18 program kreis
19 use kreisflaeche
20 implicit none
21 real :: radius
22
23 write(*,'(1X,A$)') 'Geben␣Sie␣den␣Radius␣des␣Kreises␣ein:␣'
24 read(*,*) radius
25
26 write(*,*) 'Die␣Flaeche␣des␣Kreises␣betraegt:␣', area_kreis(radius)
27 write(*,*) 'Der␣Umfang␣des␣Kreises␣betraegt:␣␣', 2.0 * pi * radius
28
29 end program kreis
```

Vorteile des Konzepts von Unterprogrammen in Modulen (engl. *module procedures*):

- Bei der Compilierung einer *„module procedure“* wird stets geprüft, ob beim Aufruf dieses Unterprogramms die Datentypen der aktuellen Parameter mit den Datentypen in der Deklaration des Unterprogramms angegebenen formalen Parametern tatsächlich übereinstimmen.
- Bei `functions` braucht der Datentyp der über Module eingebundenen *„module procedure functions“* nicht mehr deklariert zu werden.

Um zeigen zu können, wie gut der Datentyp-Check beim Aufruf von *module procedures* funktioniert, wird das Beispielprogramm nochmals um eine `real function volume_kugel(r)` ergänzt, die hinter dem Hauptprogrammende angefügt wurde.
Beispielprogramm:

```
module kreisflaeche
implicit none
save
real, parameter :: pi = 3.141593
contains

real function area_kreis (r)
implicit none
real, intent(in) :: r

area_kreis = pi * r * r
return
end function area_kreis

end module kreisflaeche


program kreis
use kreisflaeche
implicit none
real :: radius
real :: volume_kugel

write(*,'(1X,A$)') 'Geben␣Sie␣den␣Radius␣des␣Kreises␣ein:␣'
read(*,*) radius

write(*,*) 'Die␣Flaeche␣des␣Kreises␣betraegt:␣', area_kreis(radius)
write(*,*) 'Der␣Umfang␣des␣Kreises␣betraegt:␣␣', 2.0 * pi * radius
write(*,*) 'Das␣Volumen␣einer␣Kugel␣betraegt:␣', volume_kugel(radius)

end program kreis


real function volume_kugel(r)
use kreisflaeche, only : pi
implicit none
real, intent(in) :: r
volume_kugel = 4.0/3.0 * pi * r**3
return

end function volume_kugel
```

Um das unterschiedliche Verhalten des Compilers bei einem Datentyp-Fehler in Zusammenhang mit

1. der *module procedure* `area_kreis(radius)` und
2. der Function am Ende des Hauptprogramms `volume_kugel(radius)`

zu provozieren, wird im Hauptprogramm absichtlich der Datentyp von `radius`

```
real :: radius
```

auf den Datentyp `integer` verstellt

```
integer :: radius
```

und das Verhalten des Compilers beobachtet. Bei der Compilierung erhält man z.B. mit dem Salford FTN95-Compiler

1. bei der *module procedure*-Routine einen Error

```
kreis2.F90(25) :
error 327 - In the INTERFACE to AREA_KREIS (from MODULE KREISFLAECHE),
the first dummy argument (R) was of type REAL(KIND=1),
whereas the actual argument is of type INTEGER(KIND=3)
```

2. und der Unterprogramm-Routine nur eine Warning

```
kreis2.F90(32) :
warning 676 - In a call to VOLUME_KUGEL from another procedure,
the first argument was of type INTEGER(KIND=3), it is now REAL(KIND=1)
```

Im ungünstigsten Fall werden Warnings übersehen oder durch spezielle Compiler-Flags unterdrückt, so dass ein fehlerhaftes Executable erzeugt werden kann.
Fazit: Für eine sorgfältige Programmentwicklung in Fortran 90/95 empfiehlt es sich also, Unterprogramme in Module zu kapseln, um beim Compilieren eine explizite Datentyp-Prüfung mit evtl. Fehlermeldungen zu erhalten und von der FORTRAN 77 - Version mit separaten Unterprogrammen (ohne das Modul-Konstrukt von Fortran 90/95) Abstand zu nehmen.

Kapitel 12

Das `interface`-Konstrukt

Das `interface`-Konstrukt bietet eine weitere Möglichkeit ein explizites Interface zu schaffen. Mit einem separaten `interface`-Block kann man - wie bei den **module procedure**s - erreichen, dass der Compiler bei der Übersetzung des Programmcodes prüft, ob die Datentypen von aktuellen und formalen Parametern zwischen aufrufender Programmeinheit und Unterprogramm tatsächlich übereinstimmen. Sollte dies an einer Stelle nicht der Fall sein, bricht der Compiler mit einer Fehlermeldung den Übersetzungsvorgang an der kritischen Stelle ab.
Beispielprogramm:

```
1  module kreiszahl
2  implicit none
3  save
4  real, parameter :: pi = 3.141593
5  end module kreiszahl
6
7  program kreis
8  use kreiszahl
9  implicit none
10
11 interface
12         subroutine kreisumfang(r,u)
13         use kreiszahl
14         implicit none
15         real, intent(in)  :: r
16         real, intent(out) :: u
17         end subroutine kreisumfang
18
19         real function area (r)
20         use kreiszahl
21         implicit none
22         real, intent(in) :: r
23         end function area
24 end interface
25
26 real :: radius, umfang
27 ! Deklaration der Function nicht mehr notwendig,
28 ! wenn interface-Block den Deklarationsteil der function
29 ! enthaelt
30 !real :: area
```

```
31
32 write(*,*) 'Berechnung von Umfang und Flaeche eines Kreises'
33 write(*,*) 'Geben Sie den Radius ein:'
34 read(*,*) radius
35 write(*,*) 'pi = ', pi
36 call kreisumfang (radius,umfang)
37 write(*,*) 'Umfang : ', umfang
38 write(*,*) 'Flaeche: ', area(radius)
39
40 end program kreis
41
42
43 subroutine kreisumfang(r,u)
44 use kreiszahl
45 implicit none
46 real, intent(in)  :: r
47 real, intent(out) :: u
48 u = 2.0 * pi * r
49 return
50 end subroutine kreisumfang
51
52
53 real function area (r)
54 use kreiszahl
55 implicit none
56 real, intent(in) :: r
57 area = pi * r * r
58 return
59 end function area
```

Der interface-Block im Hauptprogramm enthält jeweils pro „angebundenen“ Unterprogramm

- die Kopfzeile des Unterprogramms,
- den Deklarationsteil für die formalen Parameter sowie
- die Endezeile des Unterprogramms

Die Verwendung des interface-Konstrukts führt ebenso wie das Verfahren der *module procedures* dazu, dass während des Compilierens eine explizite Prüfung stattfindet, ob die Datentypen der aktuellen Parameter (beim Unterprogrammaufruf) tatsächlich mit den formalen Parameters (die im Unterprogramm deklariert wurden) übereinstimmen.
Der interface-Block wird ebenfalls eingesetzt, wenn vom einem Fortran 90/95 - Programm aus auf externe FORTRAN 77 -, C- oder vorcompilierte Bibliotheks-Routinen zugegriffen werden soll.
Das interface-Konstrukt bietet des weiteren eine elegante Möglichkeit, dynamisch allokierte Datenfelder an Unterprogramme zu übergeben, ohne dass dem Unterprogramm über die Liste der aktuellen Parameter oder ein eingebundenen Modul die Anzahl der Komponenten in den einzelnen Dimensionen mitgeteilt werden müsste.

12.1 Aufbau des Deklarationsteils im Hauptprogramm mit interface-Block

Mit dem interface-Konstrukt hat man folgende schematischen Struktur des Deklarationsteils des Hauptprogrammteils:

```
program < Programmname>
! Module einbinden mit
use < Name des Moduls >
! bzw.
use < Name des Moduls >, only : < Namen der benötigten Konstanten, Variablen
oder module procedure >
implicit none

interface
! Deklarationsteil des Unterprogramms bis
! einschliesslich der formalen Parameter
! Endezeile des Unterprogramms

! evtl. weitere Unterprogramme nach gleichem Schema
end interface

! Deklaration der Konstanten
! evtl. nach Datentyp sortiert
! und immer gut dokumentiert

! Deklaration der Variablen
! evtl. nach Datentyp sortiert
! und immer gut dokumentiert

! evtl. Deklaration der Datentypen der Functions
! (falls diese nicht schon als module procedure
! oder im interface-Block angebunden)
```

Kapitel 13

Erweiterte Datentypen

13.1 Der intrinsische Datentyp `complex`

Genaugenommen handelt es sich hier um einen in Fortran generisch enthaltenen Datentyp für komplexe Zahlen. Diese haben in der Mathematik meist die Form

$$c = a + \imath b$$

wobei sich die komplexe Zahl c aus der Menge der komplexen Zahlen aus dem Realteil a und dem Imaginärteil b zusammensetzt. Zur Darstellung komplexer Zahlen wird eine Ebene benötigt. Ein komplexer Zahlenwert ist dann ein Punkt in der komplexen Ebene. An der Achse nach rechts (der Abzisse) lässt sich der Realteil und auf der Ordinate der Imaginärteil ablesen. Der Betrag einer komplexen Zahl ist der Abstand des Punktes vom Ursprung des Koordinatensystems und berechnet sich zu

```
|c| = sqrt(a*a+b*b).
```

Der Phasenwinkel phi ist gegeben durch den Arcustangens von b/a.

```
phi = atan(b/a)
```

Komplexe Zahlen lassen sich addieren, subtrahieren, multiplizieren und dividieren. Allerdings sind komplexe Zahlen nicht geordnet, d.h. ein Vergleich, ob die komplexe Zahl c1 kleiner oder größer als die komplexe Zahl c2 ist, ist sinnlos. Was sich allerdings wieder vergleichen ließe, ist, ob |c1| kleiner oder größer als |c2| ist, weil der Betrag einer komplexen Zahl wieder eine reelle Zahl ist und weil im Vergleich zweier reeller Zahlen unterschieden werden kann, welche der Zahlen die größere von beiden ist.
Die Dartstellung komplexer Zahlen in Fortran zeigt Tabelle 13.1.

Deklaration komplexer Variablen

Komplexe Variablen werden als Datentyp `complex` deklariert, z.B. eine komplexe Zahl `c`

```
complex :: c
```

oder z.B. ein Datenfeld `feld` mit 10 komplexen Komponenten mit

```
complex, dimension(10) :: feld
```

komplexe Zahl (math.)	Fortran-Darstellung
$3+4\imath$	`(3.0,4.0)`bzw.`(3.,4.)`
$-\imath$	`(0.0,-1.0)`
1	`(1.0,0.0)`
$-0.0042+75\imath$	z.B. `(-4.2E-3,0.75E2`

Tabelle 13.1: Komplexe Zahlen in Fortran

Wertzuweisungen bei der Deklaration der Variablen lassen sich ebenfalls durchführen, wenn z.B. die Konstante `i` immer als komplexe Zahl $\imath$ geführt werden soll, kann man dies gut mit

```
complex, parameter :: i = (0.0,1.0)
```

durchführen. Danach lassen sich die komplexe Variablen mit der vordeklarierten komplexen Zahl $\imath$ gut initialisieren. Und `i` lässt sich als komplexe Zahl `i` im Programmcode gut einsetzen.
Beispielprogramm:

```
program complex_demo
implicit none
complex, parameter    :: i  = (0.0,1.0)
complex               :: z1 = 3.0 + 4*i      ! interne Datentypkonversion
complex               :: z2 = (-2.0,1.0)
complex, dimension(5) :: feld
integer               :: n

write(*,*) 'Demoprogramm mit komplexen Zahlen'
write(*,*)
write(*,*) 'voreingestellt sind: '
write(*,*)
write(*,*) '    i = ', i
write(*,*) '   z1 = ', z1
write(*,*) '   z2 = ', z2
write(*,*)
write(*,*) 'Arithmetik mit komplexen Zahlen:'
write(*,*)
write(*,*) ' 5          * i           = ', 5*i
write(*,*) ' 5          * z1          = ', 5.0*z1
write(*,*) ' 5.0        * z1          = ', 5.0*z1
write(*,*) ' (5.0,0.0)  * z1          = ', (5.0,0.0)*z1
write(*,*)
write(*,*) '         z1 + z2          = ', z1 + z2
write(*,*) '         z1 - z2          = ', z1 - z2
write(*,*) '         z1 * z2          = ', z1 * z2
write(*,*) '         z1 / z2          = ', z1 / z2
write(*,*)
write(*,*) 'Der Betrag einer komplexen Zahl:'
write(*,*) '         cabs(z1)         = ', cabs(z1)
write(*,*) '          abs(z2)         = ',  abs(z2)
write(*,*) 'Der Realteil einer komplexen Zahl :'
write(*,*) '         real(z1)         =', real(z1)
write(*,*) '         real(z2)         =', real(z2)
write(*,*) '         real(i)          =', real(i)
write(*,*) 'Der Imaginaerteil einer komplexen Zahl:'
```

```
write(*,*) '␣␣␣␣␣␣␣␣aimag(z1)␣␣␣␣␣␣␣␣=', aimag(z1)
write(*,*) '␣␣␣␣␣␣␣␣aimag(z2)␣␣␣␣␣␣␣␣=', aimag(z2)
write(*,*) '␣␣␣␣␣␣␣␣aimag(i)␣␣␣␣␣␣␣␣␣=', aimag(i)
write(*,*)
write(*,*) 'Zwei␣real-Zahlen␣zu␣einer␣komplexen␣Zahl␣&
&zusammensetzen␣mit␣cmplx'
write(*,*) '␣␣␣␣␣␣␣␣cmplx(2.5,7.1)␣␣␣=', cmplx(2.5,7.1)
write(*,*)
write(*,*) 'Verarbeitung␣von␣komplexwertigen␣Datenfeldern,␣z.B␣mit:'
write(*,*) '␣␣␣␣␣␣do␣n=1,␣5'
write(*,*) '␣␣␣␣␣␣␣␣␣␣feld(n)␣=␣cmplx(real(n),-real(n*n))'
write(*,*) '␣␣␣␣␣␣␣␣␣␣write(*,*)␣feld(n)'
write(*,*) '␣␣␣␣␣␣end␣do'
write(*,*)
do n=1, 5
  feld(n) = cmplx(real(n),-real(n*n))
  write(*,*) feld(n)
end do
write(*,*)
write(*,*) 'Anwenden␣mathematischer␣Funktionen␣auf␣komplexe␣Zahlen,␣z.B.␣'
write(*,*) '␣␣␣␣␣␣␣␣␣␣log(i)␣␣␣␣␣␣␣␣␣␣␣=', log(i)
write(*,*) '␣␣␣␣␣␣␣␣␣␣log(z1)␣␣␣␣␣␣␣␣␣␣=', log(z1)
write(*,*) '␣␣␣␣␣␣␣␣␣␣log(z2)␣␣␣␣␣␣␣␣␣␣=', log(z2)
write(*,*)
write(*,*) 'formatierte␣Ausgabe␣von␣komplexen␣Zahlen:'
write(*,*) 'im␣Formatbeschreiber␣wird␣eine␣komplexe␣Zahl'
write(*,*) 'wie␣2␣aufeinanderfolgende␣real-Zahlen␣behandelt:'
write(*,*)
write(*,'(2F5.1)') 1.5+2.0*i

end program complex_demo
```

Die komplexen Zahlen werden im Rechner von der Voreinstellung her sowohl im Realteil als auch im Imaginärteil mit der Genauigkeit des Datentyps `real` verarbeitet.

Einlesen komplexer Werte

Beispielprogramm:

```
program read_complex
implicit none
complex :: z

do
  write(*,*) 'Geben␣Sie␣eine␣komplexe␣Zahl␣ein!'
  read(*,*) z
  write(*,*) 'eingelesen␣wurde:␣', z
  write(*,*)
end do

end program read_complex
```

Achtung: der Salford FTN95 erwartet beim listengesteuerten Einlesen komplexer Zahlen den Real- und Imaginärteil in runde Klammern eingeschlossen und durch ein Komma getrennt, sonst erhält man einen Programmabbruch mit einem Runtime-Error.

Um das Einleseprogramm allgemeingültig und portabel zu gestalten, ist somit das Programm um eine Fehlerabfang-Routine zu erweitern. Beispielprogramm:

```
program read_complex_erweitert
implicit none
complex :: z
integer :: io_err

do
  write(*,*) 'Geben Sie eine komplexe Zahl ein!'
  read(*,*,iostat=io_err) z
  if (io_err /= 0) then
    write(*,*) 'Bitte geben Sie die Zahl im Format'
    write(*,*) '(a,b)'
    write(*,*) 'ein, wobei a der Zahlenwert fuer den Realteil und'
    write(*,*) 'b der Zahlenwert fuer den Imaginaerteil ist,'
    write(*,*) 'einschliesslich der Klammern und des Kommas'
    write(*,*)
    cycle
  end if
  write(*,*) 'eingelesen wurde: ', z
  write(*,*)
end do

end program read_complex_erweitert
```

13.2 Datentypen mit höherer Genauigkeit

Sowohl der Datentyp `real` als auch der der Realteil und der Imaginärteil des Datentyps `complex` werden auf den meisten Compileren intern mit 4 Byte - Darstellungsgenauigkeit codiert. Damit weisen diese beiden Zahlentypen maximal 6-7 Dezimalstellen Genauigkeit auf.
Da viele wissenschaftliche, technische und numerische Anwendungen nach höherer Darstellungsgenauigkeit verlangen, bietet Fortran 90 eine Möglichkeit die Anzahl an Genauigkeitsstellen zu erhöhen.
Alle Fortran 90/95 - Compiler bieten die Möglichkeit, ohne größeren Aufwand die Anzahl der Genauigkeitsstellen bei reellen und/oder komplexen Zahlen auf ca. 14-15 Genauigkeitsstellen zu erhöhen. In diesem Fall werden in der Regel vom Compiler für die interne Codierung reeller Zahlen mindestens die doppelte Anzahl an Bytes (4 statt 8 Byte) verwendet.
Eine übersichtliche, wenn auch nicht die modernste Art, die Anzahl der Genauigkeitsstellen für reelle Zahlen zu erhöhen, besteht darin, die bisherige FORTRAN 77 - Konvention zu nutzen.

13.2.1 `double precision`= „doppelte“ Genauigkeit (ca. 14 - 15 Stellen) für reelle Zahlen

1. statt `real` als Datentyp `double precision` oder (veraltet) `real*8` angeben, z.B.

```
double precision :: r, umfang
```

2. Ausnahmslos müssen gleichzeitig alle(!) Zahlenwerte auf ca. 14-15 Stellen genau angegeben werden und mit dem Zusatz `d0` versehen werden, bzw. es muss der Exponent statt mit `e` mit `d` eingeleitet werden, z.B.

```
double precision, parameter :: pi = 3.1415926535879d0
umfang = 2.0d0 * pi * r
```

Dies bedeutet insbesondere: Wenn Sie die Genauigkeit in der Zahlendarstellung erhöhen wollen, müssen Sie wirklich konsequent **alle (!)** Zahlenwerte in dem gesamten Programm anpassen, sonst kann es zur Reduktion von den erwünschten ca. 14-15 Stellen auf 6-7 Stellen Genauigkeit kommen!

Notwendige Umstellungen für `double precision`

- Bei allen Zahlen (in wissenschaftlicher Darstellung) muss die Exponentenmarkierung `e` durch `d` oder `D` bzw. `E` durch `d` oder `D` ersetzt werden. Z.B. statt

```
1.23e4
```

muss für `double precision` geschrieben werden

```
1.23d4
```

- Reelle Zahlenwerte in doppelter Genauigkeit ohne Exponentendarstellung müssen zusätzlich mit `d0` oder `D0` ergänzt werden. Z.B. statt

```
4.2
```

muss für `double precision` geschrieben werden

```
4.2d0
```

- Mathematische, physikalischen oder programmspezifische Konstanten müssen - soweit irgend möglich - mit der erweiterten Genauigkeit angegeben werden, z.B. statt

```
real, parameter :: pi = 3.141593
```

bei doppelter Genauigkeit

```
double precision, parameter :: pi = 3.1415926535879d0
! da genau 15 Stellen zugewiesen werden, ist hier d0 anzuhaengen nicht
zwingend noetig
```

- Zum Programm gehörende Unterprogramme müssen in der Genauigkeit ebenfalls angepasst werden, z.B. aus

```
real function (x)
implicit none
real, intent(in) :: x
```

wird bei doppelter Genauigkeit

```
double precision function (x)
implicit none
double precision, intent(in) :: x
```

- Zusätzlich ist darauf zu achten, dass bei der Umwandlung ganzer Zahlen in reelle Zahlen statt

  ```
  real( )
  ```

  nun

  ```
  dble( )
  ```

  stehen muss.

- Die Formatbeschreiber für die Ausgabe sind gegebenfalls anzupassen (Anzahl der Nachkommastellen!)

- Unter Umständen muss (z.B. für die Anzahl an Schleifendurchläufen, wenn diese nun größer als 2^(31) werden können) der Darstellungsbereich des Datentyps `integer` z.B. durch `selected_int_kind` (siehe unten) erweitert werden.

Statt `d` bzw. `e` als Kleinbuchstaben zu schreiben, kann man natürlich genausogut die Großbuchstaben verwenden, da Fortran zwischen Groß- und Kleinschreibung nicht unterscheidet.
Da beim Datentyp `double precision` für die interne Zahlendarstellung mehr als die 4 Byte beim Datentyp `real` zur Verfügung stehen, wird auch der Wertebereich darstellbarer reeller Zahlen erweitert - und zwar von ca. +/- 38 als maximalem Wert des Exponenten auf doppelt genaue Zahlen ca. 14-15 Genauigkeitsstellen und maximal bis ca. +/-308 im Exponenten (`d+308` bzw. `d-308`). Das folgende Anwendungsbeispiel ist die Berechnung der Quadratwurzel aus 2 mit doppelter Genauigkeit.
Beispielprogramm:

```
1  program sqrt_2
2  implicit none
3  double precision :: a, b, c, d
4
5  a = sqrt(2.0d0)
6  b = sqrt(2.0)
7  c = sqrt(dble(2))
8  d = sqrt(real(2))
9  write(*,*)  'Berechnung von Wurzel aus 2 mit doppelter Genauigkeit'
10 write(*,1)  'Ergebnis aus Maple: ','1.4142135623730950'
11 write(*,5)  'richtig: sqrt(2.0d0)    = ', a
12 write(*,10) 'falsch : sqrt(2.0)      = ', b, '<= !!! falsch !!!'
13 write(*,5)  'richtig: sqrt(dble(2))  = ', c
14 write(*,10) 'falsch : sqrt(real(2))  = ', d, '<= !!! falsch !!!'
15 1  format(1X,T2,A,T30,1X,A)
16 5  format(1X,T2,A,T30,F17.14)
17 10 format(1X,T2,A,T30,F17.14,T49,A)
18
19 end program sqrt_2
```

Wählt man eine unformatierte Ausgabe, so gibt der Salford FTN95 weniger Stellen (genaugenommen 12) auf dem Bildschirm aus, obwohl der Salford FTN95 Compiler intern beim Datentyp `double precision` auf ca. 14-15 Stellen genau arbeitet.
Ein weiteres Beispiel zeigt, dass bei der Umstellung eines Programms auf doppelte Genauigkeit wirklich konsequent darauf geachtet werden muss, alle reellen Zahlenwerte innerhalb des Programmcodes im Exponenten von `e` bzw. `E` auf `d` bzw. `D` umzustellen bzw. an Zahlenwerte ohne Exponent `d0` bzw. `D0` anzuhängen.
Beispielprogramm:

```
1  program kugelvolumen
2  implicit none
3  double precision, parameter :: pi = 3.141592653589793d0
4  double precision :: r, v1, v2, v3
5
6  write(*,*) 'Berechnung des Volumens einer Kugel'
7  write(*,'(1X,A$)') 'Bitte geben Sie den Radius ein: '
8  read(*,*) r
9  write(*,*) 'Eingelesen wurde: ', r
10 v1 = 4.0d0/3.0d0 * pi * r**3
11 v2 = 4.0d0/3.0 * pi * r**3
12 v3 = 4.0/3.0 * pi * r**3
13 write(*,*) 'Das Volumen der Kugel ist:'
14 write(*,*) '4.0d0/3.0d0 *pi*r**3 =', v1, ' richtig'
15 write(*,*) '4.0d0/3.0    *pi*r**3 =', v2, ' impl. Datentyp-Konversion'
16 write(*,*) '4.0/3.0      *pi*r**3 =', v3, ' <= falsches Ergebnis !!!'
17
18 end program kugelvolumen
```

Das 2. Ergebnis wird nur wegen der impliziten Datentyp-Konversionsregel von Fortran richtig berechnet. In `4.0d0/3.0` wird ein Zahlenwert des Datentyps `double precision` durch einem arithmetischen Operator (/) mit einem Zahlenwert des Datentyps `real` verknüpft. Dabei wird die 2. Zahl in den Datentyp der höheren Genauigkeit umgewandelt (aus der Zahl des Datentyps `real` wird eine Zahl des Datentyps `double precision`), bevor die arithmetische Operation (die Division) durchgeführt und das Ergebnis (des Datentyps `double precision`) berechnet wird.
Das Beispiel zeigt auch, dass der Anwender seine Zahlenwerte als „ganz normale Zahlen“ eingeben kann und nicht mit der Endung `d0` zu versehen braucht, wenn das Einlesen der Zahlenwerte listengesteuert und damit ohne Formatangabe erfolgt. Im obigen Beispiel ist es z.B. ganz ein Ordnung, wenn ein Anwender `1.95` statt `1.95d0` als Zahleneingabe schreibt, weil das Programm beim listengesteuerten Einlesen mit der impliziten Datentyp-Konversion nach `double precision` sorgt.

13.2.2 `double complex`= „doppelte“ Genauigkeit für komplexe Zahlen

Manchmal ist es notwendig, auch komplexe Zahlen mit doppelter Genauigkeit zu definieren. Auch hier kann man die Abwärtskompatibilät von Fortran 90/95 zu FORTRAN 77 nutzen.

- Deklaration komplexer Zahlen mit doppelter Genauigkeit

```
double complex ::  z1
```

oder alternativ mit

```
complex*16 :: z2
```

- Die Umwandlung reeller Zahlen doppelter Genauigkeit in komplexe Zahlen mit doppelter Genauigkeit erfolgt z.B. über

```
double precision :: r1 = 2.0d0, i1 = 1.0d0
! Umwandlung in eine komplexe Zahl mit doppelter Genauigkeit
z1 = dcmplx(r1,i1)
```

Auch hier ist es wiederum wichtig, dass systematisch darauf geachtet wird, dass konsequent im gesamten Programm alle betroffenen Zahlenwerte und Rechenoperationen auf den Datentyp mit der doppelten Genauigkeit umgestellt werden müssen. Ansonsten kann es zu Rundungsfehlern aufgrund von unbeabsichtigten Verlusten in der Darstellungsgenauigkeit und damit zu numerischen Fehlern kommen.
Das folgende Beispiel zeigt, wie wichtig es ist, bei der Umstellung auf `double complex` die Funktion `cmplx( , )` durch `dcmplx( , )` zu ersetzen und gleichzeitig darauf zu achten, dass an allen relevanten Stellen Werte des Datentyps `real` in Werte des Datentyp `double precision` konvertiert werden.
Beispielprogramm:

```
1  program komplex_double
2  implicit none
3  double complex :: i
4  double complex :: z1,z11, z12, z2
5
6  i   = dcmplx(0.0d0,1.0d0)
7  z1  = dcmplx(0.4d0,-5.0d0/3.0d0)
8  z11 = cmplx(0.4d0,-5.0d0/3.0d0)
9  z12 = dcmplx(0.4,-5.0/3.0)
10 z2  = i/z1
11
12 write(*,*) 'z1␣␣=␣dcmplx(0.4d0,-5.0d0/3.0d0)=␣', z1
13 write(*,*) 'z11␣=␣␣cmplx(0.4d0,-5.0d0/3.0d0)=␣',z11 , '<=␣falsch!!!'
14 write(*,*) 'z12␣=␣dcmplx(0.4,␣␣-5.0/3.0␣␣␣␣)=␣',z12 , '<=␣falsch!!!'
15 write(*,*)
16 write(*,*) 'z2␣=␣i␣/␣z1␣=␣', z2
17 write(*,*)
18 write(*,*) '␣abs(i)␣␣=␣', abs(i)
19 write(*,*) '␣abs(z2)␣=␣', abs(z2)
20 write(*,*) '␣exp(i)␣␣=␣', exp(i)
21 write(*,*) '␣exp(z2)␣=␣', exp(z2)
22
23 end program komplex_double
```

Ob die in Fortran eingebauten intrinsischen Funktionen wie z.B. `abs( )` und `exp( )` auch für doppelt genaue komplexe Zahlen ohne Genauigkeitsverlust arbeiten, lässt sich z.B. mit Maple gegenprüfen. Durch den Vergleich mit den Ergebnissen von Maple erkennt man, dass die intrisischen Fortran 90/95 - Funktionen (z.B. `exp( )`) ohne Genauigkeitsverlust mit `double complex` arbeiten. Es können die generischen Funktionsnamen weiterverwendet werden und der Compiler sorgt dafür, dass intern diejenigen Routinen aufgerufen werden,

die dem übergebenen Datentyp entsprechen.
Achtung: in FORTRAN 77 konnte es (je nach Compiler) unter Umständen zu Genauigkeitsverlusten kommen, falls z.B. nicht `CDEXP( )` statt `CEXP( )` und alle anderen intrinsischen Funktionsnamen auf die Datentypen mit doppelter Genauigkeit umgeschrieben wurden.

13.2.3 Der compilerabhängige `kind`-Wert

Bei der Deklaration von Variablen bzw. Konstanten lässt sich in Fortran 90/95 mit Hilfe von `kind` die Genauigkeit in der Zahlendarstellung umstellen.
`kind` ist darüber hinaus eine in Fortran 90/95 enthaltene intrinsische Funktion, die einen (leider(!) compilerabhängigen) Zahlenwert für die einzelnen Werte zurückgeben kann.
Test-Programm für die `kind`-Werte:

```
1 program kind_demo
2 implicit none
3
4 write(*,*) 'kind(0.0)␣=', kind(0.0)
5
6 write(*,*) 'kind(0.0D0)␣=', kind(0.0D0)
7 end program kind_demo
```

Bildschirm-Ausgabe beim g95-Compiler:

```
kind(0.0) = 4
kind(0.0D0) = 8
```

Der Salford FTN95 Compiler bringt beim gleichen Programm jedoch:

```
kind(0.0) = 1
kind(0.0D0) = 2
```

Wie wir oben gesehen haben, ist `0.0` ein Wert vom Datentyp `real` mit einfacher Genauigkeit und `0.0D0` eine reelle Zahl mit doppelter Genauigkeit.
Eine der Möglichkeiten, sich in Fortran 90/95 eine Zahl des höheren (doppelten) Genauigkeitstyps zu deklarieren, wäre z.B. mit dem g95 die folgende:

```
real(kind=8) :: x, y  ! Achtung: compilerabhaengige Version
```

welche an die Fortran 77 - Version

```
REAL*8  X
```

erinnert. Zahlwerte mit doppelter Genauigkeit ließen sich nun in Fortran 90/95 (compilerabhängig und nicht mehr universell portierbar) z.B. als

```
x = 2.6_8
y = .7_8
```

zuweisen. Problematisch ist jedoch bei diesem Verfahren, dass - wie wir von oben wissen - der `kind`-Wert für den reellen Datentyp mit doppelter Genauigkeit compilerabhängig ist. Würde man wie oben vorgehen, so müsste jeder doppelt genaue Zahlenwert im Programmcode für z.B. den g95 oder auch den Compaq Visual Fortran - Compiler mit `_8` und z.B. für

den Salford-Compiler mit `_2` versehen werden. Zu diesem äussert unschönen Weg gibt es jedoch bessere Alternativen.
Gleich zu Beginn des Deklarationsteils werden zwei `integer`-Konstanten deklariert, die die Genauigkeitsangabe beinhalten. Wird eine Genauigkeitsangabe benötigt, so wird mit den Konstantennamen gearbeitet:

```
integer, parameter :: single = 4 ! muss compilerabhaengig angepasst werden
integer, parameter :: double = 8 ! muss compilerabhaengig angepasst werden
real(kind=single) :: a
real(kind=double) :: x, y
```

Den Variablen lassen sich dann innerhalb des Programms die Zahlwerte lassen z.B. als

```
a = 100.2_single
x = 2.6_double
y = .7_double
```

zuweisen.
Wird ein solches Programm von einem Compiler mit einem `kind`-Wert von 8 für die doppelte Genauigkeit (bzw. 4 für einfache Genauigkeit) arbeitenden Compiler auf einem Rechner mit einem Compiler, der als Zahlenwerte 2 (für doppelte) bzw. 1 (für einfache) Genauigkeit erwartet, transferiert, so muss in dem gesamten Programmcode nur noch zu Beginn die Zahlenwerte für `double` bzw. `single` angepasst werden.
Die divergierende Entwicklung unterschiedlicher `kind`-Werte für reelle Zahlen einfacher und doppelter Genauigkeit bei den Compilerherstellern, wurde durch das für den Fortran 90/95 - Standard entwickelte `selected_real_kind`-Verfahren wieder aufgehoben.

13.2.4 `selected_real_kind` = compiler-unabhängiges Verfahren zur Einstellung der Genauigkeit reeller Zahlen

Um den `kind`-Wert einer Zahl compilerunbhängig einzustellen zu können, wurde mit

```
selected_real_kind(p=<Anzahl Genauigkeitsstellen>)
```

oder

```
selected_real_kind(r=<max.Exponentenwert zur Basis 10>)
```

oder

```
selected_real_kind(p=<Anzahl Genauigkeitsstellen>,r=<max.Exponentenwert
zur Basis 10>)
```

eine Methode geschaffen, die es erlaubt, die compilerabhängige `kind`-Zahl zu finden und auszuwählen, welche mindestens die vorgegebene Anzahl an Genauigkeitsstellen darstellen kann. Die Obergrenze setzt hier der verwendete Compiler. Zur Zeit gehen - mit wenigen rühmlichen Ausnahmen (ein Beispiel siehe unten) - die meisten Compiler nicht über die Genauigkeit von `double precision` hinaus.
Mit `p=<Anzahl Genauigkeitsstellen>` lassen sich die benötigte Anzahl der Genauigkeitsstellen explizit angeben. Wenn man z.B. mindestens 8 Stellen Genauigkeit haben möchte, lässt sich dies formulieren als

```
integer, parameter :: s8 = selected_real_kind(p=8)
```

oder als

```
integer, parameter :: s8 = selected_real_kind(8)
```

Mit r=<max.Exponentenwert zur Basis 10> lässt sich der Zahlenbereich angeben, zu dem der Wertebereich dieser Zahlen definiert wird.
Mit

```
integer, parameter :: sp = selected_real_kind(p=13,r=200)
```

bzw.

```
integer, parameter :: sp = selected_real_kind(13,200)
```

lässt sich der `kind`-Wert reeller Zahlen festlegen, die mit 13 Stellen Genauigkeit im Darstellungsbereich bis zum Betrag von 10^{200} darstellbar sein sollen.
Beispielprogramm:

```
program hoehere_genauigkeit
implicit none
! eine Moeglichkeit, sich Variablen mit hoeherer Genauigkeit
! als die 6-7 Stellen Genauigkeit des Datentyps real zu definieren

! Es wird ein real-Datentyp vereinbart, der mindestens 12 Stellen
! Genauigkeit aufweist

integer, parameter :: s12 = selected_real_kind(p=12)

real            :: x1
real(kind=s12) :: x2, x3

x1 = 2.9/7.34
x2 = 2.9_s12 / 7.34_s12      ! Achtung: Alle Zahlenwerte muessen
                             ! explizit mit der neuen
                             ! Genauigkeitsangabe
                             ! versehen werden, sonst
                             ! wird weiterhin nur mit den
                             ! 6-7 Stellen des Datentyps real
                             ! gearbeitet
x3 = 2.9/7.34

write(*,*) 'x1 (Genauigkeit des Datentyps real)          = ', x1
write(*,*) 'x2 (12 Dezimalstellen Genauigkeit)           = ', x2
write(*,*) 'x3 (Genauigkeitsverlust durch falsche Angabe)= ', x3
write(*,*) '2.9/7.34 (Ergebnis von Maple)                = 0.395095367847411'
write(*,*)
write(*,*) 'compilerabhaengig: selected_real_kind(p=12) = ', s12
write(*,*)

end program hoehere_genauigkeit
```

Achtung: Wenn Sie die Genauigkeit in der Zahlendarstellung erhöhen wollen, müssen Sie konsequent **alle (!)** Zahlenwerte in dem gesamten Programm anpassen, sonst kann es passieren, dass die gewünschte Genauigkeit unbeabsichtigt auf 6-7 Stellen reduziert wird.

Die interne Zahlendarstellung erfolgt anhand der vom Anwender gewählten Zahlenwerte für `p` bzw. `r` in `selected_real_kind(p=, r= )` mit demjenigen `kind`-Wert der der einfachen oder der doppelten Genauigkeit entspricht. Insofern kommt man mit den meisten Compilern mit dieser Methode auch nicht über die Darstellungsgenauigkeit von ca. 14-15 Stellen des Datentyps `double precision` bzw. dem Darstellungsbereich von `double precision` von ca. -10^{308} bis 10^{308} für die größten bzw. von ca. -10^{-308} bis 10^{-308} für die kleinsten Zahlen hinaus.
Eine Ausnahme bilden hier Compiler, die den Datentyp `real` intern mit noch mehr Byte (z.B. 16 Byte statt 4 Byte) codieren können (z.B. der Sun f95 - Compiler auf dem Computeserver des Rechenzentrums `btrzxn`).

```
btrzxn> uname -a
SunOS btrzxn 5.8 Generic_117350-28 sun4u sparc SUNW,Sun-Blade-1000
```

Hier kann man z.B. ohne größeren Aufwand bis zu ca. 33 Stellen genau rechnen:

```
1  program real_16byte
2  implicit none
3  integer, parameter :: p33 = selected_real_kind(p=33)
4  real(kind=p33) :: a, b
5
6  a = log(5.0_p33)
7  b = log(5.0)
8
9  write(*,*)            'log(5) (Ergebnis von Maple)   =  &
10 &  1.60943791243410037460075933322618763 95'
11 write(*,'(1X,A,(ES41.33))') 'log(5.0_p33)                    = ', a
12 write(*,'(1X,A,(ES41.33),A)') 'log(5.0)                      = ', b ,' <= !!!'
13 write(*,*)
14 write(*,*) 'kind-Wert (compilerabhaengig) von 5.0_p33 : ', kind(5.0_p33)
15
16 end program real_16byte
```

Dieses Beispiel zeigt zweierlei: mit dem entsprechenden Compiler lässt mit Hilfe von `selected_real_k` `)` ein Fortran 90/95 - Programm leicht so umstellen, dass mit einer nochmals deutlich höheren Genauigkeit als `double precision` gerechnet werden kann. Und zum anderen zeigt sich auch hier, dass um Genauigkeitsverluste zu vermeiden, der Programmierer peinlichst genau darauf achten muss, wirklich **alle(!)** relevanten Stellen im Programmcode auf die Version mit höheren Genauigkeit umzustellen:

- alle reellen Zahlenwerte müssen mit der Genauigkeitsmarkierung versehen werden, z.B.

  ```
  integer, parameter :: p10=selected_real_kind(p=10)
  real(kind=p10) :: ... ! Variablenliste
  ```

  Z.B. statt

  `1.23e4` bzw. `4.2`

  muss für für den Datentyp der höheren Genauigkeit geschrieben werden

  `1.23e4_p10` bzw. `4.2_p10`

- mathematische, physikalischen oder programmspezifische Konstanten müssen - soweit irgend möglich - mit der erweiterten Genauigkeit angegeben werden z.B. statt

  ```
  real, parameter :: pi = 3.141593
  ```

  bei dem konkreten Beispiel

  ```
  real(kind=p10), parameter :: pi = 3.141592654_p10
  ```

- zum Programm gehörende Unterprogramme müssen in der Genauigkeit ebenfalls angepasst werden, z.B. aus

  ```
  real function (x)
  implicit none
  real, intent(in) :: x
  ```

  wird

  ```
  real(kind=p10) function (x)
  implicit none
  integer, parameter :: p10=selected_real_kind(p=10)
  real(kind=p10), intent(in) :: x
  ```

- Zusätzlich ist darauf zu achten, dass bei der Umwandlung ganzer Zahlen in reelle Zahlen statt

  ```
  real( )
  ```

  nun (wiederum nach der im Beispiel festgelegten Wert)

  ```
  real( ,kind=p10)
  ```

  stehen muss.

- Die Formatbeschreiber für die Ausgabe sind gegebenfalls anzupassen (Anzahl der Nachkommastellen!)

- Unter Umständen muss (z.B. für die Anzahl an Schleifendurchläufen, wenn diese nun größer als 2^{31} werden können) der Darstellungsbereich des Datentyps `integer` z.B. durch `selected_int_kind` (siehe unten) erweitert werden.

13.2.5 Die Multi-Precision-Library

Benötigt man reelle Zahlen mit den mehr als 14-15 Genauigkeitsstellen und steht einem kein Compiler zur Verfügung der in der Lage ist reelle Zahlen in 16 Byte zu codieren oder reicht dies immer noch nicht aus, so kann man z.B. noch auf die Multi-Precision-Library zurückgreifen.

13.2.6 `selected_int_kind` = compiler-unabhängiges Verfahren zur Einstellung der Genauigkeit ganzer Zahlen

Der Wertebereich aus dem Daten vom Datentyp `integer` bei den üblichen 4-Byte - Binärdarstellung umfasste bisher -2**31+1 bis 2**(31). Für den Datentyp `integer` bietet dementsprechend `selected_int_kind(r=<Exponentenwert>)` eine Möglichkeit, den Darstellungsbereich der ganzen Zahlen zu erweitern. Der Zahlenwert `Exponentenwert` in der Klammer gibt die Anzahl der Stellen an. Dies entspricht einem Wertebereich ganzer Zahlen im Bereich von `-10**(Exponentenwert)` bis `10**(Exponentenwert)`.

13.3 Benutzerdefinierte Datentypen (Die `type`- Vereinbarungsanweisung)

Die in Fortran enthalten Datentypen lassen sich zu benutzerdefinierten Datentypen verknüpfen mittels

```
type :: <Name des zusammengesetzten Datentyps >

    sequence ! optional: sorgt dafuer, dass die einzelnen
    !Bestandteile zusammenhaengend im Speicher abgelegt werden
    ! Deklaration der einzelnen Komponenten
    ...

end type <Name des zusammengesetzten Datentyps >
```

Eine Variable des selbstkonstruierten Datentyp lässt sich deklarieren durch

```
type(<Name des zusammengesetzten Datentyps >) :: <Variablennamen>
```

Zum Beispiel lässt sich leicht ein benutzerdefinierter Datentyp `student` erzeugen, der sich aus dem Vornamen, dem Nachnamen und der Matrikel-Nummer der Studentin oder des Studenten zusammensetzt. Mit dem durch eine `type`-Vereinbarungsanweisung deklarierten Datentyp kann man genauso, wie bisher geschehen, durch die Angabe von `dimenension( )` ein Datenfeld (Array) mit diesen Komponenten erzeugen (siehe Beispielprogramm).
Innerhalb eines Programmes kann auf eine im `type`-Verbund enthaltene Variable durch Angabe des Variablennamens gefolgt von einem Prozent-Zeichen und dem Namen der Unterkomponente zugegriffen werden.
Referenzierung auf eine Komponente in einem `type`-Verbund

```
<Name der type-Variable>%<Name einer Verbundkomponente>
```

Durch den Zugriff auf eine Unterkomponente des `type`-Verbunds kann deren Wert bei Bedarf verändert bzw. ausgeben werden. Soll eine
Wertzuweisungen an die Gesamtheit aller Komponenten eines zusammengesetzten Datentyps erfolgen (z.B. als Initialisierung), so kann dies z.B. über

```
<Name der type-Variable>=<Name des zusammengesetzten Datentyps>(Liste
der Werte)
```

geschehen.
Beispielprogramm:

```
module datenbankdefinition
implicit none
save
integer, parameter           :: n = 3  ! Anzahl Datensaetze
character(len=35), parameter :: str = '(2X,I2,3X,A25,1X,A25,1X,A6/)'
                                    ! fuer formatierte Ausgabe von student
type :: student
  character(len=25) :: vorname, nachname
  character(len=6)  :: matrikel_nr
end type student

type(student), dimension(n)   :: personen
end module datenbankdefinition


program type_einfach
use datenbankdefinition
implicit none
character(len=6)     :: nummer
integer              :: i

!----------------------
!  Formatbeschreiber
!----------------------

15 format(1X,A$)      ! Ausgabeformat: Zeichenkonstante mit Unterdrueckung
                      ! des Zeilenvorschubs
20 format(A6)         ! Einleseformat Matrikelnummer
25 format(A25)        ! Einleseformat Matrikelnummer

!-------------------------------------------------------------
! Wertzuweisung fuer den zusammengesetzten Datentyp
!-------------------------------------------------------------

personen(1) =  student('Hans','Maier','123456')
personen(2) =  student('Irene','Huber','111111')
personen(3) =  student('Maria', 'Rose','000001')

call datenbank_ausgabe

!-------------------------------------------------------------
! Den Nachnamen einer Person aendern
!-------------------------------------------------------------

write(*,*)
write(*,*) '====================================='
write(*,*)      'Funktion: Nachname aendern:'
write(*,*) '====================================='
write(*,15) 'Bitte geben Sie die Matikelnummer ein: '
read(*,20) nummer

do i = 1, n
   if (nummer == personen(i)%matrikel_nr) then
```

```
      write(*,*)
      write(*,str) i, personen(i)%vorname, personen(i)%nachname, &
      personen(i)%matrikel_nr
      write(*,15) '      >>>  neuer Nachname   =>   '
      read(*,25)   personen(i)%nachname
   end if
end do

call datenbank_ausgabe

end program type_einfach


subroutine datenbank_ausgabe
use datenbankdefinition
implicit none
integer :: i

write(*,*)
write(*,*) '====================================='
write(*,*) 'Funktion: Ausgabe der Datenbank:'
write(*,*) '====================================='
do i = 1, n
   write(*,str) i, personen(i)%vorname, personen(i)%nachname, &
   personen(i)%matrikel_nr
end do
return

end subroutine datenbank_ausgabe
```

Beispielprogramm:

```
! ------------------------------------------
! Das Programm berechnet aus
!         Radius und Laenge sowie
!         der Dichte
! die Masse von massiven Zylinderkoerpern
! ------------------------------------------

module koerperdefinition
implicit none
save

real, parameter :: pi = 3.141593
character(len=74), parameter :: fb_zylinder = &
"(/3X,'Radius =',1X,G12.5/,3X,'Laenge =',1X,G12.5/,3X,'Dichte =',1X,G12.5)"

type :: zylinder
  sequence
  real :: radius
  real :: laenge
  real :: dichte
end type zylinder
end module koerperdefinition


```

```
25 module koerpermethoden
26 use koerperdefinition
27 implicit none
28 save
29 contains
30
31 subroutine ausgabe(zyli)
32 ! formatierte Ausgabe der Zylinderdaten
33 implicit none
34 type(zylinder), intent(in) :: zyli
35
36 write(*,fb_zylinder) zyli%radius, zyli%laenge, zyli%dichte
37 end subroutine ausgabe
38
39 subroutine einlesen(zyli)
40 ! formatierte Ausgabe der Zylinderdaten
41 implicit none
42 type(zylinder), intent(out) :: zyli
43 real :: r, l, d
44
45 5 format(1X,A$)
46 write(*,*)
47 do
48   write(*,5) ' Radius = '; read(*,*) r
49   if ( r > 0.0 ) exit
50 end do
51 do
52   write(*,5) ' Laenge = '; read(*,*) l
53   if ( l > 0.0 ) exit
54 end do
55 do
56   write(*,5) ' Dichte = '; read(*,*) d
57   if ( d > 0.0 ) exit
58 end do
59 write(*,*)
60 zyli = zylinder(r,l,d)
61 end subroutine einlesen
62
63 real function volumen(zyli)
64 ! berechnet das Volumen eines Zylinders
65 implicit none
66 type(zylinder),intent(in) :: zyli
67
68 volumen = pi * (zyli%radius)**2 * (zyli%laenge)
69 return
70 end function volumen
71
72 real function masse(zyli)
73 ! berechnet die Masse des Zylinders
74 implicit none
75 type(zylinder),intent(in) :: zyli
76
77 masse = pi * (zyli%radius)**2 * (zyli%laenge) * (zyli%dichte)
78 return
79 end function masse
```

```
80
81  end module koerpermethoden
82
83
84  program gewichtsberechnung
85  use koerpermethoden
86  implicit none
87  type(zylinder) :: koerper1 = zylinder(2.0,5.0,19.3)
88  type(zylinder) :: koerper2
89
90  write(*,*) 'Der 1. Zylinder wird beschrieben durch:'
91  call ausgabe(koerper1)
92  write(*,*)
93  write(*,*) 'Berechnete Groessen:'
94  write(*,*) '     Volumen = ', volumen(koerper1)
95  write(*,*) '     Masse   = ', masse(koerper1)
96  write(*,*)
97  write(*,*)
98  write(*,*) 'Fuer den 2. Zylinder werden nun von Ihnen die Werte eingelesen'
99  call einlesen(koerper2)
100 write(*,*) ' Die von Ihnen eingegebenen Werte '
101 call ausgabe(koerper2)
102 write(*,*)
103 write(*,*) 'Berechnete Groessen:'
104 write(*,*) '     Volumen = ', volumen(koerper2)
105 write(*,*) '     Masse   = ', masse(koerper2)
106 write(*,*)
107
108 end program gewichtsberechnung
```

Das letzte Beispiel zeigt die Definition eines benutzerdefinierten Datentyps `zylinder` in einer `type`-Vereinbarung.
Das `module koerperdefinition` wird zum einen zur `type`-Vereinbarung des Datentyps genutzt, zum anderen werden gleichzeitig eng mit diesem Datentyp verbundene Konstantenwerte festgelegt. Neben dem Wert von pi wird ein Ausgabeformatbeschreiber definiert, der später zur Ausgabe des verknüpften Datentyps eingesetzt wird.
In einem zweiten Module `koerpermethoden` werden sämtliche Unterprogramme definiert, die im Umgang mit dem Datentyp `zylinder` gebraucht werden. Dies sind hier zum einen eine Routine zur Ausgabe, eine zum Einlesen und zwei Berechnungsroutinen.
Der Ansatz sich jeweils Module zur Datentyp-Definition und für die Methoden im Umgang mit dem Datentyp hat als Vorteile Wiederverwendbarkeit, klare Struktur und die Vereinfachung des Umgangs mit abgeleiteten Datentypen.
Gewissermaßen handelt es sich um einen **objektorientierten Ansatz**. Es werden Objekte und Methoden im Umgang mit diesen Objekten definiert. Die Objekte und Methoden sind „gekapselt“ und ein anderer Programmierer, der diese Module nicht selbst entwickelt hat, kann mit den Objekten „arbeiten“, wenn er nur genaue Kenntnisse über die Schnittstellen und das Verhalten der Routinen hat. Detailkenntisse über den internen Aufbau würden in diesem Fall nicht benötigt (Stichwort: modulare Softwareentwicklung, modularer Softwareaufbau).

Kapitel 14

Fortgeschrittene Unterprogramm-Konzepte

14.1 Unterprogramme als formale Parameter in anderen Unterprogrammen

Die Programmiersprache Fortran macht es möglich, Unterprogramme von anderen Unterprogrammen aus aufzurufen.
Im folgenden Beispiel wird z.B. eine Routine programmiert, die ein vorgegebenes Intervall in eine vorgegebene Anzahl äquidistanter Stützstellen zerlegt, die Funktionswerte einer Function des Datentyps `real` an der Stützstellen berechnet und über das gesamte Intervall mittelt.
Programmiert man sich ein Unterprogramm, welches die Intervallzerlegung und die Mittelwertbildung übernimmt und lässt man sich die Funktionswerte mit einem separaten Programm berechnen, so muss das Mittelwertbildungs-Unterprogramm die Information mitgeteilt bekommen, welches externe Programm zur Funktionswert-Berechnung verwendet werden soll. Oder anderes ausgedrückt: man müsste den Unterprogrammaufruf zur Mittelwertbildung so gestalten, dass man als aktuellen Parameter den Namen einer Funktion einsetzen kann, etwa in der Art

```
write(*,*) mittelwert(f,x_unten,x_oben,schritte)
```

wobei in der Liste der aktuellen Parameter f der Name eines externen Unterprogramms ist und die restlichen aktuellen Parameter die aktuellen Werteangaben zum Intervall enthalten. Will man den Mittelwert der Funktionswerte einer anderen Funktion berechnen, so kann man als aktuellen Parameter einfach den Namen der anderen Funktion (hier z.B. f1 statt f) verwenden, also

```
write(*,*) mittelwert(f1,x_unten,x_oben,schritte)
```

Beim Unterprogrammaufruf wird ein Zeiger (*pointer*) auf das als aktueller Parameter aufgeführte Unterprogramm übergeben.
Im Unterprogramm steht als Platzhalter für das beim Funktionsaufruf tatsächlich eingesetzte Unterprogramm ein formaler Parameter (im Beispielprogramm `func`):

```
real function mittelwert (func, anfangswert, endwert, n)
```

Wichtig: Damit Unterprogramme als formale und aktuelle Parameter beim Unterprogrammaufruf verwendet werden können, müssen bei der Datentypanmeldung für die Functions diese bei der Deklaration mit dem Attribut `external` versehen werden. Dies trifft sowohl auf `func` in der `function mittelwert`

```
real function mittelwert (func, anfangswert, endwert, n)
! Unterprogramm zur Mittelwertberechung
implicit none
real, external :: func
real, intent(in) :: anfangswert, endwert
integer, intent(in) :: n
```

als auch auf die Functions `f` und `f1` im Hauptprogramm zu, da diese als aktuelle `function`-Parameter beim Aufruf der Function `mittelwert` eingesetzt werden. Deshalb müssen diese Functions im Hauptprogramm mit dem Attribut `external` angemeldet werden

```
program external_demo
implicit none
real :: mittelwert
real, external :: f, f1
```

wohingegen dies für die `function` Mittelwert nicht der Fall ist.
Beispielprogramm:

```
1  program external_demo
2  implicit none
3  real            :: mittelwert
4  real, external  :: f, f1
5
6  write(*,*) 'Mittelwert␣␣f(x)␣␣von␣␣␣[0.0,10.0]␣(n=3)␣␣=␣', &
7             mittelwert(f,0.0, 10.0, 3)
8  write(*,*) 'Mittelwert␣␣f1(x)␣von␣␣␣[0.0,1.0]␣(n=10)␣␣=␣', &
9              mittelwert(f1,0.0, 1.0, 10)
10
11 end program external_demo
12
13
14 real function f(x)
15 ! Die nutzerdefinierte Funktion f(x)
16 implicit none
17 real, intent(in) :: x
18 f  = x
19 return
20 end function f
21
22
23 real function f1(x)
24 ! Die nutzerdefinierte Funktion f1(x)
25 implicit none
26 real, parameter :: pi = 3.141593
27 real, intent(in) :: x
28 f1 = 2.0 * pi * x**3
29 return
```

```
30 end function f1
31
32
33 real function mittelwert (func, anfangswert, endwert, n)
34 ! Unterprogramm zur Mittelwertberechung
35 implicit none
36 real, external :: func
37 real, intent(in)    :: anfangswert, endwert
38 integer, intent(in) :: n
39 real               :: delta, summe
40 integer            :: i
41
42 delta = (endwert - anfangswert) / real(n-1)
43 summe = 0.0
44
45 do i = 1, n
46   summe = summe + func(real(i-1)*delta)
47 end do
48
49 mittelwert = summe / real(n)
50 return
51 end function mittelwert
```

Im Programm werden sowohl `f` als auch `f1` aktuelle Function-Parameter beim Aufruf der Function `mittelwert` verwendet. Dabei wird z.B. bei der ersten Berechnung für die in der `function f` vorgegebene Funktion $f(x) = x$ im Intervall $[0.0, 10.0]$ an drei äquidistant voneinander entfernten Stützstellen der Mittelwert der Funktionswerte berechnet. Hier also $(f(0) + f(5) + f(10))/3 = 15/3 = 5$.
Der Einsatz von Unterprogrammen als formale und aktuelle Parameter hat den Vorteil, dass sich Programmpakete sehr viel universeller einsetzbar gestalten lassen und dass bei Bedarf leicht einzelne Routinen durch andere ausgetauscht werden können.
Sollen statt Fuctions Subroutines als formale und aktuelle Parameter eingesetzt werden, so werden statt bei der Datentyp-Deklaration der `function` das Attribut `external` zu verwenden, die involvierten Subroutines als

```
external < Name der Subroutine >
```

deklariert.

14.2 Unix: Der Aufbau von Programmpaketen und die `make`-Utility

Wie bereits erwähnt wurde, sind unter anderen die Vorteile von Unterprogrammen dass sich klar strukturierte übersichtliche Programme schaffen lassen und dass sich einzelne Unterprogrammen in ähnlichen Projekten meist gut wiederverwenden lassen. Will man vermeiden, die benötigen Unterprogramme am Ende des Hauptprogramms einkopieren zu müssen, kann man statt dessen das Konzept des **Makefiles** einsetzen. Hier befinden sich meist die einzelnen Programmteile (Hauptprogramm und zugehörige Unterprogramme) als eingeständige Dateien auf gleiche Hierarchie-Ebene nebeneinander. Ein sogenanntes **Makefile** regelt, dass die Programmeinheiten jeweils in einen sogenannten Object-Code compiliert

und danach zu einem Executable miteinander gelinkt werden.
Das Makefile gibt die Regelsätze vor, wie die einzelnen Programmeinheiten zu Objektcode compiliert und wie die Objektcodes zu einem ausfürbaren Programm verknüpft (gelinkt) werden sollen.
Es sollen sich nun in einem Verzeichnis nur die Programmteile des Beispielprogramms befinden:

```
> ls -l
total 3
-rw-r--r--   1 c02g00   c02g           575 Jan 28 16:54 main.f90
-rw-r--r--   1 c02g00   c02g           132 Jan 28 16:54 f.f90
-rw-r--r--   1 c02g00   c02g           495 Jan 28 16:54 mittelwert.f90
```

Das Hauptprogramm ist nun:

```
! Das Programm ermittelt mit Hilfe der function mittelwert den
! an n aequidistanten Stuetzstellen ermittelten mittleren Funktionswert
! einer Funktion f (vorgeben durch die Funktion f) im Intervall
! [anfangswert,endwert]

program external_demo
implicit none
real :: mittelwert
real, external :: f

write(*,*) 'Der Mittelwert aus den Stuetzstellenwerten betraegt:', &
            mittelwert(f,0.0, 10.0, 3)
            ! f: Unterprogramm als aktueller Parameter beim Aufruf
            ! der Function mittelwert

end program external_demo
```

Die mathematische Funktion als `real function f(x)`:

```
real function f(x)      ! Die nutzerdefinierte Funktion
implicit none
real, intent(in) :: x
f =x
return

end function f
```

Die Funktion zur Berechnung des Mittelwertes als
`real function mittelwert(func, anfangswert, endwert, n)`:

```
real function mittelwert ( func, anfangswert, endwert, n)
implicit none
real, external      :: func       ! Unterprogramm als formaler Parameter
real, intent(in)    :: anfangswert, endwert
integer, intent(in) :: n
real :: delta, x0
real :: summe = 0.0
integer :: i

delta = (endwert - anfangswert) / real(n-1.0)
```

```
11 do i = 1, n
12   x0 = real(i-1) * delta
13   summe = summe + func(x0)
14 end do
15 mittelwert = summe / real(n)
16 return
17
18 end function mittelwert
```

Mit

```
man make
```

kann man sich unter Unix über die Make-Utility informieren. Will man sich über die bei make voreingestellten Makro-Definitionen und Regelsätze informieren, hilft einem

```
make -p
```

weiter.
Eine ausgezeichnete Erklärung zur make-Utility bietet eine Seite des *ZAM des Forschungszentrums Jülich* (http://www.zam.kfa-juelich.de/zam/docs/bhb/bhb_html/d0140/chapter_23/section_1.html). Weitere detaillierte Informationen finden sich bei *D. Faulbaum (Elektronenspeichering Bessy)* (http://www-csr.bessy.de/seminar/Tutorial/Make/Make.html).
Will man ein Makefile erstellen, so erstellt man dieses in dem Verzeichnis, in dem sich nun die Programmdateien befinden.
Man kann dieses makefile oder Makefile nennen. Das make-Utility wird aufgerufen, in dem in dem Verzeichnis, welches die Programmdateien und das Makefile enthält, an der Kommandozeile einfach nur

```
make
```

eingegeben wird.
Natürlich könnte man dem Makefile auch einen anderen Namen als makefile oder Makefile geben. Allerdings müsste man dann

```
make -f < alternativer Name des Makefiles>
```

aufrufen, um die Make-Utility zu starten.
In einem Makefile werden Abhängigkeiten, Compilier und Link-Regeln definiert. Mit der Anweisung

```
f.o: f.f90
~f90 -c f.f90
```

wird im ersten Teil festgelegt, dass die Grundlage von f.o die Fortran 90 - Programmeinheit f.90 ist. In der zweiten Zeile, die unbedingt mit einem Tabulator-Zeichen beginnen muss, steht die Anweisung, wie f.o erzeugt werden soll. Und zwar wird angegeben, dass f.90 mit dem f90-Compiler in Object-Code compiliert werden soll. Dass es Objectcode werden soll, regelt die Compiler-Flag -c. Dieselben Abhängigkeiten und Verfahren gelten für die beiden anderen Programmeinheiten: main.f90, dem Hauptprogramm und dem Unterprogramm

aus der Datei `mittelwert.f90` gemacht. Die oberste Zeile beschreibt die Abhängigkeit von `hauptprogramm` von den Objektcode - Dateien und die zweite Zeile gibt an, dass diese zu dem Executable `hauptprogramm` gelinkt werden sollten.
Nachdem an der Kommandozeile mit

```
make
```

die Make-Utility gestartet wurde, erhält man von `make` die Rückmeldung, was gerade getan wird. Es erscheinen nacheinander beim obigen Beispiel die Zeilen:

```
f90 -c main.f90
f90 -c f.f90
f90 -c mittelwert.f90
f90 -o gesprog main.o f.o mittelwert.o
```

Dies wären die Befehle, die man auch nacheinander an der Kommandozeile eingeben müsste, wenn man ohne die Make-Utility arbeiten wollte. Im Verzeichnis stehen nun zusätzlich zum Fortran-Quelltext die Object-Files. Wir der Quelltext in einem der Programmeinheiten verändert und man ruft erneut die Make-Utility auf, so wird nur noch diese Programmeinheit neu übersetzt und die Objectcode-Files neu gelinkt.
Zum Beispiel wird nun im Hauptprogramm das Intervall, aus an 3 äquidistant voneinander entfernten Stellen (Anfangspunkt des Intervalls, Endpunkt des Intervalls und Intervallmitte) bereits bei x=6.0 beendet. Der Unterprogrammaufruf vom Hauptprogramm aus (in `main.f90`) lautet nun

```
mittelwert(f,0.0, 6.0, 3)
```

Als Mittelwert der Funktionswerte erwartet man $(f(0) + f(3) + f(6))/3 = 3$.
Doch zunächst wollen wir noch betrachten, wie sich nun die Make-Utility verhält. Nach Eingabe von

```
make
```

an der Kommandozeile erhält man

```
f90 -c main.f90
f90 -o gesprog main.o f.o mittelwert.o
```

Die Make-Utility geht also ökonomisch vor: Es werden beim wiederholten Aufruf der Make-Utility nur noch diejenigen Teile neu compiliert (und dann natürlich gelinkt), die in der Zeit seit dem letzten Aufruf der Make-Utility verändert worden sind.
Das ausführbare Programm wird nun an der Kommandozeile gestartet

```
./gesprog
```

Für das gewählte Funktions- und Zahlenbeispiel erhält man erwartungsgemäß

```
Der Mittelwert aus den Stuetzstellenwerten betraegt:   3.000000
```

Ruft man

```
make clean
```

auf, so werden alle Dateien mit der Endung `*.o` (alle als Objectcode vorliegenden Zwischencompilierstufen) gelöscht.

```
make cleanall
```

löscht zusätzlich das vorher erzeugte Executable `gesprog` mit weg.

14.3 Unix: Die automatische Generierung eines universellen Makefiles

Michael Wester stellt im Internet ein Perl-Script zur Verfügung, welches ein universelles, für individuelle Bedürfnisse noch adaptierbares Makefile erstellt. Voraussetzung ist, dass auf dem Unix-System Perl installiert sein muss. Testen Sie dies z.B. mit

```
which perl
```

Die Unix-Antwort sollte lauten

```
/usr/local/bin/perl
```

Gibt Ihnen Ihr System einen anderen Pfad an, z.B. `/usr/bin/perl`, so kann man dieser Pfadangabe die 1. Zeile des Perl-Scripts von Michael Wester anpassen.
Das Perl-Script heisst makemake und lässt sich von den Seiten der University of New Mexico herunterladen.

1. Kopieren Sie bitte makemake (http://math.unm.edu/~wester/utilities/makemake) in das Verzeichnis, in dem sich ausschliesslich die Dateien befinden, die Sie compilieren wollen.

2. Machen Sie bitte dieses Perl-Script für Sie ausführbar. Dies geht z.B. mit

   ```
   chmod u+x makemake
   ```

3. Rufen Sie nun das Perl-Skript auf

   ```
   ./makemake gesprog
   ```

Dadurch wird in Ihrem Verzeichnis ein Makefile erzeugt:

```
PROG = gesprog

SRCS = f.f90 main.f90 mittelwert.f90

OBJS = f.o main.o mittelwert.o

LIBS =

CC = cc
CFLAGS = -O
FC = f77
FFLAGS = -O
F90 = f90
F90FLAGS = -O
LDFLAGS = -s

all: $(PROG)
```

```
$(PROG): $(OBJS)
	$(F90) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)

clean:
	rm -f $(PROG) $(OBJS) *.mod

.SUFFIXES: $(SUFFIXES) .f90

.f90.o:
	$(F90) $(F90FLAGS) -c $<
```

In der ersten Zeile wurde hinter `PROG =` der von Ihnen gewüschte Name für das ausführbare Programm (Executable) festgelegt.
Nun können Sie an der Kommandozeile die Make-Utility starten mit

```
make
```

Ausgeführt wird

```
f90 -O -c f.f90
f90 -O -c main.f90
f90 -O -c mittelwert.f90
f90 -s -o gesprog f.o main.o mittelwert.o
```

Nachdem die einzelnen Programmdateien mit dem f90-Compiler übersetzt wurden, werden die Object-Codes zum ausführbaren Programm `gesprog` verknüpft.
Die Syntax des mit dem Perl-Script erzeugten Makefiles ist universeller als die Syntax im ersten Beispiel. Das 2. Makefile enthält Makros und Argumentlisten. Voreingestellt waren im Vergleich zum ersten händischen Beispiel bei der Object-Code-Generierung die Flag zur weitestgehenden Optimierung (`-O`) und beim Linken die Anweisung dass bestimmte symbolische Informationen nicht im Executable vorhanden sein sollen.
Die Compiler- und Linkerflags sollten in den `Makefiles` den indiduellen Bedürfnissen angepasst werden. Weitere Informationen zu den Flags findet man mit

```
man f90
```

bzw. mit

```
man ld
```

Will man genauer wissen, wie Compiler und Linker arbeiten, kann man sowohl bei den `F90FLAGS` als auch bei den `LDFLAGS` die Flag `-v` für *engl. verbose* (wortreich) hinzufügen. Dann kann man auch sehen, dass beim Linken der aus dem Programmcode erzeugte Objectcode mit Systembibliotheken zum ausführbaren Programm zusammengesetzt wird.
Achtung: wenn Sie Ihr Executable mit einem Debugger später weiter untersuchen wollen, sollte als Compilerflag `-g` verwendet und als Linkerflag nicht `-s` verwendet werden.

14.4 Windows: die `make`-Utility beim Compaq Visual Fortran Compiler v6.6

Beim Compaq Digital Visual Fortran Compiler Version 6.6 kann man sich ebenfalls ein Makefile erzeugen lassen und den Compiler änlich wie unter Unix von der Kommandozeile aus einsetzen.
Hierzu kann man folgendes „Rezept“ einsetzen (Verbesserungs- und Optimierungsvorschläge bitte per Mail an die Autorin):

1. Developer Studio starten
2. leeres Projekt (Fortran Console Application) erzeugen und benennen
   - Menü File -> New -> Project -> Fortran Console Application -> Projektnamen einfügen (hier: `testmake`) -> den Speicherort für das Projekt auswählen (und „Create New Workspace und Win32-Platform“ aktiviert lassen)
   - Die Frage „What kind of console application do you want to create?“ mit „An empty project“ beantworten.
3. Im WorkSpace-FileBrowser-Fenster in dem „File“-Register das Projekt aufklappen und „Source Files“ mit der rechten Maustaste anklicken -> „Add files to folder“ wählen
   - entweder: nach und nach einzelne Programmeinheiten neu generieren bzw. aus den Verzeichnissen einfügen (ein Programmpaket darf nur ein Hauptprogramm bei beliebig vielen Unterprogrammen enthalten)
4. Makefile erzeugen
   - in der Menuleiste: Project -> Export Makefile mit „Write dependencies when writing makefiles“ wählen
   - Das von der Entwicklungsumgebung (Developer Studio) erzeugte Makefile (im Beispiel: `testmake.mak`) ähnelt in Struktur und Logik dem Unix-Makefile. Es lässt sich gut mit der Applikation WordPad betrachten und weiterbearbeiten (z.B. andere Compiler-Flags einbauen) In der Entwicklungsumgebung könnte man wie gewohnt weiterarbeiten und bei Bedarf die Compiler-Flags und die Linker-Flags über ->Projekt -> Settings modifizieren.
5. ausführbares Programm generieren
   - Menü: Build -> Build <Name des Projekts`.exe`> (hier: Build `testmake.exe`) wählen. Die einzelnen Teilprojekte werden dann gemäß der im Makefile enthaltenen Regeln compiliert und zu einem ausführbaren Programm gelinkt.
   - Im konkreten Beispiel werden folgende Zwischenschritte ausgeführt:

```
----------Configuration: testneu - Win32 Debug----------
Compiling Fortran...
D:\Program Files\Microsoft Visual Studio\MyProjects\testneu\mittelwert.f90
D:\Program Files\Microsoft Visual Studio\MyProjects\testneu\f.f90
D:\Program Files\Microsoft Visual Studio\MyProjects\testneu\main.f90
```

```
Linking...

testneu.exe - 0 error(s), 0 warning(s)
```

Anhand der Rückmeldung sieht man, dass die drei Quelltext-Dateien compiliert und danach über den Linker miteinander verbunden wurden. Das erzeugte Executable lässt sich wie gewohnt starten.

6. Feineinstellungen / Anpassungen (Beispiele)
   - In der Menüleiste können Sie unter Projekt -> Settings -> Fortran die voreingestellten Compiler-Flags (Project Options) modifizieren.
   - Unter Projekt -> Settings -> Link lässt sich der Name und der Speicherort des Executables einstellen (nimmt man hier keine Änderungen vor, liegt das ausführbare Progamm im Projekt-Unterordner Debug und trägt den Namen des Projekts gefolgt von der Endung .exe.
   - Das aktuelle Arbeitsverzeichnis (Working directory) lässt sich über Menü -> Project Settings -> unter „Debug“ festlegen.
   - Sollten Sie hier &Auuml;derungen vorgenommen haben, empfiehlt es sich, ein aktualisiertes Makefile zu schreiben.
7. **Optional: Compilieren und Linken an der Kommandozeile**
   Man kann bei Bedarf z.B. nachdem man ein Makefile erzeugt hat, das Developer Studio verlassen und an der **Kommandozeile** (in der DOS-Box) weiterarbeiten.
   Dazu geht man in das Verzeichnis, in dem das Projekt abgespeichert wurde und das Makefile liegt. Durch

   ```
   nmake /f <Name des Makefiles> <Name des zu erzeugenden Executables>
   ```

   kann man die Make-Utility unter Windows starten. Im Beispiel wäre das entsprechende Kommando

   ```
   nmake /f testmake.mak
   ```

   Hat man die Default-Einstellungen nicht verändert (d.h. unter 6. keine Anpassungen vorgenommen), so wird im oben betrachteten Beispiel ein Executable mit den Namen `testmake.exe` im Unterordner `Debug` abgelegt. In dem Unterordner Debug findet sich auch der im Zwischenschritt erzeugte Objectcode. Mit

   ```
   cd Debug
   testmake.exe bzw. testmake
   ```

   lässt sich das von der Make-Utility erzeugte Executable aufrufen.
   Verändert man eine Quelltextdatei im ranghöheren Projekt-Ordner (z.B. `f.f90`) kann man erneut

   ```
   nmake /f testmake.mak
   ```

   aufrufen. Nun wird genauso wie unter Unix nur noch aus der abgeänderten Programmeinheit erneut Objectcode erzeugt und dieser mit den bereits vorliegenden Objectcode zu einem (modifizierten) Executable gelinkt.

14.5 Windows: Der CVF-Compiler an der Kommandozeile (in der DOS-Box)

Der Compaq Visual Fortran Compiler lässt sich auch ausserhalb der Entwicklungsumgebung (des Developer Studios) einsetzen.
Dazu startet man ein DOS-Fenster (z.B. je nach Windows-Version findet man im Startmenü unter Programme oder Programme -> Zubehör den Eintrag „Eingabeaufforderung“ oder „Command Prompt“ oder ähnliches und kann damit ein Fenster starten, in dem sich MSDOS-Befehle direkt eingeben lassen. Mit

```
cd < Angabe des Pfades >
```

kann man in das Verzeichnis wechseln, in dem der Quelltext der Fortran-Programm liegt.
Handelt es sich nur um ein einzelnes Fortran 90/95-Programm, z.B. `beispiel.f90`, so kann man dieses mit

```
df beispiel.f90
```

übersetzen, linken und ein ausführbares Programm erzeugen lassen. Das Executable wird als

```
beispiel.exe
```

erzeugt. Die Ausführung lässt sich durch die Angabe Executable-Namens (`beispiel.exe` oder Erweiterung einfach als `beispiel`) starten.
Achtung: damit vor dem ersten Aufruf des Compilers auch alle notwendigen Pfade und Makrovariablen korrekt gesetzt wurden, sollte zunächst die Datei

```
DFVARS.BAT
```

ausgeführt worden sein. Beim erstmaligen Start der Entwicklungsumgebung geschieht dies automatisch und hinterher stehen die benötigten Einstellungen zur Verfügung.
Ganz analog zu UNIX lassen sich auch hier die Compiler-Flags hinter dem Compiler-Aufruf angeben, z.B. um auf Bereichsüberschreitungen in Datenfeldern zu prüfen und um gleichzeitig dem Executable einen anderen Namen zuzuweisen

```
df /check:bounds beispiel.f90 /exe:ausfuehrbar.exe
```

Besteht ein Programm aus mehreren Teilen und sollen diese zu Objectcode compiliert und danach gelinkt werden, kann man dies z.B. sukzessive oder in einem Schritt tun. Für das obige Beispiel könnte man ein einem Schritt

```
df main.f90 f.f90 mittelwert.f90
```

die einzelnen Programmteile in Objektcode compilieren, linken und zusammen mit Systembibliotheken zm Executable `main.exe` verbinden lassen.

```
C:\temp>df main.f90 f.f90 mittelwert.f90
Compaq Visual Fortran Optimizing Compiler Version 6.6
Copyright 2001 Compaq Computer Corp. All rights reserved.
```

```
main.f90
f.f90
mittelwert.f90
Microsoft (R) Incremental Linker Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

/subsystem:console
/entry:mainCRTStartup
/ignore:505
/debugtype:cv
/debug:minimal
/pdb:none
C:\DOCUME 1\btr029\LOCALS 1\Temp\obj37.tmp
dfor.lib
libc.lib
dfconsol.lib
dfport.lib
kernel32.lib
/out:main.exe
```

Soll das Executable einen anderen Namen erhalten, z.B. `gesamt.exe` so wird dieser am Ende angegeben

```
df main.f90 f.f90 mittelwert.f90 /exe:gesamt.exe
```

Der Vorgang liesse sich auch in mehreren Teilschritten ausführen

```
df /compile_only /object:main.obj main.f90 df /compile_only /object:f.obj
f.f90 df /compile_only /object:mittelwert.obj mittelwert.f90 df /link
main.obj f.obj mittelwert.obj /out:gesamt.exe
```

Der CVF-Compiler lässt sich auch unabhängig von der Microsoft Developer Studio - Entwicklungsumgebung an der Kommandozeile einsetzen. Zum Schreiben Ihres Quellcodes genügt ein einfacher Editor z.B. das Notepad.
Weitere Hinweise zu den Compilerschaltern, Optimierungmöglichkeiten, zum Einbinden von Bibliotheksroutinen und der Entwicklung von gemischtsprachigen Programmen aus Fortran 90/95 und C++ findet man in den Hilfeseiten des CVF.

14.6 Windows: Der Salford FTN95 - Compiler an der Kommandozeile

Beispielprogramm:

```
1 module kreisflaeche
2 implicit none
3 save
4 real, parameter :: pi = 3.141593
```

```
5  contains
6          real function area_kreis (r)
7    implicit none
8    real, intent(in) :: r
9    area_kreis = pi * r * r
10   return
11   end function area_kreis
12 end module kreisflaeche
13
14 program kreis
15 use kreisflaeche
16 implicit none
17 real :: radius
18
19 write(*,'(1X,A$)') 'Geben␣Sie␣den␣Radius␣des␣Kreises␣ein:␣'
20 read(*,*) radius
21
22 write(*,*) 'Die␣Flaeche␣des␣Kreises␣betraegt:␣', area_kreis(radius)
23 write(*,*) 'Der␣Umfang␣des␣Kreises␣betraegt:␣␣', 2.0 * pi * radius
24
25 end program kreis
```

Das Programm `kreis` soll nun mit dem Salford FTN95 - Compiler an der Kommandozeile compiliert, gelinkt und ausgeführt werden.
Dazu wird zunächst vom Betriebssystem Windows XP oder Windows 2000 aus das Fenster mit der Eingabeaufforderung geöffnet

- z.B. mit Start -> Programme -> Zubehör -> Eingabeaufforderung
- oder mit Start -> Ausführen -> **cmd** eintippen ->
- oder, falls vorhanden über das Desktop-Icon „FTN95 Command Prompt“ mit dem OK-Button bestätigen

In dem sich öffnenden Fenster könnten die alten DOS-Befehle eingesetzt werden, um z.B.mit

- **dir** (den Inhalt des aktuellen Verzeichnisses anzeigen lassen)
- **cd <Name eines zur Verfügung stehenden Verzeichnisses>** (in das entsprechende Verzeichnis wechseln)

In dem Eingabefenster wird nun mit cd in das Verzeichnis gewechselt, in dem das Programm `kreis` abgespeichert wurde.
Mit folgender Befehlssequenz lässt sich ein Win32-Executable erzeugen und ausführen:

- `ftn95 kreis.f90`
  (**Compilieren** mit dem Salford Fortran95 Compiler, es wird aus dem Programmcode `kreis.obj` und aus dem Modul `kreisflaeche` die Datei `KREISFLAECHE.MOD` (Objectcode) erzeugt)
- `slink kreis.obj -file:kreis`
  (**Linken** des im 1. Schritt erzeugten Objectcodes mit den Betriebsystembibliotheken mit dem Salford Linker)

- `kreis`
  (**Ausführen** des im 2. Schritt erzeugten Executables `kreis.exe`)

Alternativ bietet der Salford-Compiler die Möglichkeit, in einem Schritt die gesamte Sequenz ausführen zu lassen:

- `ftn95 kreis.f90 /link /lgo` (**Compilieren, Linken, Ausführen** in einem Schritt, dabei wird die im Zwischenschritt erzeugte `kreis.obj`-Datei gleich wieder gelöscht, `KREISFLAECHE.MOD` bleibt erhalten)

Die Compiler-Anweisung (Compilerflag) `/lgo` (als Abkürzung für „*load and go*“) startet die sofortige Ausführung des Executables `kreis.exe`. Lässt man `/lgo` weg, hat man 2 Schritte:

- `ftn95 kreis.f90 /link`
  (**Compilieren** und **Linken**)
- `kreis`
  (**Ausführen** des im 1. Schritt erzeugten Executables `kreis.exe`)

14.7 Compiler-Options und Directiven (`options`- und `include`-Anweisung)

Mit `options` lassen sich innerhalb des Programmcodes Anweisungen an den Compiler (Compiler-Flags) eintragen. Wie diese lauten müssen muss im Einzelfall in den Hilfeseiten des jeweiligen Compilers nachgelesen werden.
Die im Programmtext enthaltenen Compiler-Options haben Vorrang vor den an der Kommandozeile eingegebenen oder in der Entwicklungsumgebung eingestellten Flags.
Werden Compiler-Options verwendet, so müssen diese beim Wechsel des Compilers und der Rechnerarchitektur (z.B. von Windows zu Unix) überprüft und in der Regel an das neue System angepasst werden.
Die Compiler-Options sind also nicht standardisiert und Code mit einer `options`-Anweisung ist nicht mehr universell portabel.
Unter Windows kann man z.B. mit `/silent` unterdrücken, dass der Salford Fortran95 Compiler beim Übersetzen Warnings ausgibt. An der Kommandozeile würde man in diesem Fall

```
ftn95 kreis.f90 /silent
```

schreiben. In der Plato3-Umgebung lässt sich diese Option mit Tools -> Options -> Environment -> Project templates -> in der Zeile Check z.B. statt `/CHECK` eintragen. Größ- oder Kleinschreibung wird nicht beachtet.
Um evtl. auftretende Warnings während des Compilierens zu unterdrücken, könnte man - anstatt diese Flag - an der Kommandozeile oder in der Entwicklungsumgebung einzustellen, im Falle des Salford Compilers unter Windows an den Beginn des Programmcodes die Zeile

```
options(silent)
```

stellen. Natürlich könnte man auch

```
options(SILENT)
```

oder

```
OPTIONS(SILENT)
```

schreiben. Der sonst zu den Flags gehörende / fällt auf jedem Fall in der Programmcode-Version weg. Weitere gewüschte Optionen lassen sich hinter durch Kommata getrennt hinzufügen.
Mit der Compiler-Directive include lassen sich während des Compilierens Teile von auf externen Dateien ausgelagerten Quelltexten einbinden. include-Anweisungen dürfen bei Bedarf für alle Stellen des Programmcodes eingesetzt werden. Über include eingebundenene Programmteile dürfen wiederum include-Anweisungen enthalten. Die maximale Schachtelungstiefe ist dabei 10. Im folgenden Beispiel befindet sich in der Datei kreisflaeche_mod.f90 der Quelltext für das Modul kreisflaeche.

```
module kreisflaeche
implicit none
save
real, parameter :: pi = 3.141593
contains
  real function area_kreis (r)
  implicit none
  real, intent(in) :: r

  area_kreis = pi * r * r
  return
  end function area_kreis
end module kreisflaeche
```

Der Inhalt der Datei kreisflaeche_mod.f90 wird im Programm kreis.f90 an der Stelle im Programm, an welcher sonst im Programmcode die Module stehen, eingebunden.

```
options(silent)     ! Salford FTN95-Compiler-Directive:
                    ! waehrend der Compilierung werden Warnings unterdrueckt

include "kreisflaeche_mod.f90"

program kreis
use kreisflaeche
implicit none
real :: radius
real, external :: volume_kugel

write(*,'(1X,A$)') 'Geben␣Sie␣den␣Radius␣ein:␣'
read(*,*) radius
write(*,*) 'Die␣Flaeche␣des␣Kreises␣betraegt:␣', area_kreis(radius)
write(*,*) 'Der␣Umfang␣des␣Kreises␣betraegt:␣␣', 2.0 * pi * radius
end program kreis
```

Sind weitere Teile eines Programmcodes auf externe Dateien ausgelagert, so sind diese an der entsprechend richtigen Stelle mit include einzubinden.
Der Dateinamen hinter include ist als Zeichenkette, d.h. in einfache oder doppelte Anführungszeichen eingeschlossen, anzugeben. Falls die einzuschließende Datei nicht im gleichen

Verzeichnis wie der Programmtext stehen sollte, muss der Pfad, der zur Datei führt, mit angegeben werden.

14.8 Windows: Makefiles für den Salford FTN95 - Compiler

Das bereits bekannte und schon mehrfach verwendete Programm zur Berechnung von Fläche und Umfang eines Kreises sowie des Volumes einer Kugel wurde in 3 Teile zerlegt:

- Das Modul `kreisflaeche` befindet sich in der Datei `kreisflaeche_mod.f90`
- das Hauptprogramm liegt als Datei `kreis4.f90` vor
- Das Unterprogramm mit der Function `volume_kugel` befindet sich im File `volume_kugel.f90`

Wie schon vorher wird das Modul im Hauptprogramm über die Compiler-Directive

```
include "kreisflaeche_mod.f90"
```

eingebunden.
Diesmal müssen jedoch Hauptprogramm und Unterprogramm separat in Objectcode übersetzt und zusammen mit den Betriebssystem-Bibliotheken zu einem ausführbaren Programm verlinkt werden. Per Hand kann man dies an der Kommandozeile mit 4 Befehlen durchführen lassen:

- **Compilieren**:
  - `ftn95 kreis4.f90`
  - `ftn95 volume_kugel.f90`
- **Linken**:
  - `slink kreis4.obj volume_kugel.obj -file:kreis`
- **Ausführen**:
  - `kreis`

Besonders bei größeren Programmpaketen, die aus einer Vielzahl an Einzelbestandteilen bestehen, kann der Vorgang sehr viel unübersichtlicher und umständlicher werden.
Jedoch gibt es auch hier eine gute Nachricht: auch unter Windows ist es inzwischen ähnlich wie unter Unix möglich, compiler- und progammabängige Makefiles zu erzeugen, mit denen sich die Schritte des Compilierens und Linkens automatisieren lassen.
Im Falle des Salford FTN95 - Compilers könnte das Makefile `makefile` (Default-Name für ein Makefile) folgendermaßen aussehen:

```
kreis4.exe: kreis4.obj volume_kugel.obj
      slink kreis4.obj volume_kugel.obj -file:kreis

kreis4.obj: kreis4.f90
```

```
	ftn95 kreis4.f90 /check

volume_kugel.obj: volume_kugel.f90
	ftn95 volume_kugel.f90 /check
```

Analog zu den Makefiles unter Unix werden jeweils in einer Zeile Abhängigkeiten definiert, und in der folgenden Zeile, was zu tun ist. Die Abfolge beginnt mit dem Gesamtschritt und gliedert sich sukzessive in Einzelschritte.
Zum Beispiel bedeuten die Zeilen

```
kreis4.exe: kreis4.obj volume_kugel.obj
	slink kreis4.obj volume_kugel.obj -file:kreis
```

dass das ausführbare Programm `kreis4.exe` von den Objectcode-Dateien `kreis4.obj` und `volume_kugel.obj` abhängt. Die Zeile

```
slink kreis4.obj volume_kugel.obj -file:kreis
```

enthält die Vorschrift wie `kreis4.exe` zu generieren wäre. Mit

```
slink kreis4.obj volume_kugel.obj
```

würde das Executable `kreis4.exe` tatsächlich erzeugt. Jedoch besagt der Zusatz

```
-file:kreis
```

dass das Executable den Namen `kreis.exe` tragen soll. Die beiden folgenden Blöcke definieren analog, von was die beiden Objectcode-Dateien abhängen und wie sie generiert werden sollen.
Der Aufruf des Makefiles erfolgt für den Salford FTN95 an der Kommandozeile aus dem Verzeichnis heraus, in dem sowohl Makefile als auch die Quelldateien liegen müssen, mit

```
mk32
```

Sollte das Makefile einen anderen Namen als `makefile` tragen, so würde entsprechend

```
mk32 -f < Name des Makefiles >
```

an der Kommandozeile verwendet werden können, um die Umsetzung der im Makefile geschriebenen Anweisungen, wie die einzelnen Programmteile compiliert und gelinkt werden sollen, zu starten.
Drei zusätzliche Hinweise:

- Muss in einem Makefile eine Zeile fortgesetzt werden, so wird dies am Ende der Zeile mit einem Backslash \markiert.
- In Unix muss in der 2. Zeile, in der die jeweilige Aktion definiert ist, als 1. Zeichen ein (nicht unmittelbar sichtbares) Tabulator-Zeichen stehen. In dem obigen Makefile kann als 1. Zeichen ein Tabulator-Zeichen stehen.

- Unter Windows sind die Unterschiede in den Compiler- und Linker-Flags, dem Aufbau der Makefiles sowie den einzusetzenden Namen für Compiler, Linker und Make-Utility sehr groß (vergleiche hierzu: Die Make-Utility beim Compaq Visual Fortran Compiler). Die Unterschiede werden noch größer, wenn man von den integrierten graphischen Entwicklungsumgebungen aus mit Projekten arbeitet.

Ähnlich wie unter Unix lässt sich auch hier ein Makefile mit Hilfe eines zugrundeliegenden generellen Syntax allgemeingültiger und kompakter formulieren.
In dem File `default.mk` (Default-Name für die generalisierten Make-Syntax-Regeln beim Salford FTN95-Compiler) werden die allgemeinen Regelsätze zum Compilieren und Linken abgelegt:

```
.SUFFIXES: .f90 .obj .exe

OPTIONS=

OBJFILES=


.f90.obj:
      ftn95 $< $(OPTIONS)

.obj.exe:
      slink $(OBJFILES) -FILE:$@
```

Das zusätzlich notwendige Makefile `makefile`

```
OPTIONS= /check

OBJFILES= kreis4.obj
      volume_kugel.obj

kreis4.exe: $(OBJFILES)
```

ist nun sehr kompakt und bei Bedarf leicht anzupassen. Durch den Aufruf von

```
mk32
```

an der Kommandozeile aus dem Verzeichnis heraus, in dem sich sowohl `default.mk`, das neue `makefile` sowie die Quelldateien `kreis4.f90` und `volume_kugel.f90` befinden müssen, werden die Dateien der Endung `kreis4.f90` und `volume_kugel.f90` mit dem `ftn95` unter Berücksichtigung der Compiler-Flag `/check` in Objectcode compiliert. Aus den Objectcode-Files `kreis4.obj` und `volume_kugel.obj` wird mittels des Linkers slink zusammen mit den notwendigen Betriebssystembibliotheken das Executable `kreis4.exe` erzeugt.

14.9 Allgemeines zu Numerischen Bibliotheken („*Numerical Libraries*“)

Durch den Einsatz von numerischen Bibliotheken lassen sich häufig die Entwicklungszeiten für Programme zur Lösung numerischer Probleme erheblich reduzieren. Im Rahmen

numerischer Bibliotheken stellen Wissenschaftler oder Firmen die von Ihnen entwickelten numerischen Routinen zur Lösung typischer Fragestellungen zur Verfügung. Es gibt z.B. fertig entwickelte Routinen für

- numerische Integration mit adaptiver Schrittweitensteuerung
- Lösung linearer Gleichungssysteme
- Eigenwert- und Eigenvektorenbestimmung von Matrizen
- Fouriertransformationen
- Interpolationen und Extrapolationen von Kurven- und Oberflächen
- Lösung gewöhnlicher Differentialgleichungssysteme
- Lösung partieller Differentialgleichungssysteme
- Erzeugung von Zufallszahlen
- Regressions- und Korrelationsanalyse
- Multivariate Analyse
- Analyse von Zeitreihen
- Bibliotheken für Zahlen höherer Genauigkeit als z.B. reelle Zahlen des Typs `kind(1.0D0) = double precision` (Multi-Precision Library)

Prinzipiell lassen sich 2 Arten numerischer Bibliotheken unterscheiden:

- freie Bibliotheken mit veröffentlichtem Quellcode
- kommerzielle Bibliotheken

Bekannte freie Bibliotheken sind z.B.:

- Numerical Recipies z.B. als Buch mit CD oder unter
  http://lib-www.lanl.gov/numerical/
- Link-Sammlung zu freien numerischen Fortran-Bibliotheken bei
  http://www.fortran.de
- Weitere gängige freie Libraries sind z.B. BLAS, EISPACK, LAPACK, LINPACK, Multi Precision Library ...

Häufig eingesetzte kommerzielle Bibliotheken sind u.a.:

- NAG-Libraries
  http://www.nag.co.uk/numeric/FN/manual/html/FNlibrarymanual.asp
- IMSL Fortran Numerical Library
  http://www.vni.com/products/imsl/fortran/overview.html

Beide Hersteller vertreiben Ihre Produkte für eine Vielzahl an Rechnerarchitekturen und Compiler-Herstellern (u.a. auch für Vektor- und Parallelrechner) und bieten extrem gut ausgereiften Code an.
Der Nachteil der kommerziellen Bibliotheken ist zum einen, dass sie Geld kosten (wobei wir an der Universität Bayreuth eine Campuslizenz der NAG-Libraries zur Verfügung haben, so dass die NAG-Libraries an jedem Lehrstuhl zu minimalen Kosten eingesetzt werden können) und zum anderen, dass der Quellcode der kommerziellen numerischen Libraries nicht eingesehen werden kann und aus Know-How-Gründen nur in einer vorcompilierten Form verkauft wird.
Sowohl bei den kommerziellen als auch den freien Libraries sind die Routinen sehr ausgereift. Die Schnittstellen zu den Unterprogramm-Routinen sind sehr gut dokumentiert, so dass man für die meisten wissenschaftlichen Probleme, solange sie sich in eine Form bringen lassen, die einem „Standard"-Problem mit existierender fertig entwickelter Bibliotheks-Routine entspricht, mittels der Library sehr schnell und ohne größeren eigenen Programmieraufwand zu einer sauberen Lösung kommen kann.
Wie gut die mit den vorgefertigen Routinen gewonnenen Ergebnisse sind, muss natürlich im Einzelfall von dem Forscher genauestens geprüft werden. Ist das Ergebnis positiv, hat sich der Wissenschaftler durch den Einsatz der numerischen Libraries eine Menge an Entwicklungs- und Programmierzeit erspart.
Ist die wissenschaftliche Fragestellung allerdings so beschaffen, dass sie nicht mit vorgefertigten Routinen bearbeitet werden kann, besteht vielleicht noch die Möglichkeit, auf den Source-Code der freien Libraries so zu umzuarbeiten, dass damit die vorliegende Fragestellung gelöst werden kann oder es muss tatsächlich eigener Code zur Lösung der speziellen Fragestellung entwickelt werden.

14.10 Beispiel zum Einbinden der NAG-Libraries unter Unix

Unter `http://btrcx1.cip.uni-bayreuth.de/fl90/un.html` finden Sie die Einstiegs- Dokumentation zu den auf der `btrcx1` installierten NAG-Libraries.
Dokumentation: `http://www.nag.co.uk/numeric/FN/manual/html/FNlibrarymanual.asp`
Das Softwarepaket ist größtenteils installiert in dem Verzeichnis

```
/usr/local/src/nagfl90/fndau04db
```

Die Beispiele finden sich im Unterverzeichnis `examples` und sind identisch mit den auf der Website compilierten.
Die Beispiele (z.B. dasjenige zu `nag_quad_1d_ex01` lassen sich mit

```
nagexample nag_quad_1d_ex01
```

kopieren, compilieren und ausführen.
Ausgeführt wird in dem Beispiel

```
Copying nag_quad_1d_ex01.f90 to current directory
cp /usr/local/src/nagfl90/fndau04db/examples/source/nag_quad_1d_ex01.f90
```

```
Compiling and linking nag_quad_1d_ex01.f90 to produce executable file
a.out
f95 -I/usr/local/lib/fl90_modules nag_quad_1d_ex01.f90 -lnagfl90

Running a.out

a.out
Example Program Results for nag_quad_1d_ex01

a - lower limit of integration = 0.0000
b - upper limit of integration = 6.2832
result - approximation to the integral = -2.54326
```

Bei dem Beispielprogramm `nag_quad_1d_ex01.f90`

```
MODULE quad_1d_ex01_mod

! .. Implicit None Statement ..
IMPLICIT NONE
! .. Default Accessibility ..
PUBLIC
! .. Intrinsic Functions ..
INTRINSIC KIND
! .. Parameters ..
INTEGER, PARAMETER :: wp = KIND(1.0D0)
! .. Local Scalars ..
REAL (wp) :: pi

CONTAINS

FUNCTION f(x)

! .. Implicit None Statement ..
IMPLICIT NONE
! .. Intrinsic Functions ..
INTRINSIC SIN, SIZE, SQRT
! .. Array Arguments ..
REAL (wp), INTENT (IN) :: x(:)
! .. Function Return Value ..
REAL (wp) :: f(SIZE(x))
! .. Executable Statements ..
f = x*SIN(30.0_wp*x)/SQRT(1.0_wp-x*x/(4.0_wp*pi*pi))

END FUNCTION f

END MODULE quad_1d_ex01_mod

PROGRAM nag_quad_1d_ex01

! Example Program Text for nag_quad_1d
! NAG fl90, Release 4. NAG Copyright 2000.

! .. Use Statements ..
USE nag_examples_io, ONLY : nag_std_out
```

```
40 USE nag_math_constants, ONLY : nag_pi
41 USE nag_quad_1d, ONLY : nag_quad_1d_gen
42 USE quad_1d_ex01_mod, ONLY : wp, f, pi
43 ! .. Implicit None Statement ..
44 IMPLICIT NONE
45 ! .. Local Scalars ..
46 REAL (wp) :: a, b, result
47 ! .. Executable Statements ..
48 WRITE (nag_std_out,*) 'Example␣Program␣Results␣for␣nag_quad_1d_ex01'
49
50 pi = nag_pi(0.0_wp)
51 a = 0.0_wp
52 b = 2.0_wp*pi
53
54 CALL nag_quad_1d_gen(f,a,b,result)
55
56 WRITE (nag_std_out,'(/,1X,A,F10.4)') &
57 'a␣␣-␣lower␣limit␣of␣integration␣=␣', a
58 WRITE (nag_std_out,'(1X,A,F10.4)') 'b␣␣-␣upper␣limit␣of␣integration␣=␣', b
59 WRITE (nag_std_out,'(1X,A,F9.5)') &
60 'result␣-␣approximation␣to␣the␣integral␣=', result
61
62 END PROGRAM nag_quad_1d_ex01
```

handelt sich um die Integration einer Funktion `f`

$$\frac{2x\sin(30x)}{\sqrt{4-\frac{x^2}{\pi^2}}}$$

die in dem Programm in dem Modul `quad_1d_ex01_mod` festgelegt und an die NAG- Integrationsroutine durch

```
CALL nag_quad_1d_gen(f,a,b,result)
```

übergeben wird. `a` und `b` sind die Integrationsgrenzen und `result` entspricht dem von der NAG-Routine ermittelten Wert des Integrals.
Ob die NAG-Routine richtig gearbeitet hat, lässt sich z.B. mit Maple überprüfen.
Bei der Compilierung des Programms muss das Verzeichnis, über das die Modul-Definitionen der NAG-Routinen zugänglich sind, eingebunden und die eigentliche NAG-Library (diese liegt unter `/usr/lib` als `libnagfl90.a`) gelinkt werden. Dies geschieht über

```
f95 -I/usr/local/lib/fl90_modules nag_quad_1d_ex01.f90 -lnagfl90
```

Mit diesem Befehl (natürlich geht auch `f90` statt `f95`) würden Sie kompilieren, sobald Sie Änderungen an Ihrem Programm vorgenommen haben.
Ein ergänzender Hinweis:
Falls Sie auf der btrcx1 die Beispiele nachvollziehen wollen, so wird der letzte Befehl von `nagexample` zunächst nicht umgesetzt. Grund ist, dass auf dem Rechner in der tcsh-Shell der lokale Pfad nicht mit im Suchpfad eingeschlossen ist. Sie können dies nachholen, indem Sie an der Kommandozeile

```
setenv PATH .:$PATH
```

eingeben. Dann können Sie Ihr Executable direkt mit

```
a.out
```

aufrufen und brauchen nicht mehr das aktuelle Verzeichnis über

```
./a.out
```

zu referenzieren.

14.11 Verwenden der NAG-Libraries mit dem Salford FTN95 - Compiler unter Windows

Beim Start der Anwendung werden über eine Batch-Datei die Umgebungsvariablen sowohl für den Compiler als auch für die Libraries gesetzt. Mit

```
cd <Name eines Unterverzeichnises>
```

können Sie bei Bedarf in ein von Ihnen angelegtes Unterverzeichnis wechseln.
Mit

```
mkdir <Name eines neuen Unterverzeichnises>
cd <Name des neuen Unterverzeichnises>
```

können Sie ggfs. ein neues Unterverzeichnis anlegen und in diesen wechseln. Zum Beispiel

```
y:\>mkdir nag_test
y:\>cd nag_test
```

Sie können nun beispielsweise ein von NAG zur Verfügung gestelltes Beispielprogramm in das von Ihnen gewählte aktuelle Verzeichnis kopieren und ausführen lassen. Unter Windows geht dies z.B. für das Beispiel `nag_quad_1d_ex01` (weitere Details s.o.) mit

```
nagex_dll nag_quad_1d_ex01
```

Das Beispielprogramm wird in das aktuelle Verzeichnis kopiert, compiliert und ausgeführt. Das Executable wurde ausgeführt und das Ergebnis in die Datei `nag_quad_1d_ex01.r` geschrieben. Hinterher wurde allerdings `nag_quad_1d_ex01.EXE` gleich wieder gelöscht:

```
y:\nag_test>dir

Datenträger in Laufwerk Y: ist USR
Datenträgernummer: 84B4-EF26

Verzeichnis von Y:\nag_test

18.03.2005 10:06     <DIR>      .
18.03.2005 10:06     <DIR>      ..
11.04.2003 06:07         1.815 nag_quad_1d_ex01.f90
18.03.2005 10:07           195 nag_quad_1d_ex01.r
           2 Datei(en)      2.010 Bytes
           2 Verzeichnis(se),      95.227.904 Bytes frei
```

In der DOS-Box kann man sich mit `type <Name eines Programms>` den Inhalt von ASCII-Dateien anzeigen lassen:

```
Y:\nag_test>type nag_quad_1d_ex01.r
Example Program Results for nag_quad_1d_ex01

a - lower limit of integration = 0.0000
b - upper limit of integration = 6.2832
result - approximation to the integral = -2.54326
```

Aus dem obigen Beispiel kann man ableiten, welcher Befehl zum Compilieren und Linken - auch für die selbstentwickelten Programme - eingesetzt werden muss:

```
ftn95 <Name der Fortran90/95-Datei> /LINK /LIBR %NAGFL90DLL%
```

%NAGFL90DLL% ist die NAG-Umgebungsvariable mit der Pfadangabe zu den dynamisch zu linkenden Fortran90-Libraries. Falls Sie bei sich auf dem PC eine lokale Installation der NAG-Libraries für den Salford-Compiler vornehmen, wird der Wert dieser Umgebungsvariablen entsprechend vorbesetzt.
Mit

```
ftn95 <Name der Fortran90/95-Datei> /LINK /LIBR %NAGFL90DLL%
```

kann man installationsunabhängig mit den Salford FTN95-Compiler bei installierten NAG-Libraries (für den Salford FTN95 benötigt man die Version `FNW3204DS` bei anderen Compilern andere) compilieren und linken

```
Y:\nag_test>ftn95 nag_quad_1d_ex01.f90 /LINK /LIBR [FTN95/Win32 Ver. 4.6.0
Copyright (C) Salford Software Ltd 1993-2004]
     Licensed to: Personal Edition User
     Organisation: www.silverfrost.com

     PROCESSING MODULE [ FTN95/Win32 v4.6.0]
          NO ERRORS [ FTN95/Win32 v4.6.0]
     NO ERRORS [ FTN95/Win32 v4.6.0]
     NO ERRORS [ FTN95/Win32 v4.6.0]
Creating executable: nag_quad_1d_ex01.EXE
```

und das Executable ausführen lassen:

```
Y:\nag_test>nag_quad_1d_ex01
Example Program Results for nag_quad_1d_ex01

a - lower limit of integration = 0.0000
b - upper limit of integration = 6.2832
result - approximation to the integral = -2.54326
```

Die Ausgabe erfolgt hier auf der Standardausgabe (in der aktuellen DOS-Box), könnte jedoch mit Hilfe eines > -Zeichens in eine Datei umgeleitet werden, z.B.

```
nag_quad_1d_ex01 > nag_quad_1d_ex01.dat
```

Analog wie unter Unix kann man auch unter DOS statt von der Standardeingabe (der Tastatur) aus einer Datei Werte einlesen (Eingabeumleitung). Dies lässt sich z.B. gut mit dem NAG-Example zum Lösen linearer Gleichungssysteme ausprobieren (`nag_gen_lin_sys_ex01`). Das von NAG zur Verfügung gestellte Programm soll nun dazu hergenommen werden, um eine Lösung zu folgendem Problem zu finden

$$4x + 0.5y + 6z = 26.5$$

$$7x + y + 3z = 24$$

$$3x + 2y + z = 11$$

Dazu braucht das Beispielprogramm nicht verändert, sondern nur mit Hilfe eines Editors (z.B. Notepad unter Windows) eine neue Eingabedatei erstellt und im aktuellen Verzeichnis abgespeichert zu werden.

```
Y:\nag_lin_glsys>cat beispiel.dat
Beispielprogramm fuer ein lineares Gleichungssystem mit 3 Variablen
 3          : Value of n
 'N'          : Value of trans
4.0 0.5 6.0
7.0 1.0 3.0
3.0 2.0 1.0
26.5
24.0
11.0
```

Aus dem NAG-Beispielprogramm wird ein Executables erzeugt

```
ftn95 nag_gen_lin_sys_ex01.f90 /LINK /LIBR %NAGFL90DLL%
```

Das Ergebnis mit den neuen Beispieldaten wird berechnet und in die Datei `beispiel.erg` geschrieben.

```
ftn95 nag_gen_lin_sys_ex01.EXE < beispiel.dat > beispiel.erg
```

Inhalt der Ausgabedatei:

```
Example Program Results for nag_gen_lin_sys_ex01

kappa(A) (1/rcond)
   1.08E+01

Solution
   2.0000
   1.0000
   3.0000

Backward error bound
```

```
      2.02E-17

   Forward error bound (estimate)
      9.15E-15
```

Selbstverständlich lassen sich die NAG-Beispielprogramme (*.f90) den jeweiligen Bedürfnissen anpassen. Auf den Webseiten von NAG (http://www.nag.co.uk/numeric/FN/manual/html/FNlibrarymanual.asp) finden Sie die Dokumentation und wie Sie den Aufruf der numerischen Routinen an Ihre spezifische Problemstellung anpassen können.

Kapitel 15

Ergänzende Bemerkungen zu Datenfeldern

15.1 Untersuchungsfunktionen für Datenfelder

In dem Kapitel zu den Datenfeldern wurde bereits erwähnt, dass es Untersuchungsfunktionen (engl. *inquiry functions*) zu Datenfeldern gibt. Die wichtigsten davon sollen nun nochmal wiederholt werden (Programm `test_array.f90`).

```
1  program test_array
2  implicit none
3  real, dimension (-5:4,3) :: a = 0.0
4  integer, dimension(2) :: lindex, uindex
5
6  write(*,*) 'shape(a)␣␣␣=␣', shape(a)
7  write(*,*)
8  write(*,*) 'size(a)␣␣␣␣=␣', size(a)
9  write(*,*)
10 write(*,*) 'lbound(a)␣␣=␣', lbound(a)
11 write(*,*)
12 write(*,*) 'ubound(a)␣␣=␣', ubound(a)
13 write(*,*)
14 lindex = lbound(a)
15 write(*,*) 'Der␣kleinste␣Zeilenindex␣ist:␣', lindex(1)
16
17 end program test_array
```

Die Bedeutung der einzelnen Funktionen lässt sich direkt aus dem Beispielprogramm ableiten.
Diese Untersuchungsfunktionen werden benötigt, wenn man mit dem Konzept der *assumed-shaped arrays* als formale Parameter in Unterprogrammen arbeiten möchte.

15.2 Übergabe von Datenfeldern an Unterprogramme ohne Größenangaben (engl. *assumed-shape arrays*)

Beim Prinzip der *assumed-shape arrays* (in etwas zu übersetzen mit „Datenfelder als formale Parameter mit übernommener Gestalt“) werden im Unterprogramm die Datenfelder als

formale Parameter ohne Angabe der Indexbereiche in Fortran 90/95 deklariert und formrichtig (gestaltgerecht) an das Unterprogramm übergeben. Dies funktioniert allerdings nur, wenn zusätzlich in der aufrufenden Programmeinheit der `interface`-Block für das Unterprogramm angeführt wird.
Beispielprogramm:

```
program assumed_shape
implicit none

interface
   subroutine maximale_Komponente_suchen(m,v)
   implicit none
   real, intent(in)  :: m(:,:)
   real, intent(out) :: v(:)
   end  subroutine maximale_Komponente_suchen
end interface


integer :: nu, no, mu, mo                    ! Indexbereiche Zeilen/Spalte
real, allocatable, dimension(:,:) :: matrix
real, allocatable, dimension(:)   :: vektor
integer :: error_alloc, i

write(*,*) 'Das␣Programm␣arbeitet␣mit␣der␣Matrix'
write(*,*) '␣␣␣␣␣␣matrix(nu:no,mu:mo)'
write(*,*)
write(*,'(1X,A$)') 'Geben␣Sie␣bitte␣die␣Zeilenindexwerte␣an:␣␣(nu,no):␣'
read(*,*) nu, no
write(*,*)
write(*,'(1X,A$)') 'Geben␣Sie␣bitte␣die␣Spaltenindexwerte␣an:␣(mu,mo):␣'
read(*,*) mu, mo
write(*,*)

allocate(matrix(nu:no,mu:mo),stat=error_alloc)
if (error_alloc /= 0) stop 'Fehler␣beim␣Allokieren.␣Programmabbruch'
allocate(vektor(nu:no),stat=error_alloc)
if (error_alloc /= 0) stop 'Fehler␣beim␣Allokieren.␣Programmabbruch'

write(*,*) 'Die␣Matrix␣wird␣nun␣mit␣Zufallszahlen␣vorbelegt'
call random_seed            ! restart des Zufallsgenerators
call random_number(matrix)  ! 0 <= Zufallszahlen < 1

write(*,'(/1X,A)') 'Ausgabe␣der␣Matrix:'
write(*,*)
do i = nu, no
  write(*,*) matrix(i,:)
end do

write(*,*)
write(*,*)
write(*,*) 'Die␣Matrix␣laesst␣sich␣beschreiben␣durch:'
write(*,*)
write(*,*) 'allocated(matrix)␣=␣', allocated(matrix)
write(*,*) 'shape(matrix)␣␣␣␣␣=␣', shape(matrix)
write(*,*) 'size(matrix)␣␣␣␣␣␣=␣', size(matrix)
```

```
50  write(*,*) 'lbound(matrix)␣␣␣␣=␣', lbound(matrix)
51  write(*,*) 'ubound(matrix)␣␣␣␣=␣', ubound(matrix)
52  write(*,*)
53
54  call maximale_Komponente_suchen(matrix,vektor)
55
56  write(*,'(/1X,A)') 'Ausgabe␣der␣Matrix:'
57  write(*,*)
58  do i = nu, no
59    write(*,*) matrix(i,:)
60  end do
61  write(*,'(/1X,A)') 'Ausgabe␣des␣Vektors␣mit␣den␣maximalen␣Zeilenwert␣:'
62  write(*,*)
63  do i = nu, no
64    write(*,*) vektor(i)
65  end do
66
67  deallocate(matrix)
68  deallocate(vektor)
69  end program assumed_shape
70
71
72  subroutine maximale_Komponente_suchen(m,v)
73  implicit none
74  real, intent(in)       :: m(:,:)          ! assumed-shape array
75  real, intent(out)      :: v(:)            ! assumed-shape array
76  integer, dimension(2) :: lindex, uindex
77  integer                :: i
78
79  write(*,*) '---------Unterprogrammausgabe--------------'
80  write(*,*)
81  write(*,*) 'Ausgabe␣der␣Strukturfaktoren␣der␣Matrix:'
82  write(*,*)
83  write(*,*) 'shape(m)␣␣␣␣␣=␣', shape(m)
84  write(*,*) 'size(m)␣␣␣␣␣␣=␣', size(m)
85  write(*,*) 'lbound(m)␣␣␣␣=␣', lbound(m)
86  write(*,*) 'ubound(m)␣␣␣␣=␣', ubound(m)
87  write(*,*)
88
89  lindex = lbound(m)
90  uindex = ubound(m)
91
92  write(*,*) 'lindex␣=␣', lindex
93  write(*,*) 'uindex␣=␣', uindex
94  write(*,*)
95  write(*,*) 'Ende:---------Unterprogrammausgabe--------------'
96
97
98  ! den Vektor mit der jeweiligen maximalen Zeilenkomponenten belegen
99  do i = lindex(1), uindex(1)
100    v(i) = maxval(m(i,:))
101 end do
102
103 return
104 end subroutine maximale_Komponente_suchen
```

Datenfelder lassen sich mit dem Prinzip der *assumed-shape arrays* elegant an Unterprogramme übergeben und dort weiterverarbeiten, solange im Unterprogramm nur mit relativen und nicht mit absoluten Indices auf die einzelnen Komponenten zugegriffen werden muss. Im Beispielprogramm werden die Maximas der Zeilenkomponenten gesucht. In diesem Fall und in vielen anderen Anwendungsbeispielen spielt es keine Rolle, dass nach Übergabe des Datenfeldes als *assumed-shape array* im Unterprogramm die Indices-Werte mit 1 beginnend gezählt werden.

Kapitel 16

FORTRAN 77 nach Fortran 90/95 - Konverter

Will man alten FORTRAN 77 - Quellcode in Fortran 90/95 - Code umwandeln, müssen zumindestens drei Dinge angepasst werden:

- Die Endung des Dateinamens muss von `.f` bzw. `.for` auf `.f90` umgestellt werden.
- Das Zeichen `C`, welches in FORTRAN 77 in der 1. Zeile für die Kennzeichnung eines Kommentars steht, muss zu `!` gewandelt werden.
- Das `*` in der 6. Spalte, das in FORTRAN 77 - Programmen eine Fortsetzungszeile kennzeichnete muss entweder durch ein `&` am Ende der fortzusetzenden Zeile und/oder durch ein `&` am Beginn der Fortsetzungszeile ersetzt werden.

Dann liegt bereits ein Programm vor, das die Kriterien der Fortran 90 „free source form“ erfüllt. Ein mittels der obigen Änderungen nur minimal angepasstes Programm lässt sich mit einem Fortran 90 - Compiler übersetzen, weil bei der Normung der Fortran 90/95 - Sprache die Rückwärtskompatibiltät zu FORTRAN 77 eingearbeitet wurde.
Die beiden letzten notwendigen Anpassungen kann man z.B. bei kleineren Programmen in einem Editor per Hand durchführen. Bei längeren Programmen kann man einfache Konverter zu Hilfe nehmen. Komplexere Konverter setzen neben diesen syntaktischen Änderungen zusätzlich Programmierkonstrukte vom veralteten FORTRAN 77 - Syntax in Fortran 90 - Syntax um, z.B. durch die Umwandlung geschachtelter Schleifen in logische Strukturen, den Einbau Fortran 90 spezifischer moderner neuer und syntaktisch einfacherer Konstrukte und eine Fortran 90 spezifische Datentypdeklaration bis zur Code-Optimierung.
Folgende FORTRAN 77 - Programme wurden später konvertiert:
`test.f` (Testprogramm 1: ein FORTRAN 77 - Programm aus DO-Schleife und zwei Fortsetzungszeilen)

```
      PROGRAM TEST
C diesmal bewusst ohne IMPLICIT NONE
      INTEGER I
      REAL    X
      DO 10 I=1,10
      X = 10.0
     *          +I
      WRITE(*,*) I, X
```

```
 10    CONTINUE
       WRITE(*,*) 'Dies␣wird␣eine␣Fortsetzungszeile␣ueber␣die␣Spalte␣72
␣␣␣␣␣*␣hinaus'
       END
```

`goto.f`(Testprogramm 2: ein „schlecht“ programmiertes FORTRAN 77-Programm)

```
      PROGRAM F01
      DOUBLE PRECISION X, Y, Z
C
      WRITE(*,*) 'Geben␣Sie␣bitte␣die␣Werte␣fuer␣X␣und␣Y␣ein!'
      READ(*,*) X,Y
      IF ( X .GT. Y ) GOTO 10
      Z = SQRT(X)
      GOTO 20
  10  Z = SQRT(Y)
  20  CONTINUE
      WRITE(*,*) 'Z␣=␣␣', Z
      END
```

`trapez.f` (Testprogramm 3: ein FORTRAN 77 - Programm mit einem formalen Unterprogrammparameter und einem COMMON-Block)

```
      PROGRAM P905M
C orig. Autor: Walter Alt
C
C  Trapezregel
C
      IMPLICIT NONE
      CHARACTER W
      REAL      A,B,R,S
      REAL      TRAPEZ,F,G
      COMMON    R,S
      EXTERNAL  F,G
C
    1 CONTINUE
         WRITE(*,*) 'Intervallgrenzen␣a,b:'
         READ(*,*) A,B
         WRITE(*,*) 'r,s:'
         READ(*,*) R,S
         WRITE(*,*) 'Integral␣von␣f:␣',TRAPEZ(A,B,F)
C
C weiteres Beispiel des Unterprogrammaufrufs, diesmal mit
C der FUNCTION G(X) als aktuellem Parameter
C
         WRITE(*,*) 'Integral␣von␣g:',TRAPEZ(A,B,G)
         WRITE(*,*)
         WRITE(*,*) 'Weitere␣Auswertungen␣(j/n)?'
         READ(*,'(A)') W
      IF (W .EQ. 'j' .OR. W .EQ. 'J') GOTO 1
      END
C
C
      REAL FUNCTION TRAPEZ(A,B,F)
C
C F ist ein formaler Unterprogrammparameter, der beim
```

```
C jeweiligen Unterprogrammaufruf (z.B. aus dem Hauptprogramm heraus
C durch den aktuellen Unterprogrammparameter ersetzt wird)
C
      IMPLICIT NONE
      REAL A,B
      REAL F
      TRAPEZ= 0.5*(B-A)*(F(A)+F(B))
      END
C
C
      REAL FUNCTION F(X)
      IMPLICIT NONE
      REAL X
      REAL R,S
      COMMON R,S
      F= EXP(-R*X**2+S)
      RETURN
      END
C
      REAL FUNCTION G(X)
      IMPLICIT NONE
      REAL X
      G = 1.0*X
      RETURN
      END
```

Übersicht über die eingesetzten Konverter und deren Ergebnisse:
a) convert.f90 (einfacher Freeware-Konverter mit geringfügigem Konversionspotential)
http://www.rz.uni-bayreuth.de/lehre/fortran90/vorlesung/V12/convert.f90
läßt sich z.B. compilieren mit

```
f90 -o f90convert convert.f90
```

Das erzeugte Executable `f90convert` wird daraufhin an der Kommandozeile gestartet (RETURN-Taste). Das laufende Programm erwartet die folgenden Eingaben

1. den Programmnamen ohne die Extension `.f`
2. einen /, um die Defaultwerte (3,10,F,F) zu übernehmen oder
3. die Angabe dieser Werte, wobei die folgenden Default-Werte durch Eingabe eines / übernommen werden können.

Die erste Zahl bestimmt die Anzahl der Spalten, die bei DO oder IF-Anweisungsblöcken eingerückt werden soll. Die zweite Zahl gibt an, wie oft dies bei geschachtelten Anweisungen maximal getan werden soll. Würde der dritte Wert auf T (für `.TRUE.`) gesetzt, so werden Leerzeichen bei nicht standardkonformen Deklarationsanweisungen eingefügt bzw. gelöscht. Falls der 3. Wert auf T gesetzt wurde, bewirkt ein T beim 4. Wert, daß nur Interface-Blöcke generiert werden.
Mit `convert.f90` werden unter anderem angepaßt:

1. Einbau von Leerzeichen nach Schlüsselwörtern

2. `CONTINUE` am Ende einer `DO`-Schleife wird durch `END DO` ersetzt
3. `END`-Anweisungen werden durch `END` <Art des zu beendenden Unterprogrammteils> <Name des Progammteils> beendet

Ergebnisse:

`goto.f90convert.f90`

```
      PROGRAM FO1
      DOUBLE PRECISION X, Y, Z
!
      WRITE(*,*) 'Geben Sie bitte die Werte fuer X und Y ein!'
      READ(*,*) X,Y
      IF ( X .GT. Y ) GOTO 10
      Z = SQRT(X)
      GOTO 20
   10 Z = SQRT(Y)
   20 CONTINUE
      WRITE(*,*) 'Z =  ', Z
      END
```

`test.f90convert.f90`

```
      PROGRAM TEST
! diesmal bewusst ohne IMPLICIT NONE
      INTEGER I
      REAL    X
      DO 10 I=1,10
      X = 10.0                                                        &
     &          +I
      WRITE(*,*) I, X
   10 END DO
      WRITE(*,*) 'Dies wird eine Fortsetzungszeile ueber die Spalte 72  &
     &&hinaus'
      END
```

`trapez.f90convert.f90`

```
      PROGRAM P905M
! orig. Autor: Walter Alt
!
!  Trapezregel
!
      IMPLICIT NONE
      CHARACTER W
      REAL      A,B,R,S
      REAL      TRAPEZ,F,G
      COMMON    R,S
      EXTERNAL  F,G
!
    1 CONTINUE
         WRITE(*,*) 'Intervallgrenzen a,b:'
         READ(*,*) A,B
         WRITE(*,*) 'r,s:'
         READ(*,*) R,S
         WRITE(*,*) 'Integral von f: ',TRAPEZ(A,B,F)
!
```

```
! weiteres Beispiel des Unterprogrammaufrufs, diesmal mit
! der FUNCTION G(X) als aktuellem Parameter
!
         WRITE(*,*) 'Integral von g:',TRAPEZ(A,B,G)
         WRITE(*,*)
         WRITE(*,*) 'Weitere Auswertungen (j/n)?'
         READ(*,'(A)') W
      IF (W .EQ. 'j' .OR. W .EQ. 'J') GOTO 1
      END
!
!
      REAL FUNCTION TRAPEZ(A,B,F)
!
! F ist ein formaler Unterprogrammparameter, der beim
! jeweiligen Unterprogrammaufruf (z.B. aus dem Hauptprogramm heraus
! durch den aktuellen Unterprogrammparameter ersetzt wird)
!
      IMPLICIT NONE
      REAL A,B
      REAL F
      TRAPEZ= 0.5*(B-A)*(F(A)+F(B))
      END
!
!
      REAL FUNCTION F(X)
      IMPLICIT NONE
      REAL X
      REAL R,S
      COMMON R,S
      F= EXP(-R*X**2+S)
      RETURN
      END
!
      REAL FUNCTION G(X)
      IMPLICIT NONE
      REAL X
      G = 1.0*X
      RETURN
      END
```

b) to_f90.f90 (empfehlenswerter Konverter vom Alan J. Miller)
http://www.rz.uni-bayreuth.de/lehre/fortran90/vorlesung/V12/to_f90.f90
Compilieren mit `f90 -o to_90 convert.f90` Das erzeugte Executable `to_f90` wird daraufhin an der Kommandozeile gestartet (RETURN-Taste). Diesmal den Namen des zu konvertierenden FORTRAN 77 - Programms mit der Extension `.f` eingeben. Nach der erfolgreichen Konvertierung einen weiteren Programmnamen eingeben oder mit ^C `to_f90` beenden.
Features:

1. konvertiert die FORTRAN 77 „fixed source form“ in Fortran 90 „free source form“
2. oft wird in DO-Schleifen-Konstrukte die Programmablaufsteuerung über Anweisungsnummern ersetzt durch DO - END DO gegebenfalls ergänzt mit CYCLE und EXIT.
3. arithmetische IF-Anweisungen werden ersetzt durch die IF - THEN – ELSE – END IF - Struktur

4. Datentypenvereinbarungen werden mehr dem Fortran 90 Stil entsprechend vorgenommen, bei Unterprogrammen wird die Datentypdeklaration der formalen Parameter mit dem zugehörigen INTENT - Attribut ergänzt

5. Zusammengehörige Anweisungsblöcke werden eingerückt

Ergebnisse:

goto.to_f90.f90

```
PROGRAM f01

! Code converted using TO_F90 by Alan Miller
! Date: 2001-01-31  Time: 10:11:49

DOUBLE PRECISION :: x, y, z

WRITE(*,*) 'Geben Sie bitte die Werte fuer X und Y ein!'
READ(*,*) x,y
IF ( x > y ) GO TO 10
z = SQRT(x)
GO TO 20
10  z = SQRT(y)
20  CONTINUE
WRITE(*,*) 'Z =  ', z
END PROGRAM f01
```

test.to_f90.f90

```
PROGRAM test

! Code converted using TO_F90 by Alan Miller
! Date: 2001-02-15  Time: 10:39:29

! diesmal bewusst ohne IMPLICIT NONE
INTEGER :: i
REAL :: x
DO  i=1,10
  x = 10.0 +i
  WRITE(*,*) i, x
END DO
WRITE(*,*) 'Dies wird eine Fortsetzungszeile ueber die Spalte 72 hinaus'
END PROGRAM test
```

trapez.to_f90.f90

```
PROGRAM p905m

! Code converted using TO_F90 by Alan Miller
! Date: 2001-02-15  Time: 10:45:50

! orig. Autor: Walter Alt

!  Trapezregel

IMPLICIT NONE
CHARACTER (LEN=1) :: w
```

```
12 REAL :: a,b,r,s
13 REAL :: trapez,f,g
14 COMMON    r,s
15 EXTERNAL  f,g
16
17 1 CONTINUE
18 WRITE(*,*) 'Intervallgrenzen a,b:'
19 READ(*,*) a,b
20 WRITE(*,*) 'r,s:'
21 READ(*,*) r,s
22 WRITE(*,*) 'Integral von f: ',trapez(a,b,f)
23
24 ! weiteres Beispiel des Unterprogrammaufrufs, diesmal mit
25 ! der FUNCTION G(X) als aktuellem Parameter
26
27 WRITE(*,*) 'Integral von g:',trapez(a,b,g)
28 WRITE(*,*)
29 WRITE(*,*) 'Weitere Auswertungen (j/n)?'
30 READ(*,'(A)') w
31 IF (w == 'j' .OR. w == 'j') GO TO 1
32 END PROGRAM p905m
33
34
35 REAL FUNCTION trapez(a,b,f)
36
37 ! F ist ein formaler Unterprogrammparameter, der beim
38 ! jeweiligen Unterprogrammaufruf (z.B. aus dem Hauptprogramm heraus
39 ! durch den aktuellen Unterprogrammparameter ersetzt wird)
40
41
42 REAL, INTENT(IN OUT)                    :: a
43 REAL, INTENT(IN OUT)                    :: b
44 REAL, INTENT(IN)                       :: f
45 IMPLICIT NONE
46
47
48
49 trapez= 0.5*(b-a)*(f(a)+f(b))
50 END FUNCTION trapez
51
52
53 REAL FUNCTION f(x)
54
55 REAL, INTENT(IN)                       :: x
56 IMPLICIT NONE
57
58 REAL :: r,s
59 COMMON r,s
60
61 f= EXP(-r*x**2+s)
62 RETURN
63 END FUNCTION f
64
65 REAL FUNCTION g(x)
66
```

```
67 REAL, INTENT(IN)                              :: x
68 IMPLICIT NONE
69
70
71 g = 1.0*x
72 RETURN
73 END FUNCTION g
```

Besonders beim letzten Beispiel fällt auf, dass `to_f90` im Vergleich zu `f90convert` den Code sehr viel tiefgehender auf Fortran 90-Syntax umstellt.

```
btrzxi> to_f90

Enter name of Fortran source file: trapez.f
No. of lines read = 58
REAL FUNCTION TRAPEZ(A,B,F)
REAL FUNCTION F(X)
REAL FUNCTION G(X)
     Converting DO-loops, 3-way IFs, & computed GO TOs
     Determining INTENTs of dummy arguments
     Writing file: trapez.f90
```

Die Konvertierung gelang im diesem komplexeren Beispiel nicht perfekt:
`f90 trapez.f90` bringt 5 Fehlermeldungen. Zur Behebung muss man jeweils in den Unterprogrammen die Zeile mit IMPLICIT NONE vor die Datentypdeklaration mit dem `INTENT`-Attribut stellen und in der `FUNCTION TRAPEZ` ist `F` ein formaler Unterprogramparameter, der als solcher vor dem hinzugefügten Attribut `INTENT(IN)` befreit und mit dem Attribut `EXTERNAL` versehen werden muss.

c) Vast/77to90
Dies ist ein kommerzieller FORTRAN77 nach Fortran90 Konverter, proprietär und sehr teuer (Abrechnung nach den zu konvertierenden Code-Zeilen, z.B. 5000 Zeilen für 450 US-Dollar, 10 Prozent Nachlass für wissenschaftliche Institutionen) mit graphischem User- Interface.
http://www.crescentbaysoftware.com/vast_77to90.html

Kapitel 17

Mixed-Language-Programming (Einbinden von C/C++ - Routinen in Fortran- Programme)

Ein einfaches Beispiel:
Eine C++-Routine bestimmt nach der Formel von Pythagoras die Länge der Hypotenuse in einem rechtwinkligen Dreieck.

Die C++-Routine `crout.cpp`:

```
// C++ - Routine
#include <iostream.h>
#include <math.h>

extern „C“ void pythagoras (float a, float b, float *c)
{
     *c = (float) sqrt(a*a+b*b);
}
```

Diese C++-Routine soll in ein Fortran 90 - Programm eingebunden werden. (Handelt es sich nicht um C++, sondern um reinen C-Code (`.c`) so sollte `iostream.h` durch `stdio.h` ersetzt werden. Auch wird `extern "C"` vor den C-Routinen nicht gebraucht. Die weiteren Abläufe sind für C++ und C-Programme identisch.
Das Fortran-Programm `fmain.f90`:

```
1  ! Aufrufen einer C++ bzw. C-Routine von Fortran aus
2
3  ! Die C-Routine wird innerhalb eines Modules deklariert
4  module cproc
5  interface  ! die Schnittstelle zur C++-Routine
6     subroutine pythagoras (a, b, res)
7     !DEC$ ATTRIBUTES C :: pythagoras      ! Compiler-Directiven
8     !DEC$ ATTRIBUTES REFERENCE :: res     ! systemabhaengig: hier CVF, Windows
9     real :: a, b, res
10    end subroutine
```

```
11 end interface
12 end module
13
14
15 program fmain
16 use cproc
17 implicit none
18 real :: x, y, z
19
20 write(*,*) ' Berechnung der Hypotenusenlaenge eines rechtwickligen Dreiecks'
21 write(*,*) ' Geben Sie die beiden Seitenlaengen ein , die den rechten Winkel'
22 write(*,*) ' einschliessen:'
23 read(*,*) x, y
24
25 call pythagoras(x,y,z)
26 write(*,*) ' Die Laenge der Hypotenuse betraegt: ', z
27 end program fmain
```

Im Fortran-Programm wird fuer die C-Routine ein Interface-Block definiert, der Angaben zum Datenaustausch zwischen Fortran und C enthält. Dieser Interface-Block wird in ein Modul eingebunden, welches dann wieder von anderem Programmteilen (hier vom Hauptprogramm) genutzt wird.
Weitere Informationen und Unterstützung zu der Thematik erhalten Sie im „Programming Guide“ unter „Mixed Language Programming“. Das Beispiel ist im wesentlichen dem Handbuch vom CVF-Compiler (unter Programming Guide, Mixed Language Programming) entnommen. Dort sind auch die notwendigen Compiler-Directiven angeführt. Die Anweisungen mit !DEC$ sind COMPILER-Directriven Notwendigkeit von Form und Inhalt sind abhängig von den verwendeten Systemen. Für weitere Informationen zum „Mixed-Language-Programming“ mit den dem CVF unter Windows empfiehlt es sich, in den Hilfeseiten des CVFs nachzuschlagen.

Windows: Compilieren mit dem CVF-Compiler (in der Entwicklungsumgebung)

1. Als Pendant zum CVF v6.6 wird als C/C++-Compiler der Microsoft Visual C++ v6.0 benötigt. Dieser muss auf dem Rechner installiert sein.
2. Compilieren in der Entwicklungsumgebung
   (a) leeres Projekt erzeugen (Fortran Console Application) und benennen (z.B. `hypo_test`)
   (b) das `fmain.90` - Programm erstellen bzw. hinzufügen
   (c) in der Menüleiste unter Project -> Export Makefile wählen und `hypo_test.mak` schreiben lassen. Dabei „Write dependencies when writing makefiles“ aktivieren
3. Das Programm „builden“
   - Beim Linken kommt noch eine Warnung. Diese lässt sich vermeiden, wenn man unter Project -> Project Settings auf der „Karteikarte“ mit dem Titel „Link“ das Häkchen bei „Link incrementally“ entfernt.

4. Programm ausführen und testen

   - arbeitet als Fortran Console Application und bringt die richtigen Ergebnisse

Windows: Compilieren mit dem CVF-Compiler (an der Kommandozeile)

1. Als Pendant zum CVF v6.6 wird als C/C++-Compiler der Microsoft Visual C++ v6.0 benötigt. Dieser muss auf dem Rechner installiert sein.

2. Compilieren an der Kommandozeile (in der DOS-BOX)
   Die beiden Programme `crout.cpp` und `fmain.f90` sollen sich im gleichen Verzeichnis befinden.

3. Zur Initialisierung von Pfad- und Makrovariablen muss zunächst die Datei `DFVARS.BAT` aus dem Verzeichnis

   ```
   ...\Microsoft Visual Studio\DF98\BIN
   ```

   ausgeführt werden

4. Compilieren und Linken (Builden)

   - ```
     cl -c crout.cpp
     df fmain.f90 crout.obj /link /out:prog.exe
     ```

   Hier wird zunächst mit der Kommandozeilenform des Intel-C++-Compilers aus der C++-Routine ein Objectcode erzeugt (`crout.obj`). Im zweiten Schritt wird die Fortran 90 - Routine `fmain.f90` mit dem CVF-Compiler (`df`) in Objectcode übersetzt. Die beiden Objectcode-Files werden danach mit dem `df` zusammen mit den notwendigen Systembibliotheken zum ausführbaren Programm `prog.exe` verknüpft.

5. Programm ausführen und testen

   - mit `prog` oder `prog.exe` an der Kommandozeile
   - bringt die richtigen Ergebnisse

Achtung: wurde der 3. Schritt vergessen, erscheint u.U. später beim Linken die Fehlermeldung „LINK: fatal error LNK1181: cannot open input file „dfor.lib““, weil aufgrund der fehlenden Initialisierung der Pfad zu den Fortran-Betriebssystem-Bibliotheken nicht gefunden wird.

Unix: Mixed-Language-Programming
Das Perl-Script **makemake** von Michael Wester (http://math.unm.edu/~wester/utilities/makemake) dazu verwendet werden, aus Quellprogrammen in C, Fortran 90 und FORTRAN 77 ein Makefile für die Compilierung der verschiedenen Quellen zu generieren.
Voraussetzung ist, dass

- auf dem Rechner Perl (und die entsprechenden Compiler) installiert sind
     ```

- das Perl-Script ausfürbar gemacht wurde
  `chmod u+x makemake`
- alle für das Programm benötigten Quelltexte (und nur diese) in einem eigenen Verzeichnis stehen

In unserem Fall handelt es sich um die C-Routine `crout.c`, die der C++-Routine `crout.cpp` entspricht:

```
/* C - Routine */
#include <stdio.h>
#include <math.h>

void pythagoras (float a, float b, float *c)
{
     *c = (float) sqrt(a*a+b*b);
}
```

die in das Fortran 90 - Programm `fmain.f90` (siehe oben) eingebunden werden soll.
Bitte beachten Sie, dass auch unter Unix weiterhin die Compiler-Directiven für den CVF-Compiler verwendet werden. Dies ist möglich und notwendig, weil als Fortran 90 - Compiler auf dem Rechner, auf dem die Compilierung vorgenommen wird, die Unix-Version dieses Compilers im Einsatz ist. Wird ein anderer Fortran 90 - Compiler verwendet, sind die benötigten Directiven auf dem Handbuch des sich im Einsatz befindlichen Compilers zu entnehmen.
Durch den Aufruf von

```
./makemake prog
```

wird das Makefile

```
PROG = prog

SRCS = fmain.f90 crout.c

OBJS = fmain.o crout.o

LIBS =

CC = cc
CFLAGS = -O
FC = f77
FFLAGS = -O
F90 = f90
F90FLAGS = -O
LDFLAGS = -s

all: $(PROG)
```

```
$(PROG): $(OBJS)
     $(F90) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)

clean:
     rm -f $(PROG) $(OBJS) *.mod

.SUFFIXES: $(SUFFIXES) .f90

.f90.o:
     $(F90) $(F90FLAGS) -c $<

fmain.o: fmain.o
```

generiert. Hier kann man noch bezüglich der Compiler und der Compilerflags Änderungen anbringen, z.B. soll als C-Compiler statt dem `cc` der `gcc` verwendet werden. Deshalb wird die Zeile

```
CC = cc
```

durch

```
CC = gcc
```

ausgetauscht. Der Aufruf von

```
make
```

lassen sich die einzelnen Programmteile fehlerfrei compilieren und zum ausführbaren Programm `prog` linken

```
f90 -O -c fmain.f90
gcc -O -c crout.c
f90 -s -o prog fmain.o crout.o
```

Mit dem 1. Befehl wird aus dem Fortran 90 - Programm `fmain.f90` mittels des Compilers `f90` optimierter (Flag `-O`) Objectcode (Flag `-c`, Dateiname `fmain.o`) erzeugt.
Mit der zweiten Anweisung wird mit dem GNU-C-Compiler (`gcc`) der C-Code `crout.c` in optimierten Objectcode (`crout.o`) übersetzt.
Der `f90` wird schließlich herangezogen, um die beiden Objectcode Dateien zusammen mit Systembibliotheken zum ausführbaren Programm `prog` zu linken. Die Flag `-s` stellt eine optimierende Linker-Flag dar und mit `-o` wird gewünschte Name des Executables (hier: `prog`) angegeben.

Anstatt mit dem makemake-Skript von Michael Wester zu arbeiten, hätte man analog an der Kommandozeile diese drei Anweisungen auch direkt eingeben können.
Der Aufruf von

```
./prog
```

startet das Executable und gibt die richtigen Resultate

```
 Berechnung der Hypotenusenlaenge eines rechtwickligen Dreiecks
 Geben Sie die beiden Seitenlaengen ein, die den rechten Winkel
 einschliessen:
3. 4.0
 Die Laenge der Hypotenuse betraegt:   5.000000
```

Fehlen die benötigten Compilerdirectiven, treten beim Linken Fehler auf. Hier ist es also notwendig, die Programme speziell nach den Erfordernissen der Compiler anzupassen. Damit werden diese Mixed-Language-Programmpakete compiler- und betriebssystemabhägig und sich sind mehr so einfach - wie reine Fortran 90 - Programme portierbar.
Unabhängig vom Betriebssystem gilt:
Die Verknüpfung von C bzw. C++-Routinen mit Fortran-Code kann, besonders wenn Zeichenketten, Pointer etc. in den Routinen vorkommen, sehr viel komplexer als in diesem einfachen Beispiel werden.

Literaturverzeichnis

Literaturempfehlungen

1. Stephen J. Chapman: „Fortran 90/95 for Scientists and Engineers“, McGrawHill, 2004
2. Dietrich Rabenstein: „Fortran 90“, Hanser, 2001
3. Nachschlagewerk des Fortran-Sprachstandards für Fortran 90- und Fortran 95- Programmierer „Fortran 95“, RRZN Skript, Hannover Mai 2001

Linksammlung

- Utilities, Tipps, Tricks usw. zu Fortran 90/95 und FORTRAN 77
  http://www.fortran.de
- http://en.wikipedia.org/wiki/Fortran
- Website von Alan J. Miller: Fortran 90 Tools und Routinen (numerische Routinen, Testroutinen, to_f90.f90 - Konverter)
  http://users.bigpond.net.au/amiller
- http://www.nsc.liu.se/~boein/fortran.html
- iX-Artikel von Wilhelm Gehrke: Fortran-90-Compiler für Linux
  http://www.heise.de/ix/artikel/1998/08/068/
- FORTRAN und „technical computing“ (auch zu Matlab, C++, Java, etc.)
  http://www.mathtools.net/

Tabellenverzeichnis

Abbildungsverzeichnis