Programmierung Paralleler Systeme

3. Programmiermodell “Single Programm Multiple Data”

3.1 Fortran95

3.1.10 E/A-System; Dateisystem

3.1.11 Module

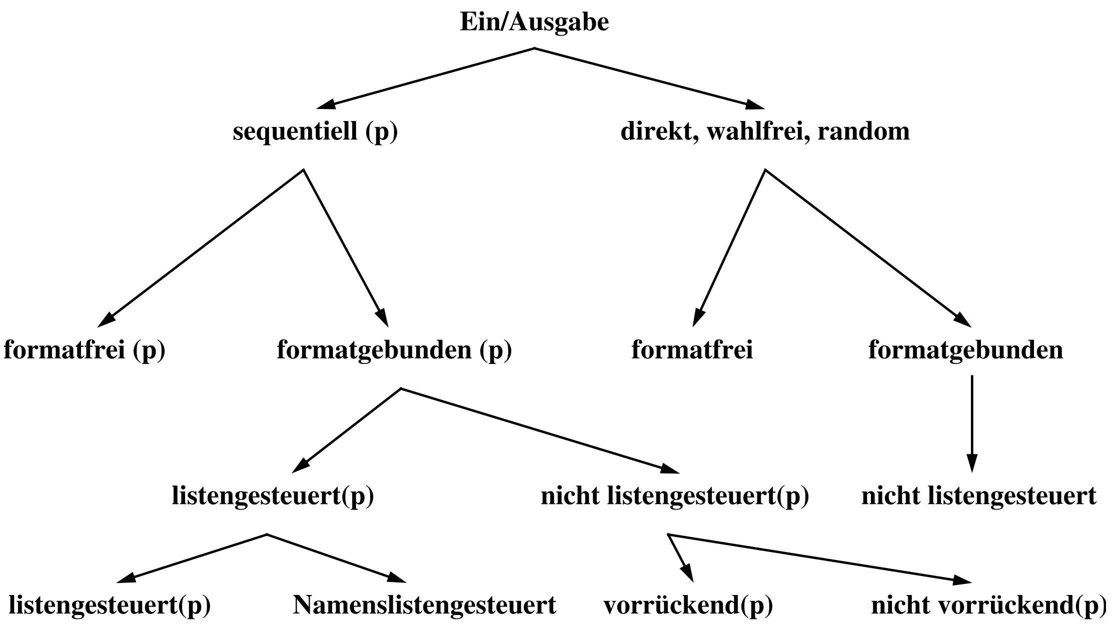
Claus-Uwe Linster

IMMD IV, Lehrstuhl für Betriebssysteme
Universität Erlangen-Nürnberg

linster@informatik.uni-erlangen.de

3.1.10 Dateisystem - FORTRAN90

- Überblick



3.1.10.1 Sequentielle Dateien

- Sequentielle Dateien
  - Datensätze (variabler Länge) können nur in der Reihenfolge gelesen werden, wie sie geschrieben worden sind. Physikalisches Vorbild: Klass. Magnetband.
  - Nach dem Lesen oder Schreiben eines Satzes, bleibt der Schreib-/ Lesezeiger vor dem nächsten Satz stehen; wenn kein Satz mehr vorhanden ist, steht er vor dem Dateiendesatz.
  - Es gibt zwei Positionieranweisungen:

```
REWIND ([UNIT=] fd, [IOSTAT = iostatus, ERR = label])
```

stellt den Schreib-/Lesezeiger an den Anfang der Datei und

```
BACKSPACE([UNIT=] fd, [IOSTAT = iostatus, ERR = label])
```

stellt den Schreib-/Lesezeiger um einen Satz zurück.

Sequentielle Dateien

- Sequentielle Dateien(cont)
- Beispiel: Sie wollen die Sätze 15, 3 und 27 (in dieser Reihenfolge) lesen:

```
Lese satz1 bis satz14; lese satz15 und speichere ihn ab,
dann REWIND; lese satz 1 bis 2; lese satz 3 und speichere ihn ab,
lese satz4 bis satz26, lese satz 27 und speichere ihn ab;...
statt REWIND wäre auch 12 x BACKSPACE möglich gewesen.
```

Sequentielle Dateien

- Sequentielle Dateien(cont)

SATZ 1	SATZ 2	SATZ 3	SATZ 4	SATZ 5	SATZ6

- Schreibt man ab Satz n über bereits vorhandenen Daten, sind sämtliche Sätze hinter Satz n nicht mehr zugreifbar und Satz n ist neu geschrieben.
- Beim Öffnen (OPEN) der Datei wird automatisch an den Anfang positioniert, wenn: `POSITION = 'REWIND'`

  und an das Ende, wenn: `POSITION = 'APPEND'`.

  Die Position wird bei einem "Reopen", nicht verändert, wenn:

  `POSITION = 'ASIS' (default)`

  Der letzte Parameterwert ist für nicht geöffnete Dateien irrelevant, d.h. es existiert keine Zusicherung seitens der Programmiersprache; also systemabhängig, also immer explizit vereinbaren!

3.1.10.2 Direkte Dateien

- Direkte Dateien (wahlfreier Zugriff, random Zugriff)
  - Datenblöcke (fester Länge) können in beliebiger Reihenfolge gelesen bzw. geschrieben werden.
  - Es gibt deshalb auch keine “Position” in der Datei und somit sind auch die Positionieranweisungen REWIND und BACKSPACE sowie die POSITIONS-Parameter bei der OPEN-Anweisung nicht erlaubt.
- Die Festlegung des Dateityps wird mit der (ersten) OPEN-Anweisung getroffen.
  - für den sequentiellen Dateityp

    `ACCESS = 'SEQUENTIELL' (default)`
  - für den direkten Dateityp

    `ACCESS = 'DIRECT'`

(Die Norm erlaubt Dateien, für die der Zugriffstyp nicht ein für allemal festliegt)

3.1.10.3 Dateien-E/A - unformatiert

- Man unterscheidet
  - formatierte Ein-/Ausgabe und
  - unformatierte (formatfreie) Ein/Ausgabe
- Unformatierte Datensätze bzw. Datenblöcke werden in maschineninterner Darstellung gelesen bzw. geschrieben. Es findet keine Datenkonvertierung statt.
  Dieses Verfahren ist sinnvoll dann, wenn:
  - die Daten auf dem gleichen Maschinentyp gelesen und geschrieben werden
  - die Daten in keiner “visuellen” Form lesbar sein müssen
    Diese Daten sind z. B. nicht mit einem Texteditor edierbar!

Verbunden mit der sequentiellen Ein-/Ausgabe ist diese Methode (an sich) das effizienteste Verfahren.

3.1.10.4 Dateien-E/A - formatgebunden

- Formatierte (formatgebundene) Datensätze oder Datenblöcke werden nach Angaben der Formatsteuerung von der maschineninternen Darstellung in eine dem Ausgabemedium entsprechenden (lesbaren) Form umgewandelt (z.B. in den ASCII-Zeichen-Satz), bzw. umgekehrt bei den einzulesenden Daten.
- Dieses Verfahren ist dann sinnvoll wenn:
  - Daten auf Standard-E/A-Geräte ausgegeben bzw. von Standard-E/A-Geräten eingelesen werden; Sichtgerät, Keyboard, Drucker
  - Daten zwischen verschiedenen Systemen (über Datenträger) ausgetauscht werden sollen
  - Daten von anderen System weiterverarbeitet werden sollen und kein gemeinsames anderes “Austauschformat” existiert.
- Man unterscheidet nicht-listengesteuerte und listengesteuerte formatierte E/A, letztere wiederum kann listengesteuert oder namenslistengesteuert sein .

3.1.10.5 E/A-Anweisungen

- Datenübertragung
  Typischerweise werden die E/A-Anweisungen in folgender Sequenz verwendet:
  1. OPEN
  2. READ, WRITE, bei sequentiellen Dateien zusätzlich auch: REWIND, BACKSPACE, ENDFILE
  3. CLOSE (automatisch mit Beendigung des Programms)

  INQUIRE (i/o-control) kann jederzeit auftreten.

E/A-Anweisungen

- OPEN-Anweisung
  - Mit der OPEN-Anweisung wird eine Datei geöffnet, d. h. dem Betriebssystem wird mitgeteilt, daß man auf eine bestimmte Datei zugreifen möchte. Das Betriebssystem trifft die entsprechenden Vorkehrungen und baut - innerhalb des Betriebssystemkerns - entsprechende Verwaltungsstrukturen auf. Um spätere Operationen auf die E/A-Einheiten auch den richtigen Dateien (bzw. Geräten) zuordnen zu können, gibt das Betriebssystem dem Benutzer als Returnparameter einen “handle” zurück, einen sogenannten "filedescriptor", der letztendlich ein Verweis auf die der Datei zugeordneten Kontrollstruktur ist. Dieses “handle” ist über die sogenannte UNIT-Nummer (E/A-Kanal) zugänglich.

```
[UNIT =] u       ! Quelle oder Ziel eines Datentransfers;
                 ! Dateideskriptor
```

  - Mit der OPEN-Anweisung wird der Zugriffsmodus der Datei festgelegt, wobei die Angabe entfallen kann, wenn man eine sequentielle Datei öffnen will.

```
ACCESS = access ! SEQUENTIELL | DIRECT (Zugriffsmethode)
                ! default: SEQUENTIELL
```

© Claus-Uwe Linster – pps3.1-4_spmd_f90: 18/3/98

E/A-Anweisungen

- OPEN-Anweisung (cont)
  - Desweiteren wird festgelegt ob formatiert oder unformatiert geschrieben bzw. gelesen werden soll, wobei die Vorbelegung hier so gewählt ist, daß die häufigsten Kombinationen Zugriffsmethode/Formatierung voreingestellt ist.

```
FORM = form      ! FORMATTED | UNFORMATTED (Strukturspez.)
                 ! default:  FORMATTED if SEQUNTIELL
                 !           UNFORMATTED if DIRECT
```

  - Die maximale Datensatzlänge **kann** bei sequentiellen Dateien mit dem Parameter “RECL=” angegeben werden, bei direkten Dateien **muß** diese Angabe gemacht werde.

```
RECL = recl      ! Datenblocklänge bzw. max. Datensatzlänge
```

  - Wie bestimmt man die Satz- bzw. Blocklänge?
    - Bei unformatierter Ausgabe ist heute die Angabe in Bytes verbreitet. Es ist also für die Längenbestimmung wichtig zu wissen, welche Datentypen wie viele Bytes in der internen Maschinendarstellung belegen (char=1; integer=4; real=4; double prec. = 8) und u. U. welche “Allignments” von der Rechnerarchitektur gefordert werden.

E/A-Anweisungen

- OPEN-Anweisung (cont)
  - Wie bestimmt man die Satz- bzw. Blocklänge? (cont)
    - Bei der formatierten Ausgabe lassen sich die Längen aus den Formatsteuerungsangaben berechnen. (Siehe später)
    - Mit der INQUIRE-Anweisung. Mit Hilfe dieser Anweisung (Funktion) können Sie systemunabhängige Programme schreiben (Siehe F. 44).
  - Die Positionierung des Schreib-/Lesezeigers (nur bei sequentiellen Dateien!) wurde schon auf Seite 8 diskutiert.

```
POSITION = pos     ! ASIS | REWIND | APPEND
```

  - Zur Erhöhung der Datensicherheit gegen versehentliches Schreiben oder nicht gewünschtem Lesen, kann man die E/A-Operationen auf eine Datei einschränken. Im allgemeine ist Schreiben und Lesen erlaubt, jedoch ist diese Voreinstellung nicht von der Norm festgelegt!

```
ACTION = action    ! READ | WRITE | READWRITE
```

  - Mit “FILE” wird der Name der zu öffnenden Datei angegeben. Der Dateiname muß den Konventionen des jeweiligen Systems entsprechen.

```
FILE = file        ! filename
```

E/A-Anweisungen

- OPEN-Anweisung (cont)
  - Mit “STATUS” wird das Verhalten von “OPEN” hinsichtlich bereits vorhandener Dateien gleichen Namens gesteuert:
    - OLD: Datei muß vorhanden sein
    - NEW: Datei darf nicht vorhanden sein
    - REPLACE: Wenn eine Datei vorhanden ist, wird sie gelöscht und mit gleichem Namen als leere Datei neu angelegt.
    - SCRATCH: Es wird eine temporäre Datei (ohne Namen) vom System angelegt und mit dem “CLOSE” oder dem Beenden des Programms wieder gelöscht.
    - UNKNOWN: Hiermit wird ein systemspezifische Voreinstellung verwendet, die nicht von der Norm vorgegeben ist. Da dies der “default” ist, empfiehlt sich grundsätzlich die Spezifikation von “STATUS”!

```
STATUS      ! OLD | NEW | REPLACE | SCRATCH | UNKNOWN
            ! default: UNKNOWN
```


E/A-Anweisungen

- OPEN-Anweisung (cont)
  - Wie bereits bei anderer Gelegenheit besprochen, ist bei Aufrufen von Systemdiensten die Fehlerauswertung von essentieller Bedeutung für die Zuverlässigkeit und Robustheit eines Programms.
    Es werden zwei Methoden angeboten:
    - durch die Angabe eines “labels” bei “ERR” wird im Fehlerfall zu dieser Programmstelle gesprungen und dort eine Fehlerbehandlung mit Auswertung des Fehlercodes in der Stausvariablen erwartet.
    - Besser für eine übersichtliche Programmstruktur ist die unmittelbar nach der Ausführung der E/A-Anweisung sich anschließende Überprüfung der Statusvariablen (“Einseitiges IF”). Sie ist immer dann größer als Null, wenn ein Fehler aufgetreten ist. Man ruft dann eine Fehlerbehandlungsroutine auf, die die Auswertung des Fehlers und die Reaktion darauf übernimmt (Siehe auch Demoprogramme).

```
ERR = label;          !Verzweigungen zur Ausnahmebehandlung
IOSTAT = statusvar    !Ausgabe von E/A-Statusinformation
```

E/A-Anweisungen

- OPEN-Anweisung (cont)
  - Begrenzungszeichen, Leerzeichen, bzw. Füllzeichen

```
DELIM = delim   ! APOSTROPHE | QUOTE | NONE; default: NONE
Blank = blank   ! NULL | ZERO; default: NULL
PAD = pad       ! YES | NO; default: YES
```

  - Beispiele:

```
OPEN (UNIT = fd,FILE = 'datei.data', ACCESS = 'SEQUENTIELL',&
         FORM = 'UNFORMATTED', POSITION = 'APPEND', &
         ACTION = 'WRITE', STATUS = 'OLD', IOSTAT = iostatus)

OPEN (UNIT = fd,FILE = 'datei.data', ACCESS = 'DIRECT',&
         FORM = 'UNFORMATTED', RECL = 256,&
         ACTION = 'READWRITE', STATUS = 'NEW', IOSTAT =
iostatus)

OPEN (UNIT = fd,FILE = 'datei.data', ACCESS = 'SEQUENTIELL'&
         FORM = 'FORMATTED' POSITION = 'REWIND', &
         ACTION = 'READ', STATUS = 'OLD', IOSTAT = iostatus)
```


E/A-Anweisungen

Im folgenden wird nur noch auf die E/A-Parameter eingegangen, die entweder grundsätzlich oder nur in der Semantik verschieden zur OPEN-Anweisung sind.

- CLOSE-Anweisung

Mit der CLOSE-Anweisung wird eine Datei geschlossen. Dabei wird dann automatisch ein Dateiendesatz bei einer sequentiellen Datei geschrieben, wenn die letzte Dateioperation eine Ausgabeoperation war und danach keine Veränderung der Dateiposition erfolgte. CLOSE wird in jedem Fall implizit bei der Beendigung des Programms ausgeführt.

```
[UNIT =] u      ! Quelle oder Ziel eines Datentransfers;
                ! Dateideskriptor
IOSTAT = statusvar! Ausgabe von E/A-Statusinformation
ERR = label;    ! Verzweigungen zur Ausnahmebehandlung
```

Mit dem STATUS-Parameter kann man mit dem CLOSE eine Datei automatisch löschen.

```
STATUS = status !KEEP | DELETE; default: KEEP
```

E/A-Anweisungen

- CLOSE-Anweisung (cont)
  - CLOSE kann (ohne Effekt) auf geschlossene Dateien ausgeführt werden.
  - Beispiele:
  - CLOSE (UNIT = fd, STATUS = 'DELETE', IOSTAT = statusvar)
  - Beim CLOSE ist eine Fehlerbedingung außerordentlich selten, da selbst bei einem CLOSE auf geschlossene Dateien keine Fehlerreaktion auslöst wird.
  - Auch "falsche" Dateien können, wenn sie geöffnet sind, geschlossen werden, da das System die fehlerhafte Verwendung von der "Unitnumber" nicht feststellen kann.

Also weitverbreitet:

```
CLOSE (fd)
```

Aufgabe: Gibt es eine Fehlermeldung, wenn man versucht eine "READ-ONLY"-Datei mit CLOSE zu löschen?

E/A-Anweisungen

- POSITIONIERUNG seq. Dateien
  - Anweisungen zur expliziten Positionierung von Schreib-/Lese-Zeigern in sequentiellen Dateien
    - Setze Dateiposition um einen Satz zurück

```
BACBSPACE ([UNIT =] u [,IOSTAT = statusvar][,ERR = label])
```

- Setze den Schreib-/Lese-Zeiger an den Anfang der Datei

```
REWIND([UNIT =] u [,IOSTAT = statusvar][,ERR = label])
```

- Schreibe einen Dateiendesatz.

```
ENDFILE([UNIT =] u [,IOSTAT = statusvar][,ERR = label])
```

Ein Dateiendesatz wird immer dann implizit geschrieben, wenn nach einer Schreiboperation eine BACKSPACE-Anweisung oder REWIND-Anweisung folgt oder wenn die Datei explizit oder implizit (Programmende) geschlossen wird.

Beispiele:

```
BACKSPACE (UNIT = fd, IOSTAT = iostatus)
REWIND(UNIT = fd, IOSTAT = iostatus)
ENDFILE(UNIT = fd, IOSTAT = iostatus)
```

E/A-Anweisungen

- Datentransfer-Anweisungen

```
READ (ea-parameterliste) [eingabeliste]
READ format [,eingabeliste]
WRITE (ea-parameterliste) [ausgabeliste]
PRINT format [,ausgabeliste]
```

Die jeweilige Kurzform kann nur für die Ein-/Ausgabe auf Standard-E/A-Geräte verwendet werden.

E/A-Parameter: zunächst werden nur die für die unformatierte Ein-/Ausgabe relevanten Parameter besprochen:

```
[UNIT =] u      ! Quelle oder Ziel eines Datentransfers;
                ! Dateideskriptor
```

- Nur bei Dateien mit direktem Zugriff **muß** die Blocknummer angegeben werden

```
REC = datenblocknummer! Datensatznummer eines Datensatzes
```

- Fehlerbehandlung

```
IOSTAT = statusvar! Ausgabe von E/A-Statusinformation
ERR = label     ! Verzweigungen zur Ausnahmebehandlung
```


E/A-Anweisungen

- Datentransfer-Anweisungen (cont)
  - Nur bei Dateien mit sequentiellem Zugriff kann bei der READ-Anweisung auf das Dateiende abgeprüft und bei Erreichen des Dateiendes nach “label” verzweigt werden, um dort ggf. eine Ausnahmebehandlung vorzunehmen.

```
END = label;
```

- Beispiele:

```
WRITE (UNIT = fd, REC = 12, IOSTAT = iostatus) variable
READ (UNIT = fd, REC = 12, IOSTAT = iostatus) variable
WRITE (UNIT = fd, IOSTAT = iostatus) variable
WRITE (UNIT = fd, IOSTAT = iostatus) variable
READ (UNIT = fd, END = 4000 , IOSTAT = iostatus) variable
```

E/A-Anweisungen

- Beispiele (Teil der Demoversionen in ~uelinste/tmp/datei-ea.f90)
  - Beispiel 1: E/A auf sequentielle Dateien, formatfrei

```
SUBROUTINE seqff(feld, max, debug, fd)
! Aus/Eingabe auf eine sequentielle, unformatierte Datei.
! Formalparameter:
INTEGER, INTENT (IN) :: max, fd
LOGICAL, INTENT (IN) :: debug
INTEGER , DIMENSION (max) :: feld

! Lokale Variable:
INTEGER readrec, writerec, statusvar
LOGICAL :: openstatus = .FALSE.

      WRITE  (*, 1200, ADVANCE='NO')
1200  FORMAT ('wie viele Saetze schreiben?: ')
      READ *, writerec
      WRITE  (*, 1300, ADVANCE='NO')
1300  FORMAT ('wie viele Saetze lesen?: ')
      READ *, readrec
```


E/A-Anweisungen

```
      ! Feststellen ob Datei bereits geoeffnet ist

      INQUIRE (UNIT=fd, OPENED = openstatus, IOSTAT = statusvar)
      IF (statusvar > 0)          &
          CALL exception ('INQUI', 'seqff', statusvar)
      If ( .NOT. openstatus) THEN

          ! Oeffnen der Datei

          OPEN(UNIT=fd, FILE = 'seq.ff', ACCESS = 'SEQUENTIAL',&
            FORM = 'UNFORMATTED', POSITION = 'ASIS',    &
            ACTION = 'READWRITE',    &
            STATUS = 'REPLACE', IOSTAT = statusvar)
          IF (statusvar > 0)       &
            CALL exception ('OPEN ', 'seqff', statusvar)
      END IF
```

E/A-Anweisungen

```
! Schreiben auf Datei

DO i = 1, writerec    ! writrec = Anzahl der Saetze
    feld(1) = i + 1000! Kennzeichnung der Satznummer

    ! Schreiben auf Datei, jedes "WRITE" ein Satz

    WRITE (UNIT=fd, IOSTAT = statusvar) feld
    IF (statusvar > 0)      &
      CALL exception ('WRITE', 'seqff', statusvar)
END DO

! Zurücksetzen an den Anfang

REWIND (UNIT=fd, IOSTAT = statusvar)
IF (statusvar > 0)          &
    CALL exception ('REWIN', 'seqff', statusvar)
```


E/A-Anweisungen

```
    ! Lesen von Datei

    DO i = 1, readrec! readrec = Anzahl der Saetze

        ! Lesen von Datei, jedes "READ" ein Satz

        READ (UNIT=fd, IOSTAT = statusvar) feld
        IF (statusvar > 0)      &
          CALL exception ('READ ', 'seqff', statusvar)
    END DO

    ! Nur der letzte gelesene Satz bleibt in "feld" gespeichert

    IF (debug) PRINT *, 'Letzter gelesener Datensatz: ',feld(1)

END SUBROUTINE seqff
```

E/A-Anweisungen

- Beispiel 2: E/A auf direkte Dateien, formatfrei

```
SUBROUTINE dirff(feld, max, debug, fd)

! Aus/Eingabe auf eine random (direct) , unformatierte Datei.
! Formalparameter
INTEGER, INTENT (IN) :: max, fd
LOGICAL, INTENT (IN) :: debug
INTEGER , DIMENSION (max) :: feld

! Lokale Variable:
INTEGER blockbegin, blockend, writerec, statusvar, recordlength
LOGICAL :: openstatus = .FALSE.

! Eingabe Schreibparameter:
      WRITE  (*, 1200, ADVANCE='NO')
1200  FORMAT ('wie viele Bloecke schreiben?: ')
      READ *, writerec
      WRITE  (*, 1300, ADVANCE='NO')
1300  FORMAT ('wie lang soll ein Block sein?: ')
      READ *, recordlength
```


E/A-Anweisungen

```
! Feststellen ob Datei bereits geoeffnet ist

INQUIRE (UNIT=fd, OPENED = openstatus, IOSTAT = statusvar)
IF (statusvar > 0)                  &
   CALL exception ('INQUI', 'dirff', statusvar)
If ( .NOT. openstatus) THEN

   ! Oeffnen der Datei

   OPEN(UNIT=fd, FILE = 'dir.ff' , ACCESS = 'DIRECT',&
      FORM ='UNFORMATTED',POSITION = 'ASIS', &
      ACTION = 'READWRITE', RECL = recordlength,  &
      STATUS = 'REPLACE', IOSTAT = statusvar)
   IF (statusvar > 0)               &
      CALL exception ('OPEN ', 'dirff', statusvar)
END IF
```

E/A-Anweisungen

```
! Schreiben auf Datei

DO i = writerec, 1, -1! writrec = Anzahl der Bloecke
  feld(1) = i + 1000! Kennzeichnung der Satznummer

  ! Schreiben auf Datei, jedes "WRITE" ein Block

  WRITE (UNIT=fd, REC = i, IOSTAT = statusvar) feld
  IF (statusvar > 0)      &
     CALL exception ('WRITE', 'dirff', statusvar)
END DO
```


E/A-Anweisungen

```
      ! Eingabe Leseparameter

      WRITE  (*, 1400, ADVANCE='NO')
1400  FORMAT ('Ab welchen Block lesen? : ')
      READ *, blockbegin
      WRITE  (*, 1500, ADVANCE='NO')
1500  FORMAT ('Bis welchen Block lesen? : ')
      READ *, blockend

      ! Lesen von Datei, jedes "READ" ein Block
      DO i = blockbegin, blockend
          READ (UNIT=fd, REC = i, IOSTAT = statusvar) feld
          IF (statusvar > 0)      &
             CALL exception ('READ ', 'dirff', statusvar)
      END DO

      ! Nur der letzte gelesene Block bleibt in "feld"
gespeichert!

      IF (debug) PRINT *, 'Letzter gelesener Datensatz: ',feld(1)

END SUBROUTINE dirff
```

E/A-Anweisungen

- Formatgebundene E/A
  - Formatgebundene E/A kann ebenfalls für beide Zugriffsverfahren formuliert werden.
  - Außerdem sind Sonderformen für das sequentielle Zugriffsverfahren implementiert und zwar
    - listengesteuert die Methoden
      - listengesteuert E/A
      - namenslistengesteuert E/A
    - und nicht-listengesteuert die Methode
      - nicht vorrückende E/A
  - im folgenden wird zunächst allgemein die formatgebundene Ein-/ Ausgabe behandelt.

E/A-Anweisungen

- Datentransfer-Anweisungen

```
READ (ea-parameterliste) [eingabeliste]
WRITE (ea-parameterliste) [ausgabeliste]

[UNIT =] u          ! Quelle oder Ziel eines Datentransfers;
                    ! Dateideskriptor
REC = datenblocknummer! Datensatznummer eines Datensatzes
IOSTAT = statusvar! Ausgabe von E/A-Statusinformation
ERR = label         ! Verzweigungen zur Ausnahmebehandlung
```

  - Mit dem FMT-Parameter wird die Struktur der zu lesenden oder zu schreibenden Daten spezifiziert.

```
[FMT =] format      ! Formatspezifikation; Strukturinformation
                    ! der zu transferierenden Daten
```

  - Falls der 1. Parameter in der E/A-Parameterliste die UNIT-Nummer ist und ohne “UNIT=” vereinbart, dann darf, wenn die Formatspezifikation der 2. Parameter ist, auch das “FMT=” fehlen.

E/A-Anweisungen

- Es gibt zwei Methoden der Spezifikation
  - mittels der FORMAT-Anweisung, zum Beispiel:

```
   WRITE (UNIT = fd, FMT = 3010,......) e_a-liste
3010 FORMAT (10I6, 3F10.2)
   READ (fd, 3020,......) e_a-liste
3020 FORMAT (5I2, F10.2)
! Sind die Formate gleich, können mehrere Transfer -
! Anweisungen die gleiche Format-Anweisung referenzieren.
```

- mittels einer Zeichenkonstanten, zum Beispiel:

```
WRITE (UNIT = fd, FMT = '(10I6, 3F10.2)',......) e_a-liste
READ (fd, FMT = '(10I6, 3F10.2)',......) e_a-liste

oder auch:

CHARACTER :: format1 *16 = '(10I6, 3F10.2)'
....
WRITE (UNIT = fd, FMT = format1,......) e_a-liste
READ (fd, FMT = format1,......) e_a-liste
```


E/A-Anweisungen

- Die Format-Angaben der E/A-Parameterliste und die Variablen in der E/A-Liste müssen zueinander korrespondieren.
- Die Listen werden von links nach rechts abgearbeitet.
- Textausgabeformate ('abcdef...') werden direkt ausgeführt, das gleiche gilt für Formularsteuerungen (Siehe später).
- Man unterscheidet:
  - Datenfeldbeschreibungen
  - Textausgabeformate
  - E/A-Steuerungsformate

E/A-Anweisungen

Formate

- Überblick (aus RRZN)

Datenfeldbeschreibungen					
Format	Typ	Format	Typ	Format	Typ
A	Zeichen	**ENw.d**	numerisch	**Iw**	numerisch
Aw	Zeichen	**ENw.dEe**	numerisch	**Iw.m**	numerisch
Bw	numerisch	**ESw.d**	numerisch	**Lw**	logisch
Bw.m	numerisch	**ESw.dEe**	numerisch	**Ow**	numerisch
Dw.d	numerisch	**Fw.d**	numerisch	**Ow.m**	numerisch
Ew.d	numerisch	**Gw.d**	bel.vordef.	**Zw**	numerisch
Ew.dEe	numerisch	**Gw.dEe**	bel.vordef.	**Zw.m**	numerisch

In der obigen Tabelle geben Großbuchstaben die Art der Umwandlung an, während Kleinbuchstaben bestimmte vom Programmierer einzufügende Angaben bezeichnen. **w, d, e** und **m** sind vorzeichenlose ganzzahlige literale Konstanten in der Schreibweise ohne Typparameter.

E/A-Anweisungen

Formate(cont)

- **w** gibt die *Datenfeldweite* der externen Darstellung an. Die Datenfeldweite ist gleich der Anzahl aller Zeichen, die bei Abarbeitung der Datenfeldbeschreibung übertragen werden ggf. einschließlich führender Leerzeichen, Vorzeichen, Dezimalpunkt und/oder Exponent. Die Datenfeldweite muß positiv sein.
- **d** gibt die Anzahl der Zeichen des gebrochenen Teilfeld rechts vom Dezimalpunkt an. Die Anzahl der Dezimalstellen darf auch Null sein.
- **e** gibt die Anzahl der Ziffern im Exponenten an. Der Wert von **e** muß positiv sein.
- **m** gibt die Anzahl der Ziffern an, die *mindestens* ausgegeben werden sollen. Der Wert von **m** darf Null sein und muß ≤ **w** sein.

Die zulässigen Werte von **d, e** und **m** sind abhängig vom Wert der Datenfeldweite **w**

E/A-Anweisungen

■ Formate

E/A-Steuerungsformate				Textausgabeformate	
Format	Typ	Format	Typ	Format	Typ
BN	Steuerung der	**nX**	E/A-Tabulator	**cH**	Zeichenausgabe
BZ	num. Eingabe	**Tn**		**'**	
kP	Skalierfaktor	**TRn**		**"**	
SP	Steuerung der	**TLn**			
SS	numerischen	**:**	Formatsteuerung		
S	Ausgabe	/	Datensatzende		

- **k** ist eine ganzzahlige literale Konstante (mit oder ohne Vorzeichen) und heißt **Skalierfaktor**. Der Skalierfaktor muß ohneTypparameter geschrieben werden.
- **n, c** sind vorzeichenlose ganzzahlige Konstanten ungleich Null, die ohne Typarameter geschrieben werden müssen.

E/A-Anweisungen

■ ausgewählte Format-Spezifikationen

Im folgenden sollen beispielhaft die häufigsten Spezifikationen näher besprochen werden, für weitere Formate verwende man das RRZN-Handbuch oder andere einschlägige Literatur.

- X-Format: nX: Die Übertragung des nächsten Zeichens wird an Zeichenposition [+ n nach rechts] fortgesetzt
- A-Format: A Aw : Ein-/Ausgabe von Zeichendaten
  A: Formattyp; w: Datenfeldweite;
  länge: Länge der E/A-Listenelemente

```
Eingabe:
w < länge:linksbündig; Auffüllung mit Füllzeichen
      z.B.<blanks>
w = länge:paßt
w > länge:Es werden 'w' Zeichen übertragen, wovon die
      letzten 'länge' Zeichen gespeichert werden
w = undef:w ist gleich der Länge des Eingabeelements
```

E/A-Anweisungen

- ausgewählte Format-Spezifikationen(cont)

```
Ausgabe:
w < länge:die ersten w Zeichen werden ausgegeben
w = länge:paßt
w > länge:rechtbündig mit führenden Füllzeichen
w = undef:w ist gleich der Länge des Ausgabeelements
```

Beispiele:

```
CHARACTER c *10, cc *9
READ (UNIT=12,FMT='(A8)',IOSTAT=iostatus,ERR=1000 END=2000)c
READ (UNIT=14,FMT='(8X,A)',IOSTAT=iostatus,ERR=1000 END=2000)cc

Getriebegehaeuse: 1. Eingabesatz
Getriebeabstufung:2. Eingabesatz

in c enthalten:Getriebe<b><b>
in cc enthalten:abstufung
```


E/A-Anweisungen

- ausgewählte Format-Spezifikationen(cont)
  - F-Format: Fw.d: reell, doppeltgenau, complex
    - F: Formattyp
    - w: Datenfeldweite
    - d: gebrochener Anteil (rechts vom Dezimalpunkt)

```
Eingabe:[Vorzeichen][Ziffernfolge].d Ziffern [Exponent?]
Beispiel:
Eingabedatenfeld  Format
-534.9            F6.1
.123456           F7.6
Ausgabe:[führende Leerzeichen][Vorzeichen][Ziffernfolge].d
Ziffern
***: wenn Zahl im angegebenen Format nicht darstellbar
Wert         Format      Eingabedatenfeld
21.145       F6.3        21.145
21.145       F11.3             21.145
221.231      F6.3        ******
.221231      F6.3        .221 (ggf. mit führender Null)
221.456      F6.0        221.
```

E/A-Anweisungen

- ausgewählte Format-Spezifikationen(cont)
  - I-Format:Iw;Iw.m:Integer
  - I : Formattyp
  - w: Datenfeldweite
  - m: Anzahl der Ziffern, die mindestens ausgegeben werden müssen; m <= w

```
Eingabe:[führende Leerzeichen][Vorzeichen]Ziffernfolge
            123456789012345678901234567890<--Position
            539   -325       163400001 <--Eingabefeld
            (I3,I6,I7,I10,I4) <--Format

Ausgabe: analog

      [führende Leerzeichen][Vorzeichen]Ziffernfolge

      ***:  wenn Zahl im angegebenen Format nicht
            darstellbar
```


E/A-Anweisungen

- ausgewählte Format-Spezifikationen(cont)
- L-Format:Lw: logische Werte
- Vorzeichensteuerung
  - S: systemabhängige Darstellung
    SP: + wird bei positiven Werten als Vorzeichen ausgegeben
    SS: Vorzeichen wird unabhängig von der Systemvorgabe unterdrückt
- Vorschub/Zwischenraumsteuerung
  - / - Format; Tn TLn TRn - Format; BN BZ-Format
  - Druckersteuerung: "blank", 0, 1, +;
    erzeugt auf dem Drucker: eine Zeile, zwei Zeilen, neue Seite und keinen Vorschub
- Multiplikatoren
  Vor den Format-Angaben können Multiplikatoren stehen, auch Klammerausdrüche sind möglich.
  Beispiel: `3(I7), 5(5F12.1, 7F7.3, I9)`

E/A-Anweisungen

- E/A-Kontrolle
  Mit der INQUIRE-Anweisung lassen sich die aktuellen Eigenschaften einer Datei oder einer Ausgabeliste ermitteln. Es gibt drei Formen der Anweisung:
  - Abfrage über "Unitnumber"
    ```
    INQUIRE ( UNIT =] fd, weitere-parameter)
    ```
  - Abfrage über Dateinamen
    ```
    INQUIRE (FILE = dateiname, weitere-parameter)
    ```
  - und Abfrage über Ausgabeliste
    ```
    INQUIRE (IOLENGTH = length) Ausgabeliste
    ```
- Weitere Parameter
  - Zur Fehlerermittlung
    ```
    IOSTAT = eastatus        ! E/A-Status des INQUIRE-Aufrufs
    ERR = label              ! Verzweigung nach label bei Fehler-
                             ! sitiuation des INQUIRE-Aufrufs
    ```


E/A-Anweisungen

- Weitere Parameter (cont):
  - E/A-Stati der referenzierten Dateien
    ```
    EXIST = existiert        ! Existiert die Datei?; LOGICAL
    OPENED = geoeffnet       ! Datei geoeffnet?; LOGICAL
    NUMBER = unitnb          ! Unitnumber der Datei, bei Abfrage
                             ! File = filenameINTEGER
    NAMED = benannt          ! Benannte Datei?LOGICAL
    Name = filename          ! Dateiname, bei Abfrage
                             ! UNIT = fdCHARACTER
    ACCESS = seqdir          ! Zugriffsmode; undef,
                             ! wenn Datei geschlossenCHARACTER
    SEQUENTIELL = sq         ! Sequentieller Zugriff erlaubt?
                             ! YES | NO | UNKNOWNCHARACTER
    DIRECT = dir             ! Direkter Zugriff erlaubt?
                             ! YES | NO | UNKNOWNCHARACTER
    FORM = fmt               ! Formatierung?
                             ! FORMATTED | UNFORMATTEDCHARACTER
    FORMATTED = form         ! Formatierte E/A?
                             ! YES | NO | UNKNOWNCHARACTER
    UNORMATTED = uform       ! Formatfreie E/A?
                             ! YES | NO | UNKNOWNCHARACTER
    ```

E/A-Anweisungen

- Weitere Parameter (cont):

```
RECL = länge        ! Max. Satzlänge bei sequentiellen Zu-
                    ! griffsmodus oder Blocklänge;INTEGER
POSITION = pos      ! Oeffnungsposition der Datei?
                    ! REWIND | APPEND | ASIS |
UNDEFINED;CHARACTER
ACTION = rdwr       ! Zugriffsoption auf Datei?
                    ! READ | WRITE | READWRITE | UNDEFINED
                              CHARACTER
READ = lesen        ! Zugriffoption = READ?
                    ! YES | NO | UNKNOWNCHARACTER
WRITE = schreib     ! Zugriffoption = WRITE?
                    ! YES | NO | UNKNOWNCHARACTER
READWRITE = rw      ! Zugriffoption = READWRITE?
                    ! YES | NO | UNKNOWNCHARACTER
BLANK = bl          ! Leerzeicheninterpretation?
                    ! NULL |ZERO |UNDEFINEDCHARACTER
DELIM = debegrenz!! Begrenzungszeichen?
                    ! APOSTROPHE | QUOTE | NONE | UNDEFINED
                    !         CHARACTER
PAD = fuell         ! Leerzeichenauffuellung?
                    ! NO | YES  CHARACTER
```


E/A-Anweisungen

- Abfrage über Ausgabeliste
  - Bevor man eine Datei im Zugriffsmodus "DIRECT" öffnet, sollte man mit der INQUIRE-Anweisung die Länge der Ausgabeliste bestimmen lassen:
  - Beispiel:

```
TYPE :: DATUM
      INTEGER :: tag
      CHARACTER *3 :: monat
      INTEGER :: jahr
END TYPE DATUM

TYPE :: MITARBEITER
      CHARACTER *20 name, vorname
      TYPE (DATUM) geburtsdatum
      REAL gehalt
      REAL lohnsteuer
      LOGICAL verheiratet
END TYPE MITARBEITER

TYPE (MITARBEITER), DIMENSION (max1) :: pa
INTEGER reclength
```

E/A-Anweisungen

- E/A-Kontrolle

```
! Ermittlung der Ausgabeblocklänge; ein Element von pa

            INQUIRE (IOLENGTH = reclength) pa(i)
            IF (debug)PRINT *, reclength

! Feststellen ob Datei bereits geoeffnet

      INQUIRE (UNIT=fd, OPENED = openstatus, IOSTAT = statusvar)
      IF (statusvar > 0) CALL exception ('INQUI', 'dirff',
statusvar)
      If ( .NOT. openstatus) THEN

          ! Oeffnen der Datei

          OPEN(UNIT=fd, FILE = 'dir.ff' , ACCESS = 'DIRECT',&
            FORM ='UNFORMATTED',POSITION = 'ASIS', &
            ACTION = 'READWRITE', RECL = reclength,  &
            STATUS = 'REPLACE', IOSTAT = statusvar)
          IF (statusvar > 0) CALL exception ('OPEN ', 'dirff',
statusvar)
      END IF
......
```


E/A-Anweisungen

- listengesteuerte E/A
  - ist eine spezielle Art der sequentiellen, formatgebundenen E/A

```
READ (ea-parameterliste) [eingabeliste]
READ *[,eingabeliste]
WRITE(ea-parameterliste) [eingabeliste]
PRINT *[,eingabeliste]
```

  - die "ea-parameterliste" kann dann entfallen, wenn die E/A-Unit eine Standard- E/A-Gerät ist. Dies wird durch ein "*" gekennzeichnet. (Diese Form haben wir ohne formale Einführung von Beginn an verwendet)
  - wie der Name schon sagt, wird die Formatsteuerung vom System selbst anhand der Datentypen der Daten in der Ein- bzw. Ausgabeliste übernommen.
  - gekennzeichnet ist die listengesteuerte E/A durch die Parameterangabe "FMT = *", außerdem können noch die Parameter ERR, IOSTAT und END gesetzt werden; END nur bei der READ-Anweisung. Im Rahmen dieses Kurses muß der IOSTAT-Parameter angegeben sein. Die Unit-nb. muß angegeben werden, ein "*" bezeichnet die Standard-E/A-Geräte.

E/A-Anweisungen

- listengesteuerte E/A (cont)
  - Beispiele:

```
REAL stromstaerke, spannung
INTEGER, DIMENSION (100) :: feld
TYPE (MITARBEITER)
   CHARACTER *10 name
   CHARACTER *10 vorname
END TYPE
TYPE (MITARBEITER), DIMENSION (100) fabrik

READ (*, FMT = *, END = 1000, IOSTAT = iostatus) &
      stromstaerke, spannung
READ *, stromstaerke, spannung
WRITE (fd, FMT = *, IOSTAT = iostatus)feld
Print *, fabrik(10)
WRITE (*, FMT = *, IOSTAT = iostatus) fabrik(10)
```


E/A-Anweisungen

- listengesteuerte E/A (cont)
  - während man auf die Formatierung der Ausgabe nahezu keinen Einfluß hat muß man für die Eingabe einige "Spielregeln" kennen:
    - die Eingabe ist positionsfrei
    - Ein Eingabesatz enthält eine Folge von Werten, die durch Trennzeichen voneinander getrennt sind und zwar:
      - durch ein oder mehrere "blanks"
      - durch ein Komma; ggf. von "blanks" umgeben
      - durch ein "backslash"; ggf von "blanks" umgeben
    - Wann ein Eingabesatz beendet ist, bestimmt die Formatsteuerung:
    - Bei der Eingabe von einem Keyboard wird mit dem "return" die Zeile als Block übertragen; ist die Zeile zu lang, wird der Rest ignoriert, ist sie zu kurz, wird das "return" als Trennzeichen interpretiert.
    - Weitere Details siehe Handbuch; auch ein kleines Testprogramm ist sehr nützlich!

E/A-Anweisungen

- namenslistengesteuerte E/A (cont)
  - ist eine spezielle Art der sequentiellen, formatgebundenen E/A

```
READ  (ea-parameterliste)
WRITE (ea-parameterliste)
```

  - wie der Name schon sagt, wird die Formatsteuerung vom System selbst anhand der Datentypen der Daten in der Namensliste übernommen.
  - gekennzeichnet ist die namenslistengesteuerte E/A durch die Parameterangabe
"NML = gruppenname", außerdem können noch die Parameter ERR, IOSTAT und END gesetzt werden; END nur bei der READ-Anweisung. Im Rahmen dieses Kurses muß der IOSTAT-Parameter angegeben sein. Die Unit-nb. muß angegeben werden, ein "*" bezeichnet die Standard-E/A-Geräte.
  - Die Ein/Ausgabesätze enthalten am Beginn des Satzes den Gruppennamen mit einem führenden "&"

E/A-Anweisungen

- namenslistengesteuerte E/A (cont)

Beispiel:

```
PROGRAM namlist

IMPLICIT NONE
CHARACTER *10 feld
INTEGER tag, monat, jahr

NAMELIST /datum/ tag, monat, jahr
NAMELIST /auftrag/ feld

DATA feld /'Auftrag 10'/

WRITE (*, NML = auftrag)
PRINT *, 'Gebe ein: &datum tag =?, monat =?, jahr = ? /'
READ (*, NML = datum)
WRITE (*, NML = datum)

END PROGRAM namlist
```

E/A-Anweisungen

- Die nicht vorrückende E/A
  - ist nur bei der sequentiellen, formatgebundenen E/A möglich, jedoch nicht bei listengesteuerter oder namenslistengesteuerten E/A.
  - Eine nichtvorrückemde E/A wird dann ausgeführt, wenn der Parameter **`"ADVANCE = NO"`** gesetzt ist.
  - Sie unterscheidet sich nur dadurch von der vorrückenden E/A, daß bei der nichtvorrückenden E/A nicht automatisch der Schreib-/Lesezeiger auf den nächsten Satz vorgesetzt wird. Dadurch ist es möglich einen Datensatz mit mehrehren E/A-Anweisungen zu schreiben bzw. zu lesen.
  - Beim Schreiben auf das Sichtgerät hat das den Effekt, daß der Cursor am Ende der Zeile stehen bleibt.
  - Bei Angabe des Parameters **`"SIZE= laenge"`** ist der Variablen "laenge" zu entnehmen, wie viele Zeichen bei einer READ-Anweisung tatsächlich gelesen wurden.
  - Bei der Angabe des Parameters "EOR = label" wird dann bei der READ-Anweisung mit der Zeile "label" fortgefahren, wenn das Ende eines Satzes erreicht wurde.

E/A-Anweisungen

- Bei der internen E/A
  - wird auf sogenannte Intern-(Speicher-) Dateien geschrieben bzw. von ihnen gelesen.
  - Interndateien ist eine Zeichenvariablen normalen Zeichentyps, in der Regel ein Feld.
  - Diese Dateien können nicht geöffnet oder geschlossen werden, auch haben sie keine eigentliche Position.
  - Näheres siehe bei Bedarf im Manual

E/A-Anweisungen

- Beispiel Formattest

```
PROGRAM format_test
IMPLICIT NONE

REAL a1,a2,a3
INTEGER b
CHARACTER *10 c
      a1 =7.1234567
      a2 =7.1234567
      a3 =7.1234567
      b = 10
      c = 'abcdefghij'

      Print *, a1,a2,a3,b,c

      Print *, 'Format (SP, F10.5, EN10.5, ES10.5, I10, 3X, A9)'
      Write (*, 1000)a1,a2,a3,b,c
1000  Format (SP, F10.5, EN14.5E3, ES14.5, I10, 3X, A9)
```


E/A-Anweisungen

```
! Programmende: "Cntrl C"

      DO WHILE (.TRUE.)
          READ (*, 2000, END=3000) a1,a2,a3,b,c
2000      FORMAT(F10.5, EN14.5E3, ES14.5, I10, 3X, A9)

3000      Write (*, 4000)a1,a2,a3,b,c
4000      Format (SP, F10.5, EN14.5E3, ES14.5, I10, 3X, A9)
      END DO
END PROGRAM format_test

Screen:

PRINT: 7.1234565   7.1234565   7.1234565 10 abcdefghij
PRINT: Format (SP, F10.5, EN10.5, ES10.5, I10, 3X, A9)
WRITE: +7.12346 +7.12346E+000  +7.12346E+00      +10   abcdefghi
Read: +321.123 +7.12E+001     +7.12346E+00       -10   zasdcz
WRITE:+321.12299+71.20000E+000  +7.12346E+00       -10   zasdcz
```

3.1.11 Module in Fortran90

- Was leistet Fortran90 im Hinblick auf strukturierte bzw. objektorientierte Programmierung?
  - Möglichkeiten der Kapselung; *Information Hiding*
  - Möglichkeit der Implementierung von Objekten
  - keine Instantiierung
    - muß explizit implementiert werden
    - oder kann statisch (zum Compilierzeitpunkt) durch Variation der globalen Namen simuliert werden, was natürlich heißt, daß der Code auch x-mal vorhanden ist - eine Notlösung, jedoch immer noch besser als x Einzellösungen!
  - keine Klassen
  - Trotz der Einschränkungen - wer weiß wie Fortran9x aussehen wird! - sollte man sich beim Programmentwurf der Stilmittel des objektorientierten Ansatzes bedienen und prinzipiell so vorgehen. Wir werden das hier am Beispiel "Datenbasen mit verkettete Liste" demonstrieren.

Module in Fortran90

- Defintion von Modulen

```
MODULE modulname

Spezifikationsteil

Contains
...
END MODULE modulname
```

  - Module sind getrennt übersetzbar: name.f90 --> name.o

- Bindung von Programmen
  - Bei getrennt übersetzten Programmen kennt der Compiler jeweils nur die "Umgebung" des Programm, das übersetzt wird. Bleiben Referenzen offen, so erkennt der Binder(ld), daß er kein vollständiges Ladeobjekt generieren kann. Die nicht abgesättigten Referenzen werden aufgelistet. Will ein Programm ein Modul "benutzen" wird dies durch die Anweisung

```
USE modulename1 [,modulname2] ...
```

angegeben.

Module in Fortran90

- Schnittstellenspezifikation eines Modul
  - im Prinzip ist ein Modul wie ein Hauptprogramm aufgebaut.
  - Durch die Anweisungen bzw. Attribute PRIVATE und PUBLIC wird festgelegt welche Datenbereiche und Unterprogramme von außen zugreifbar sind.
- Infolge des bereits in diesem Kapitel gesagten, sollte man folgende Vorgehensweise wählen:

(1) der Zugriff auf die Datenbasis sollte von außen her nicht möglich sein.

(2) Aufteilung der Unterprogramme nach interne Unterprogramme

- im Sinne von Hilfsroutinen, deren Funktion ein Außenstehender nicht wissen darf, weil sie z. B. geeignet sind, die Datenkonsistenz zu zerstören oder auch nur Implementierungsdetails unerwünscht zu nutzen und

(3) Unterprogrammen (Methoden), die den Zustand des Moduls im Sinne einer “Benutzung” ändern. Diese Unterprogramme müssen so konstruiert sein, daß sie “robust” gegen Fehlbenutzung sind.

Module in Fortran90

- Im Rahmen dieser Vorlesung werden Module wie folgt entworfen:

```
MODULE mn
IMPLICIT NONE
PRIVATE                       ! Keine Außenbeziehungen
! Explizite Ausweisung der Schnittstelle nach außen:
PUBLIC mn_subname1, mn_subname2, mn_subname3,....
PUBLIC mn_funcname1, mn_funcnmame2,...

Spezifikationsteil
Contains
SUBROUTINE mn_subname1 ()
...
END MODULE mn
```

- Es empfiehlt sich die globalen Namen mit einem Prefix, z. B. den Modulnamen zu versehen. Damit wird sichergestellt, daß globale Namen nicht mehrfach verwendet werden - was erst zum Bindezeitpunkt bemerkt werden würde.
- Interne Hilfsfunktionen erhalten dieses Prefix nicht, was die Übersicht und Lesbarkeit des Programms erhöht.

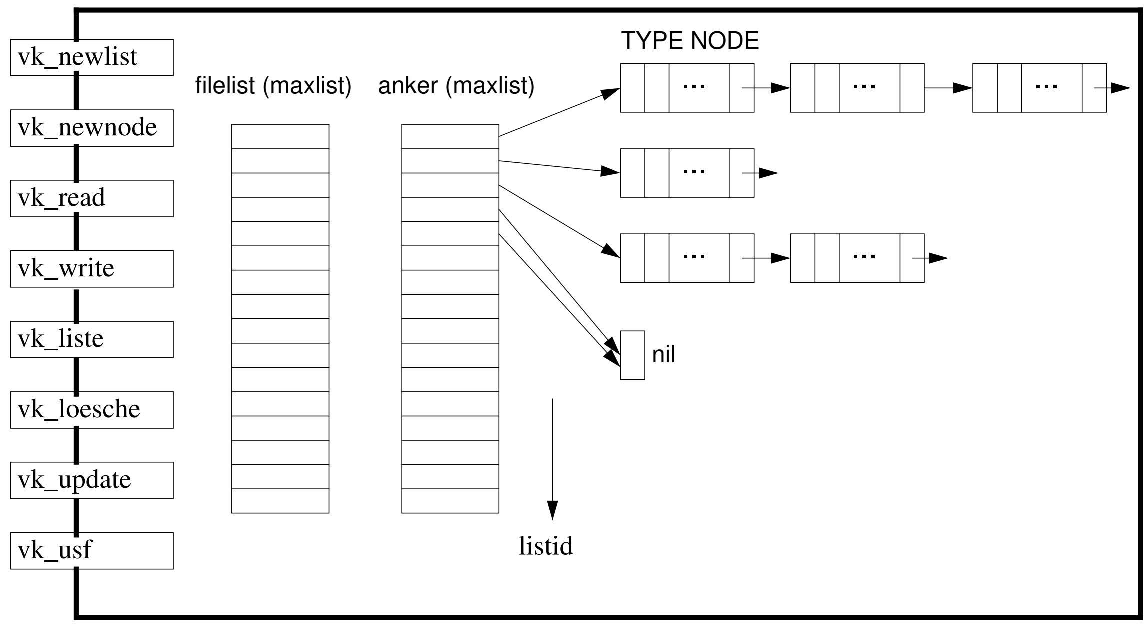
Module in Fortran90

- An sich überflüssig zu erwähnen, daß das Modulkonzept sich natürlich in Fortran90 auch beliebig anders verwenden läßt:
  - z. B. zur Bereitstellung globaler Daten, statt eines COMMON-Bereichs
  - mit USE ist auch selektive Zugriffsvereinbarung auf Datenbereiche möglich
  - als Sammlung von beliebigen Unterprogrammen
  - Globale Definitionen von Datenstrukturen, um fehlerhafte Vereinbarungen in größeren - z. B. von mehreren Programmierern zu entwickelnden -Programmsystemen, zu vermeiden.
- Übersetzen, Binden
  - Die einzelnen Module können getrennt übersetzt werden und als .o-Files später gebunden werden. Bei der Reihenfolge der Module sollten sie in “umgekehrter Definitionshierachie” aufgelistet werden, d.h. das Hauptprogramm zuletzt. Z. B.:

```
f90 vk_modul.f90 vk_main.f90 – o vk.out oder
f90 vk_modul.o vk_main.f90 – o vk.out (vk_modul ist bereits
                                        übersetzt)
```


Module in Fortran90

- Beispiel: "Dynamische Datenbasis persistenter Objekte"



Module in Fortran90

```
PROGRAM dyn_datenbasis

USE vk

IMPLICIT NONE
! Test- und Demoprogramme zur Vorlesung "Programm- und
! Datenstrukturen in FORTRAN90"
! und "Einführung in die Programmierung (FORTRAN90)"
! Copyright: C. U. Linster

INTEGER currentid
INTEGER, PARAMETER :: maxlist = 10
INTEGER listen(maxlist)
DATA listen /10*0/

CHARACTER *7 funktion
LOGICAL :: debug = .TRUE.
LOGICAL :: ende = .FALSE.
LOGICAL flag
INTEGER i
```


Module in Fortran90

```
DO WHILE (.NOT. ende)          ! Funktionen
PRINT *, ' '
PRINT *, 'Eingabe der gewünschten Fkt:'
PRINT *, ' '

PRINT *, 'newlist = Instantiierung einer neuen Liste '
PRINT *, 'readdat = Lese von Datei und Init. Datenbasis'
PRINT *, 'newnode = Eingeben eines neuen Knotens in verk.&
                                                  Liste'
PRINT *, 'l_print = Liste verkettete Liste-Daten'
PRINT *, 'lupdate = Suche Datensatz und aendere Daten'
PRINT *, 'ldelete = Loesche Datensatz'
PRINT *, 'writdat = Schreibe Datenbestand in Datei'
PRINT *, 'debugon = debug ein'
PRINT *, 'debugof = debug aus'
PRINT *, 'endprog = Beenden des Programms'
PRINT *, ' '
PRINT *, 'Funktion:'
READ *, funktion
PRINT *, ' '
```

Module in Fortran90

```
   SELECT CASE (funktion)
      CASE('newlist')
         flag = .TRUE.
         DO i = 1, maxlist
               IF ((listen(i) == 0) .AND. (flag)) THEN
               listen(i) = vk_newlist()
               currentid = listen(i)
               flag = .FALSE.
               IF (debug) PRINT *, 'current_id = ', currentid, i
               END IF
         END DO
         IF (flag) Print *,'No more lists!'
      CASE ('newnode')
         CALL vk_newnode(currentid)
      CASE ('l_print')
         CALL vk_liste(currentid)
      CASE ('readdat')
      flag = .TRUE.
         DO i = 1, maxlist
               IF ((listen(i) == 0) .AND. (flag)) THEN
               listen(i) = vk_read()
```


Module in Fortran90

```
               currentid = listen(i)
               flag = .FALSE.
               IF (debug) PRINT *, 'current_id = ', currentid, i
               END IF
         END DO
         IF (flag) Print *,'No more lists!'
      CASE ('writdat')
         CALL vk_write (currentid)
      CASE ('debugon')
         debug = .TRUE.
      CASE ('debugof')
         debug = .FALSE.
      CASE ('endprog')
         PRINT *, 'Programmende'
         ende = .TRUE.
      CASE DEFAULT
         PRINT *, 'Falsche Eingabe!'
      END SELECT
END DO

END PROGRAM dyn_datenbasis
```

Module in Fortran90

```
MODULE vk ! Implementiert verkettete Listen persistenter
          ! Informationen

IMPLICIT NONE

! Test- und Demoprogramme zur Vorlesung
! "Programm- und Datenstrukturen in FORTRAN90" und
! "Einführung in die Programmierung (FORTRAN90)"
! Copyright: C. U. Linster
! ACHTUNG: Dieses Programm ist nicht voll ausgetestet;
! insbesondere ist
! es nicht robust gegen falsche Eingabesequenzen --> Aufgabe

PRIVATE              ! Kapselung

PUBLIC :: vk_newnode, vk_read, vk_write, vk_liste, vk_newlist,
vk_loesche
PUBLIC :: vk_update
PUBLIC :: listid
```


Module in Fortran90

```
   TYPE :: NODE
      CHARACTER *10 info1
      CHARACTER *10 info2
      CHARACTER *10 info3
      TYPE (NODE), POINTER :: nachfolger
   END TYPE NODE

   INTEGER, PARAMETER :: maxlist = 10
   TYPE :: A
      TYPE (NODE), POINTER :: ancor
   END TYPE

   TYPE (A), DIMENSION (maxlist) :: anker
   TYPE (NODE), TARGET :: nil
   CHARACTER *10, DIMENSION (maxlist) :: filelist
   INTEGER, DIMENSION (maxlist) :: fdlist
   TYPE (NODE), POINTER :: neu, zp, zv
   INTEGER listid
   INTEGER, PARAMETER :: fd = 10
   LOGICAL :: debug = .TRUE.
   LOGICAL :: initialize = .FALSE.
```

Module in Fortran90

▲ init ()

```
CONTAINS
SUBROUTINE init ()

! Initialisiert die Datenbasis ; speziell das Ankerfeld
! Es muß sichgestellt werden, dass diese Routine genau einmal –
! unmittelbar vor Ausfuehrung der ersten Operation – ausgeführt
wird.

INTEGER i

    IF (.NOT. initialize) THEN
       DO i = 1, maxlist
          NULLIFY (anker(i)%ancor)
       END DO
    ELSE
       PRINT *, 'Database already initialized'
    END IF
    initialize = .TRUE.

END SUBROUTINE init
```


Module in Fortran90

▲ INTEGER FUNCTION vk_newlist ()

```
INTEGER FUNCTION vk_newlist ()
! Legt neue Liste an und gibt Listenidentifier zurück
! Lokale Variable
INTEGER i
CHARACTER *10 filename
    IF (.NOT. initialize) CALL init ()
     WRITE  (*, 1010, ADVANCE='NO')
1010 FORMAT ('Filename, Format: xxxxxx.yyy: ')
     READ *, filename
! Suche freien Anker; damit ist die listid festgelegt
     DO i = 1, maxlist
       IF (.NOT. ASSOCIATED (anker(i)%ancor)) THEN
          anker(i)%ancor => nil
          filelist(i) = filename
          fdlist(i) = fd + i
          vk_newlist = i
          IF (debug) PRINT *, 'vk_newlist = ',vk_newlist
          EXIT
       END IF
     END DO
END FUNCTION vk_newlist
```

Module in Fortran90

- INTEGER FUNCTION vk_read ()

```
INTEGER FUNCTION vk_read ()
! Liest Daten von Datei "filename" ein und baut Datenbasis als
! verkettete Liste auf.
! Lokale Variable
INTEGER i, statusvar, listid
CHARACTER *10 filename
LOGICAL openstatus
    IF (.NOT. initialize) CALL init ()
    WRITE  (*, 1010, ADVANCE='NO')
1010FORMAT ('Filename, Format: xxxxxx.yyy: ')
    READ *, filename
! Suche freien Anker; damit ist die listid festgelegt
    DO i = 1, maxlist
       IF (.NOT. ASSOCIATED (anker(i)%ancor)) THEN
          anker(i)%ancor => nil
          filelist(i) = filename
          fdlist(i) = fd + i
          listid = i
          EXIT
       END IF
    END DO
```


Module in Fortran90

- INTEGER FUNCTION vk_read ()

```
INQUIRE (FILE = filelist(listid), OPENED = openstatus, &
       IOSTAT = statusvar)
    IF (statusvar > 0) CALL exception ('INQUI', 'vk_read',&
                                                   statusvar)
       If ( .NOT. openstatus) THEN

    ! Oeffnen der Datei

       OPEN(UNIT=fdlist(listid), FILE = filelist(listid), &
          ACCESS = 'SEQUENTIAL', FORM ='UNFORMATTED', &
          POSITION = 'REWIND',ACTION = 'READ', &
          STATUS = 'OLD', IOSTAT = statusvar)
       IF (statusvar > 0) CALL exception &
                             ('OPEN ', 'vk_read', statusvar)
    ELSE
       PRINT *, 'WARNING: File already open'
       REWIND (fdlist(listid))
    END IF
```

Module in Fortran90

▲ INTEGER FUNCTION vk_read ()

```
!    Einlesen bis Dateiende
        i = 0
        DO WHILE (.TRUE.)
           ALLOCATE (neu);
           READ (UNIT=fdlist(listid), IOSTAT = statusvar,
                 END = 2000) neu%info1, neu%info2, neu%info3
           IF (debug) Print *,neu%info1, neu%info2, neu%info3
           IF (statusvar > 0) CALL exception ('READ ', 'vk_read',&
                                               statusvar)
           i = i + 1
           CALL einfuegen (anker(listid)%ancor, neu)
        END DO

2000    PRINT *, 'Datenbasis hat ', i, 'Eintraege'
        DEALLOCATE (neu)
        CLOSE (fdlist(listid))
        vk_read = listid
        IF (debug) PRINT *, 'vk_read = ',vk_read

END FUNCTION vk_read
```


Module in Fortran90

▲ SUBROUTINE vk_newnode (listid)

```
!SUBROUTINE vk_newnode (listid)

! Erzeugt eine im Dialog angebbare Anzahl
! neuer Listenelemente und liest im Dialog Datensaetze ein.

! Formalparameter:
INTEGER, INTENT (IN) :: listid
! Lokale Variable
INTEGER i, max

          WRITE  (*, 1010, ADVANCE='NO')
1010      FORMAT ('Anzahl der Datensaetze: ')
          READ *, max
          DO i = 1, max
                ALLOCATE (neu)
                WRITE  (*, 1020, ADVANCE='NO')
1020            FORMAT ('info1; max. 10 Char: ')
                READ *, neu%info1
                WRITE  (*, 1030, ADVANCE='NO')
1030            FORMAT ('info2; max. 10 Char: ')
```

Module in Fortran90

▲ SUBROUTINE vk_newnode (listid)

```
                READ *, neu%info2
                WRITE  (*, 1040, ADVANCE='NO')
1040            FORMAT ('info3; max. 10 Char: ')
                READ *, neu%info3
                CALL einfuegen (anker(listid)%ancor, neu)
          END DO

END SUBROUTINE vk_newnode
```


Module in Fortran90

▲ SUBROUTINE einfuegen (anker, neu)

```
SUBROUTINE einfuegen (anker, neu)

! Formalparameter:
TYPE (NODE), POINTER :: anker
TYPE (NODE), POINTER :: neu

! Lokale Variable:

TYPE (NODE), POINTER :: z
    IF (.NOT. ASSOCIATED (anker)) THEN
       PRINT *, 'SUBR. einfuegen: Liste nicht initialisiert'
       STOP
    END IF
    IF (ASSOCIATED (anker, nil)) THEN! Liste ist leer
       anker => neu
       NULLIFY (neu%nachfolger)
    ELSE IF (anker%info1 >= neu%info1 ) THEN
                              ! Neues Element wird 1. Element
       neu%nachfolger => anker
       anker => neu
```

Module in Fortran90

▲ SUBROUTINE einfuegen (anker, neu)

```
    ELSE                          ! Neues Element einsortieren
       z => anker

       DO WHILE (ASSOCIATED (z%nachfolger) .AND. &
                             z%nachfolger%info1 < neu%info1)
          z => z%nachfolger
       END DO
       neu%nachfolger => z%nachfolger
       z%nachfolger => neu
    END IF

END SUBROUTINE einfuegen
```


Module in Fortran90

▲ SUBROUTINE suche (anker, z, zv)

```
SUBROUTINE suche (anker, z, zv)
! Suche nach Element mit 'string';
! Formalparameter, anker = Eingabeparameter;
! z = Zeiger auf gesuchtes Element; zv Zeiger auf Vorgänger
! Formalparameter:
TYPE (NODE), POINTER :: anker, z, zv
! Lokale Variable
CHARACTER *10 string

    WRITE  (*, 1010, ADVANCE='NO')
1010FORMAT ('Suchstring: ')
    READ *, string
       z => anker; zv => anker
    DO WHILE (ASSOCIATED (z) .AND. z%info1 /= string)&
                                 zv => z; z => z%nachfolger
    END DO
    IF (.NOT. ASSOCIATED (z)) THEN
       PRINT *, 'Nicht gefunden'; RETURN
    END IF
    IF (debug) PRINT *, z%info1
END SUBROUTINE suche
```

Module in Fortran90

- SUBROUTINE vk_loesche (listid)

```
SUBROUTINE vk_loesche (listid)

! Loesche Element mit Inhalt 'string' in info1
! Formalparameter, listid = Eingabeparameter;
INTEGER, INTENT (IN) :: listid
! Lokale Variable
TYPE (NODE), POINTER :: z, zv

    CALL suche  (anker(listid)%ancor, z, zv)

    If (debug) PRINT *, z%info1
    If (debug) PRINT *, zv%info1

    IF (ASSOCIATED (zv, anker(listid)%ancor)) THEN
       anker(listid)%ancor => z%nachfolger
    ELSE
       zv%nachfolger => z%nachfolger
    END IF
    DEALLOCATE (z)
END SUBROUTINE vk_loesche
```


Module in Fortran90

- SUBROUTINE vk_write (listid)

```
SUBROUTINE vk_write (listid)
! Schreibe die Daten auf Datei "filelist(listid)"
! Formalparameter
INTEGER, INTENT(IN) :: listid
! Lokale Variable
TYPE (NODE), POINTER :: z
INTEGER i, statusvar
LOGICAL openstatus
    INQUIRE (UNIT=fdlist(listid), OPENED = openstatus, &
                                  IOSTAT = statusvar)
    IF (statusvar > 0) CALL exception ('INQUI', 'vk_writ',
statusvar)
    If ( .NOT. openstatus) THEN
    ! Oeffnen der Datei
       OPEN(UNIT=fdlist(listid), FILE = filelist(listid),&
          ACCESS = 'SEQUENTIAL', FORM ='UNFORMATTED',&
          POSITION = 'REWIND',  ACTION = 'WRITE',&
          STATUS = 'REPLACE', IOSTAT = statusvar)
       IF (statusvar > 0) CALL exception ('OPEN ', &
                                   'vk_writ', statusvar)
    ELSE
       REWIND (fd)
    END IF
```

Module in Fortran90

▲ SUBROUTINE vk_write (listid)

```
    i = 0
    z => anker(listid)%ancor
    DO WHILE (ASSOCIATED (z%nachfolger))
       WRITE (UNIT = fdlist(listid), IOSTAT = statusvar) &
          z%info1, z%info2, z%info3
       IF (statusvar > 0) CALL exception ('WRITE', &
                                      'vk_writ', statusvar)
       IF (debug) PRINT *, z%info1, z%info2, z%info3
       z => z%nachfolger
       i = i + 1
    END DO
    WRITE (UNIT=fdlist(listid), IOSTAT = statusvar) &
                                 z%info1, z%info2, z%info3
    IF (statusvar > 0) CALL exception ('WRITE', 'vk_writ',
statusvar)
    IF (debug) PRINT *, z%info1, z%info2, z%info3
    CLOSE (fdlist(listid))
2000PRINT *, i + 1, 'Elemente'

END SUBROUTINE vk_write
```


Module in Fortran90

▲ SUBROUTINE vk_liste (listid)

```
SUBROUTINE vk_liste (listid)
! Formalparameter
INTEGER, INTENT (IN) :: listid
! Lokale Variable

TYPE (NODE), POINTER :: z
    IF (debug) PRINT *, 'Listennummer = ',listid
    IF (.NOT. ASSOCIATED (anker(listid)%ancor)) THEN
       PRINT *, 'Verkettete Liste ist leer!'
       RETURN
    END IF
    z => anker(listid)%ancor
    DO WHILE (ASSOCIATED (z%nachfolger))
       PRINT *, z%info1, z%info2, z%info3
       z => z%nachfolger
    END DO
    PRINT *, z%info1, z%info2, z%info3

END SUBROUTINE vk_liste
```

Module in Fortran90

- SUBROUTINE vk_update (listid

```
SUBROUTINE vk_update (listid)

    INTEGER, INTENT (IN) :: listid

    PRINT *, 'NOT yet implemented', listid

END SUBROUTINE vk_update

! weitere Hilfsroutinen


END MODULE vk
```