Programmierung Paralleler Systeme

3. Programmiermodell “Single Programm Multiple Data”

3.1 Fortran95

3.1 .4 Selbstdef. Datentypen (Strukturen)

3.1.5 Zeiger(Pointer)

3.1.6 Unterprogramme

Claus-Uwe Linster

IMMD IV, Lehrstuhl für Betriebssysteme
Universität Erlangen-Nürnberg

linster@informatik.uni-erlangen.de

3.1.4 Selbstdefinierte Typen

▲ Typdefinition

```
TYPE [,attribut]... [::] typname
   komponentendefinition
   [komponentendefinition]
   .
END TYPE [typname]
```

Die Komponentendefinition entspricht weitgehend den Typvereinbarungsanweisungen; jedoch

- keine Initialisierung (es ist eine Definition; keine Deklaration!)
- als Attribute sind nur
  - DIMENSION (mit expliziter Gestalt) und
  - POINTER (Siehe Kapitel 1.5)

  zugelassen.

Selbstdefinierte Typen

Als Attribute der Typdefinition (dann gültig für alle Deklarationen dieses Typs) sind zulässig:

- PRIVATE (Siehe Kapitel 1.11)
- PUBLIC (Siehe Kapitel 1.11)
- SEQUENCE
  - erzwingt eine Speicherordnung gemäß der Komponentenreihenfolge; dies kann hinsichtlich der Speichernutzung Verschnitt bedeuten.
  - Verschiedene Typdefinitionen mit gleichen Komponenten (z. B. in verschiedenen Gültigkeitsbereichen)sind nur dann wirklich gleich, wenn sie das SEQUENCE-Attribut besitzen.
  - Wenn ein selbstdefinierter Typ das SEQUENCE-Attribut besitzt, dann müssen auch alle ggf. vorhandenen Typ-Komponenten, die ihrerseits selbstdefinierte Typen sind, das SEQUENCE-Attribut besitzen.
- Type - Vereinbarungsanweisung

```
Syntax:TYPE (TYPENAME) [,attribut]... [::] dg [,dg]...
```


Selbstdefinierte Typen

▲ Beispiele für Typdefinitionen

```
TYPE:: DATUM
  SEQUENCE
  INTEGER :: tag
  CHARCATER (LEN=3) :: monat
  INTEGER :: jahr
END TYPE DATUM
TYPE :: MITARBEITER
  CHARACTER *20 :: name, vorname
  TYPE (DATUM) geburtsdatum
  REAL gehalt
  REAL lohnsteuer
  LOGICAL verheiratet
  TYPE (MITARBEITER), POINTER :: alphanext ! (Siehe Kapitel 7)
END TYPE MITARBEITER
```

▲ Beispiel für Typvereinbarungen

```
TYPE (MITARBEITER), DIMENSION (100) :: gehaltsliste
TYPE (DATUM), DIMENSION (365) :: kalender
```

Selbstdefinierte Typen

- Strukturkonstante

```
Syntax: TYPNAME (komponentenliste)
  z. B. DATUM (1, 'Jan', 1996)
```

- Zuweisungen
  - Komponentenweise Zuweisung

```
variablenname%komponentenname = wert
```

  - Objektweise Zuweisung

```
variablenname = strukturkonstante
variablenname = variablenname     !gleichen Typs!
```


Selbstdefinierte Typen

- Beispiele

```
kalender(1)  = DATUM(1,'Jan',1996)
kalender(41)%tag = 10
kalender(41)%monat = 'Feb'
kalender(41)%Jahr = 1941

gehaltsliste(35) = MITARBEITER ('Mueller', 'Horst',&
     DATUM(10,'Feb',1941),4523.12,1212.00,.TRUE.,&
     gehaltsliste(25))

gehaltsliste(42)  = gehaltsliste(35)
gehaltsliste(42)%vorname = 'Anton'
gehaltsliste(42)%alphanext = gehaltsliste(35)
! Kapitel 1.5
```

Selbstdefinierte Typen

weitere Beispiele

- Typdefinition

```
TYPE :: ERSATZTEIL
  SEQUENCE
  CHARACTER *10 pkw_typ
  CHARACTER *10 part_nb
END ERSATZTEIL

TYPE :: PKW
  SEQUENCE
  CHARACTER *10 farbe
  INTEGER fahrgestellnummer
  INTEGER zulassungsjahr
  TYPE (ERSATZTEIL), DIMENSION (100) :: vorderachse
END TYPE PKW
```


Selbstdefinierte Typen

weitere Beispiele

- Typvereinbarungen

```
TYPE (PKW), DIMENSION (200) :: lager
```

- Zuweisungen

```
lager(110)%vorderachse(12)%part_nb = '1000G1019'
lager(1)%farbe = 'RAL312'
```

Selbstdefinierte Typen

```
PROGRAM personal
IMPLICIT NONE
INTEGER, PARAMETER :: max = 10
INTEGER, PARAMETER :: fd = 10
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
!  CHARACTER, DIMENSION (:), POINTER :: beurteilung! Kap. 1.5
END TYPE MITARBEITER

TYPE (MITARBEITER), DIMENSION (max) :: pa! Personal_akte
CHARACTER *4 funktion
CHARACTER zeichen
INTEGER i
LOGICAL :: ende = .FALSE.
```


Selbstdefinierte Typen

```
OPEN(fd,FORM='UNFORMATTED',file = 'database.personal')
   ! Oeffnen der Datei
DO WHILE (.NOT. ende)      !Funktionen

   PRINT *, ' '
   PRINT *, 'PRINT *, 'Eingabe der gewünschten Fkt:'; PRINT *,
' '
   PRINT *, 'init = Initialisiere Datenbasis'
   PRINT *, 'read = Lesen der Datenbasis'
   PRINT *, 'einz = Einzel-Eingabe'
   PRINT *, 'eing = Gruppeneingabe'
   PRINT *, 'korr = Korrektur;'
   PRINT *, 'ausg = Ausgabe;'
   PRINT *, 'sumg = Summe Gehaelter;'
   PRINT *, 'suml = Summe Lohnsteuer;'
   PRINT *, 'pers = Personalliste;'
   PRINT *, 'ende = Beenden des Programms'
   PRINT *, ' '
   PRINT *, 'Funktion?'
   READ *, funktion
   PRINT *, ' '
```

Selbstdefinierte Typen

```
SELECT CASE (funktion)
   CASE ('init')
      PRINT *,'Sind Sie sicher? (y)'
      READ *,zeichen
      if (zeichen /= 'y') EXIT
      DO i = 1,max
            pa(i)%name = 'frei'
            pa(i)%vorname = 'frei'
      END DO
   CASE ('read')
      READ(fd) (pa(i), i=1,max)
   CASE ('einz')
      DO i = 1,max
            IF (pa(i)%name == 'frei') THEN
                  Print *,'Name:'
                  READ *, pa(i)%name
                  PRINT *, 'Vorname'
                  READ * , pa(i)%vorname
                  EXIT
            ELSE
                  CYCLE
            END IF
      END DO
```


Selbstdefinierte Typen

```
   CASE ('eing')
      PRINT *, 'Not yet implemented'
   CASE ('korr')
      PRINT *, 'Not yet implemented'
   CASE ('ausg')
      PRINT *, 'Not yet implemented'
   CASE ('sumg')
      PRINT *, 'Not yet implemented'
   CASE ('suml')
      PRINT *, 'Not yet implemented'
   CASE ('pers')
      DO i= 1,max
            PRINT *, pa(i)%name, ';', pa(i)%vorname
      END DO
   CASE ('ende')
      PRINT *, 'Programmende'
      WRITE(fd) (pa(i), i = 1,max)
      ende = .TRUE.
   CASE DEFAULT
      PRINT *, 'Falsche Eingabe!'
   END SELECT
END DO
END PROGRAM personal
```

3.1.5 Zeiger (Pointer)

- Pointer Definition
  - Der Inhalt einer Pointervariablen ist die Adresse eines Objekts auf das der Pointer zeigt.
  - Konsequenterweise müßten Pointer - wie in anderen Sprachen (z. B. in C) - ein spezieller Datentyp sein. Da man in Fortran90 nicht direkt auf einen Pointer zugreifen kann (keine Adreßarithmetik!), handelt es sich nur um ein Attribut eines Datentyps.
  - In Fortran90 finden Pointer hauptsächlich Verwendung bei der dynamischen Speicherverwaltung (während der Laufzeit des Programms; detailiert in Kapitel 1.8 dargestellt) und beim Aufbau von Datenstrukturen wie verketteten Listen, Baumstrukturen, etc.

Zeiger (Pointer)

- Pointer-Vereinbarung

```
REAL, POINTER :: zeiger, next, current !Typ und Attribut
! oder
REAL zeiger, next, current                 !Typ
Pointer zeiger, next, current              !Attribut
```

  - Aus der Pointerdefinition folgt:
    - das Pointer-Attribut darf nur für Variablen oder Funktionen spezifiziert werden
    - folgende weitere Attribute sind unzulässig:
      ALLOCATABLE, EXTERNAL, INTENT, INTRINSIC, PARAMETER, TARGET
    - ein Pointer darf weder bei einer Typvereinbarung noch mit Hilfe einer DATA-Anweisung oder E/A initialisiert werden

Zeiger (Pointer)

Pointer-Zuweisung/Freigabe

- Will man einem Pointer eine Variable (oder Funktion) zuweisen, so muß für diese Datengröße das TARGET-Attribut spezifiziert sein.
  - Spezifikation des TARGET-Attributs

```
INTEGER, TARGET :: feld, person       ! Typ und Attribut
! oder
INTEGER feld, person    ! Typ
TARGET feld, person     ! Attribut
```

  - Zuweisung

```
pointer => zielvariable
! oder
ALLOCATE (pointer [,pointer]...) [, STAT=statusvariable]
```

  - Freigabe von Pointerzielen

```
pointer => neuezielvariable          !Überschreiben
! oder
DEALLOCATE (pointer [,pointer]...) [, STAT=statusvariable]
```


Zeiger (Pointer)

Pointer-Zuordnungsstati

- zugeordnet:
  - dem Pointer wurde eine Variable mit dem TARGET-Attribut zugewiesen.
  - dem Pointer wurde ein Pointer zugewiesen, der selbst den Status “zugeordnet” besitzt.
  - dem Pointer wurde mit der ALLOCATE-Anweisung ein Speicherbereich zugeordnet
- nicht zugeordnet:
  - dem Pointer wird mit der NULLIFY-Anweisung eine Zuordnung entzogen (Keine damit verbundene Speicherfreigabe!).
  - NULLIFY (pointer [,pointer]...)
  - dem Pointer wird mit der DEALLOCATE-Anweisung ein Speicherbereich entzogen, der ihm mit einer ALLOCATE-Anweisung zugeordnet wurde.
  - dem Pointer wird ein Pointer mit dem Status “nicht zugeordnet” zugewiesen.

Zeiger (Pointer)

Pointer-Zuordnungsstati(cont)

- undefiniert:
  - dem Pointer wurde noch kein Ziel zugewiesen (weder '=>' noch **`ALLOCATE`**)
  - das zugeordnete Ziel wurde auf "anderer Weise" entfernt;
    z. B. ein **`DEALLOCATE`** auf einen Zeiger mit dem gleichen Ziel
  - dem Pointer wird ein Pointer zugewiesen mit dem Staus "undefiniert"
  - wenn das Pointerziel bei Ausführung einer **`END`**- oder **`RETURN`**-Anweisung undefiniert wird. (Kapitel 1.6)

Zeiger (Pointer)

ASSOCIATED-Function

- Intrinsic (vordefinierte) Funktion

  `Syntax: ASSOCIATED (pointervar [,targetvar]`

  - liefert den Wert **`.TRUE.`** beim Aufruf

    `ASSOCIATED (pointervar)`

  wenn **`pointervar`** irgendeinem Ziel zugeordnet ist und beim Aufruf

    `ASSOCIATED (pointervar, targetvar)`

  wenn **`pointervar targetvar`** zugeordnet ist.

  - sonst den Wert **`.FALSE.`**

Beispiele

- zu Datenstrukturen, verkettete Listen, Baumstrukturen, etc
  siehe auch Vorl. mit Übungen : "Programm- und Datenstrukturen"
- zu **`ALLOCATE, DEALLOCATE, NULLIFY`**
  siehe auch Feldverarbeitung (Dynamische Speicherverwaltung) Kap. 1.8

Zeiger (Pointer)

```
PROGRAM queue

! Warteschlangen-Demo-Programm adaptiert aus:
! Cooper Redwine; "Upgrading to FORTRAN 90", Springer-Verlag,
! 1995
!
! Queue - Policy: first in - first out (Hier zunaechst nur
! Eingabe)

IMPLICIT NONE
TYPE :: QUEUE_ELEMENT
   CHARACTER (20) :: data
   TYPE (QUEUE_ELEMENT), POINTER :: next
END TYPE

! Zeiger auf des erste, letzte, aktuelle Element: first, last,
! this

TYPE (QUEUE_ELEMENT), POINTER :: first, last, this

CHARACTER(20) :: buffer
CHARACTER *4 :: funktion
LOGICAL :: ende = .FALSE.
INTEGER :: io_stat = 1
```


Zeiger (Pointer)


```
   DO WHILE (.NOT. ende)         !Funktionen
   PRINT *, ' '
   PRINT *, 'Eingabe der gewünschten Fkt:'
   PRINT *, ' '
   PRINT *, 'init = Initialisiere Queue'
   PRINT *, 'read = Lese Queue'
   PRINT *, 'erze = Erzeuge Elemente'
   PRINT *, 'dele = Loesche 1. Element'
   PRINT *, 'ende = Beenden des Programms'
   PRINT *, ' '
   PRINT *, 'Funktion:'
   READ *, funktion
   PRINT *, ' '
```

Zeiger (Pointer)

```
SELECT CASE (funktion)
   CASE ('init')
      NULLIFY (first, last)! Initialisiert: leere Queue
   CASE ('read')

      ! Durchlaufe die Warteschlange und drucke Inhalte als
      ! Liste aus.

      this => first           ! aktuelles Element gleich
                              ! 1. Element
      PRINT *, 'Ausgabe'; PRINT *, ' '
      DO WHILE (ASSOCIATED (this))
            PRINT *, this%data
            this => this%next ! aktuelles Element gleich
                              ! naechstes Element
      END DO
```


Zeiger (Pointer)

```
CASE ('erze')
   PRINT *, 'Eingabe:'
   DO WHILE (io_stat >= 0)
      READ (*, "(A)", IOSTAT = io_stat) buffer

      IF (.NOT. ASSOCIATED (first)) THEN

      ! Queue ist leer; fordere den Speicher fuer das
      ! 1. Element an

            ALLOCATE (first)
            last => first     ! Beide Zeiger zeigen aufs
                              ! gleiche(erste)Element
      ELSE
      ! Queue ist nicht leer, neues Element gleich
      ! letztes Element

            ALLOCATE (last%next)
            last => last%next
      END IF
      last%data = buffer
      NULLIFY (last%next)     ! Letztes Element hat
                              ! keinen Nachfolger
   END DO
```

Zeiger (Pointer)

```
          CASE ('dele')
             PRINT *, 'Not yet implemented'
          CASE ('ende')
             PRINT *, 'Programmende'
             ende = .TRUE.
          CASE DEFAULT
             PRINT *, 'Falsche Eingabe!'
          END SELECT
END DO

END PROGRAM queue
```


3.1.6 Unterprogramme

- Definitionen/Terminologie
  - Programmeinheiten (programunits) sind die Bausteine eines ausführbaren Programms
    - Hauptprogramm (genau einmal)
    - ext. Unterprogramme
    - Module
    - Blockdatenprogrammeinheiten
  - Sie enden mit der END-Anweisung, liegen in separaten Dateien, sind getrennt übersetzbar, jedoch bis auf das Hauptprogramm nicht direkt ausführbar.
  - Ein Hauptprogramm ist ausführbar, wenn es keine offenen ’Externreferenzen’ gibt - ausgenommen Bibliotheksreferenzen, wie z.b. SIN(x), COS(x), etc.
    Diese Eigenschaft haben unsere bisher geschriebenen Programme.

Unterprogramme

- Die weiteren Programmeinheiten sind nur dann ausführbar (getrennt testbar), wenn ein entsprechendes Hauptprogramm als “Testrahmen” zugebunden wird.
- Nach der Übersetzung sind diese Einheiten zunächst separate Bindemodule; sogenannte ".o-files"

- Unterprogramme werden unterschieden
  - nach ihrem “Definitionsort”
    - extern
    - eingebettet
    - vordefiniert in Bibliothek
  - und ihrer Art
    - Subroutine
    - Funktion
    - Formelfunktion

Unterprogramme

- Subroutine
  - Subroutinendefinition

```
[...] SUBROUTINE sname [(...)]        !(SUBROUTINE-Anweisung)
    [Spezifikationsteil]
    ...
        Subroutinenrumpf
    ...
[CONTAINS]
    [interne Unterprogramme]
    ...
END [SUBROUTINE [sname]]
```

  - Beispiele:

```
SUBROUTINE sname()                    !(leere Parameterliste)
SUBROUTINE sname (formalparameter[,formalparameter] ...)
```

  - Subroutinenaufruf

```
CALL sname
CALL sname()
CALL sname(Aktualparameter[,Aktualparameter]...)
```

Unterprogramme

- Beispiel - eingebettetes Unterprogramm (CONTAINS)

```
PROGRAM summentest
IMPLICIT NONE
REAL a, b, ergebnis

   Print *, 'Eingabe: summand1 summand2 <r>'
   READ *, a,b

   call summe (a,b,ergebnis)

   PRINT *, 'Ergebnis = ',ergebnis

CONTAINS

      SUBROUTINE summe (summanda, summandb, erg)
      REAL summanda, summandb, erg
         erg = summanda + summandb
      END SUBROUTINE summe

END PROGRAM summentest
```


Unterprogramme

- Geltungsbereich/Gültigkeitsbereich
  - sagt aus, in welchen Bereichen eine Variable zugreifbar ist
    - lokal Variable sind nur gültig innerhalb eines Geltungsbereichs z. B. in einem Unterprogramm
    - globale Variable sind in der gesamten Programmeinheit, in der sie definiert wurden, zugreifbar, falls sie nicht durch eine lokale Definition “verdeckt” werden.
      (“Blickt” man aus der Aufrufschachtelung zurück in Richtung Aufrufer, so sind sämtliche globale Daten der aufrufenden Ebenen zugreifbar, es sei denn, sie sind durch lokale Vereinbarungen verdeckt.)

Unterprogramme

- Geltungsbereich/Gültigkeitsbereich(cont)
  - Transfer von Informationen zwischen Geltungsbereichen durch:
    - Parameter
      - Eingangsparameter
      - Ausgangsparameter; Ergebnisparameter
      - beide Richtungen
    - globale Daten
      - im äußersten Geltungsbereich definiert; also programmeinheitsglobal
      - COMMON-Bereiche
        Überall dort gültig, wo definiert; auch über Programmeinheiten hinweg

Unterprogramme

- Parameter
  - Formalparameter (bei der Definition)
  - Aktualparameter (beim Aufruf)

    Zum Zeitpunkt des Aufrufs werden die Formalparameter und die Aktualparameter einander zugeordnet, d.h. eine “Formal-Variable” im Unterprogramm referenziert den gleichen Speicher wie die “Aktual-Variable” im aufrufenden Programm.

  Man unterscheidet

    - positionsgerechte Schreibweise
    - Schlüsselwortschreibweise z. B.

      formalparameter = aktualparameter

    Die gemischte Schreibweise ist möglich. Beginnt man jedoch mit der Schlüsselwortschreibweise, so müssen auch alle folgenden Parameter in Schlüsselwortschreibweise folgen.

Unterprogramme

INTENT-Attribut/Anweisung

- Mit dem INTENT-Attribut kann die Art der Verwendung von Formalparametern spezifiziert werden:
  - Das INTENT(IN)-Attribut besagt, daß der Formalparameter ein Eingabeparameter ist. D.h. dieser Formalparameter darf innerhalb des Unterprogramms nicht redefiniert werden, er darf also z. B. nicht auf der linken Seite einer Zuweisung auftauchen oder Variable einer READ-Anweisung sein.
    Der Aktualparameter kann also sowohl eine Variable als auch eine Konstante sein.
  - Das INTENT(OUT)-Attribut besagt, daß der Formalparameter ein Ausgabeparameter ist. D. h. der Formalparameter muß erst definiert (ein Wert zugewiesen) werden, ehe im Unterprogramm auf ihn zugegriffen wird. Hier **muß** es sich also um eine Variable handeln.

Unterprogramme

INTENT-Attribut/Anweisung

- Mit dem INTENT-Attribut kann die Art der Verwendung von Formalparametern spezifiziert werden (cont):
  - Das INTENT(INOUT)-Attribut besagt, daß der Parameter sowohl Eingabe- als auch Ausgabeparameter sein soll. Auch hier **muß** es sich also um eine Variable handeln.

Wir werden im Rahmen dieses Kurses die Formalparameter mit INTENT-Attributen versehen denn: Keine Angabe = INTENT(INOUT), es sei denn, der Aktual-Parameter ist eine Konstante.

Unterprogramme

- Beispiel:

```
PROGRAM summentest
IMPLICIT NONE
REAL a, b, ergebnis

   Print *, 'Eingabe: summand1 summand2 <r>'
   READ *, a,b

   call summe (a,b,ergebnis)

   PRINT *, 'Ergebnis = ',ergebnis

CONTAINS

      SUBROUTINE summe (summanda, summandb, erg)
      REAL, INTENT(IN) :: summanda, summandb
      REAL, INTENT(out) :: erg
         erg = summanda + summandb
      END SUBROUTINE summe

END PROGRAM summentest
```


Unterprogramme

- FUNKTION

```
[...] FUNCTION fname [(...)]!(FUNCTION-Anweisung)
    [Spezifikationsteil]
    ...
        Funktionsrumpf
        ...
[CONTAINS]
    [interne Unterprogramme]
    ...
END [FUNCTION [fname]]
```

  - der wesentliche Unterschied einer Funktion zu einer Subroutine ist, daß der Funktionsname gleichzeitig auch der Name der (lokalen) Ergebnisvariablen ist. Deshalb hat die Funktion einen Typ!
  - Der Aufruf einer Funktion kann überall dort stehen, wo auch eine Variable stehen darf; nur auf der linken Seite einer Zuweisung (natürlich) nicht.
  - Der Funktionsaufruf darf auch ein separates Statement sein - analog zur Subroutine - allerdings ohne “CALL”; jedoch mit “()”.

Unterprogramme

```
( ~uelinste/tmp/newton_fkt.f90 )

PROGRAM newton
REAL eingabe
! Author: Uwe Linster
! Anfordern und einlesen der Zahl, deren Quadratwurzel
! berechnet werden soll

   PRINT *, ' '

   ! Ausgabe auf das Sichtgerät ohne Zeilenvorschub
   ! Nichtvorrueckende Ausgabe: Kapitel 1.10

   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Zu berechnen ist die Quadratwurzel aus: ')

   READ *, eingabe
   PRINT *, ' '; PRINT *, ' '

   DO WHILE(eingabe <= 0)
      PRINT *, ' Die Eingabedaten entsprechen nicht den &
               &Anforderungen des Programms (Zahl > 0!):'
      READ *, eingabe
   END DO
```


Unterprogramme

```
!Berechnung

   ergebnis = sqroot(eingabe)

!Ergebnisausgabe

   PRINT *, ' Die Quadratwurzel aus ', eingabe ;PRINT *, ' '
   PRINT *, ' ist ', ergebnis
! Oder auch gleich:
!  PRINT *, ' ist ', sqroot(eingabe)

PRINT *, ' '; PRINT *, ' Ende des Programms'
```

Unterprogramme

```
CONTAINS

REAL FUNCTION sqroot (a)
!Berechnung der Quadratwurzel nach dem Newton'schen Verfahren
!Eingabeparameter:a; Startwert der Iteration: 1
!Iterationsformel:x(i+1) = (x(i)*x(i) + a) / (2*x(i))

   REAL , INTENT(IN) :: a
   REAL x
      !a Größe, deren Quadratwurzel zu berechnen ist
      !x Berechnete Approximation der Quadratwurzel

   !Anfangsapproximation
   x = 1
   !Iteration
   DO WHILE (.NOT. (ABS(x*x - a) <= 1E-4))
      x = (x*x + a) / (2.*x)
   END DO
   sqroot = x
END FUNCTION sqroot

END PROGRAM newton
```


Unterprogramme

- COMMON-BEREICHE
  - Durch COMMON-Bereiche werden globale Speicherbereiche spezifiziert. Variable sind in den Geltungsbereichen direkt zugreifbar, in denen die COMMON-Bereiche spezifiziert wurden; auch über verschiedene Programmeinheiten hinweg.
    Man unterscheidet:
    - einen unbenannten COMMON-Block

```
Syntax:COMMON variablenliste
```

    - und beliebig viele benannte COMMON-Blöcke

```
Syntax:COMMON /blockname/ variablenliste [, /blockname/]
              variablenliste]...
```

    - auch die Mischschreibweise ist möglich:

```
Syntax:COMMON  variablenliste [[,]/[blockname]/
              variablenliste]...
```

Unterprogramme

- COMMON-BEREICHE(cont)
  - Eigenschaften
    - Durch Angabe der Variablen wird eine Speicherfolge festgelegt; um Fehler zu vermeiden sollte man mit “copy” und “paste” arbeiten.
    - Variablennamen in verschiedenen Geltungseinheiten dürfen verschieden sein; das sollte man jedoch nur tun, wenn es dafür auch einen plausiblen Grund gibt. Man greift auf dieselbe globale Speicheradresse zu, das sollte man durch den gleiche Variablennamen auch dokumentieren!
    - Initialisiert werden COMMON-Bereiche mit einer BLOCK DATA-Programmeinheit; jedoch nur die benannten Blöcke.
    - Selbstdefinierte Datentypen müssen das Attribut SEQUENCE besitzen. (Speicherfolge!)

Unterprogramme

- COMMON-Bereich - Beispiel (~uelinste/tmp/common_demo_bubble.f90

```
PROGRAM bubble
IMPLICIT NONE

! Zweck des Programms, Loesungsverfahren:
!
! Die Eintraege in der Datei titel sollen in lexikographisch
! sortierter Reihenfolge gedruckt werden. Das Programm geht
! davon aus, dass die Datei maximal 200 Eintraege besitzt und
! jeder Eintrag aus hoechstens 20 Zeichen besteht.
! Als Sortierverfahren wird 'bubble sort' verwendet.
!-------------------------------------------------------------
! Globale Variable; in COMMON-Bereiche spezifiziert
!    name Feld zur Aufnahme der Eintraege aus der Dateri titel
!    max Anzahl der Eintraege
!  fd Kanalnummer (Filedescriptor; Unit)
!-------------------------------------------------------------
! Initialisierung
!
! Ausgabe der Gesamtueberschrift
!
PRINT *, 'Eintraege der Datei titel lexikograpgisch sortiert:'
```

Unterprogramme

```
! Oeffnen der Datei
!
   CALL oeffnen_datei()

! Einlesen der Daten
!
   CALL einlesen()
!
! Sortieren
!
   CALL bubblesort()
!
! Ausgabe des Ergebnisses
!
   CALL ausgeben_screen()
   CALL ausgeben_datei()
```


Unterprogramme

```
! Die Unterprogramme:

CONTAINS

SUBROUTINE oeffnen_datei()

! Oeffnen der Datei "titel" mit Kanalnummer 10
! Global Data:

INTEGER fd
COMMON /datei/ fd
   fd = 10     ! Entfaelt bei BLOCK-DATA-INIT!!
! Local DATA: none

   open(fd, FORM = 'FORMATTED', file = 'titel')

END SUBROUTINE oeffnen_datei
```

Unterprogramme

```
SUBROUTINE einlesen()

! Einlesen der Eintraege aus der Datei "titel" in das Feld !
"name".
! Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

!Local Data:
INTEGER i

   DO i = 1, 200
      READ(fd, 1000, END = 1010) name(i)
1000  FORMAT (a)
   END DO

1010max = i - 1
   PRINT *, 'Gelesene Datensaetze: ',max

END SUBROUTINE einlesen
```


Unterprogramme

```
SUBROUTINE bubblesort()
! Sortiert nach dem "bubble-sort"-Verfahren
! Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

!Local Data

CHARACTER *20 :: hilf
LOGICAL flag
INTEGER i, j, dl

!  flag   zeigt an, ob bei einem Durchlauf Aenderungen in der
!              Anordnung vorgenommen wurden
!              (.FALSE.: Keine Umordnungen
!               .TRUE. : Es wurden Umordnungen vorgenommen)
!  hilf   Hilfsfeld waehrend des Vertauschens zweier Eintraege
!  dl Anzahl der Sortierschleifen
!
```

© Claus-Uwe Linster – pps3.1-2_spmd_f90: 19/3/98

Unterprogramme

```
! Der Sortiervorgang

   dl = 0
   DO  i = max, 2, -1
      flag = .FALSE.

      DO j = 1, i - 1
          IF (name(j) > name(j + 1)) THEN
               hilf = name(j)
               name(j) = name(j + 1)
               name(j + 1) = hilf
               flag = .TRUE.
               dl = dl + 1
          END IF
        END DO

        if (.NOT. flag) THEN
         PRINT *, 'Anzahl der Durchlauefe: ', dl
         EXIT
      END IF

   END DO

END SUBROUTINE bubblesort
```


Unterprogramme

```
SUBROUTINE ausgeben_screen()

!Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max
COMMON name, max

! Local Data:

INTEGER i

   DO i=1,max
      Print *, name(i)
   END DO

END SUBROUTINE ausgeben_screen
```

Unterprogramme

```
SUBROUTINE ausgeben_datei()

!Global Data

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

! Local Data

INTEGER i

   REWIND (fd)
   WRITE ( fd, 1030 )(name(i), i=1, max)
1030FORMAT (a)

END SUBROUTINE ausgeben_datei

END PROGRAM bubble
```


Unterprogramme

- BLOCK DATA
  - Eine BLOCK-DATA-Programmeinheit
    - ist eine nichtausführbare Programmeinheit, sie dient der Initialisierung von Variablen in **benannten** COMMON-Bereichen.
    - COMMON-Blöcke sind globale Speicherbereiche - gültig auch über Programm-Unit-Grenzen hinweg. Eine Initialisierung darf also Programmsystemweit nur einmal erfolgen. Dies könnte innerhalb einer “ausgezeichneten” Programmeinheit erfolgen oder wie hier in einer speziellen Programmeinheit

```
BLOCKDATA [name]
     [Spezifikationsteil]
     .
     .
     .
  END BLOCKDATA [name]
```

Unterprogramme

- BLOCK DATA(cont)

```
Beispiel
BLOCKDATA car

CHARACTER vorderachse *14, hinterachse *10, motor *11
INTEGER anzahl, seriennummer, partnb

COMMON /ersatzteile/ vorderachse, hinterachse, motor
COMMON /attribute/ anzahl, seriennummer, partnb

DATA vorderachse, hinterachse /'Schwingachsen', `Inkaachsen'/
DATA motor /'Dieselmotor'/
DATA anzahl, partnb /250, 4711/ ! seriennummer bleibt
uninitialisiert

END BLOCKDATA car
```

- Da diese Programm-Unit in einer eigenen Datei liegt muß beim Aufruf des Compilers diese Datei als weiterer Parameter angegeben werden:; z. B.

```
f90 hauptprogramm.f90 blockdata.f90 -o programm.out
```


Unterprogramme

- BLOCK DATA
  - Beispiel für Programm “bubble”:

Datei: bubble_blockdata.f90

```
BLOCKDATA bubble

   INTEGER fd

   COMMON /datei/ fd

   DATA fd /10/

END BLOCKDATA bubble
```

- Aufruf des Compilers:

```
f90 common_demo_bubble.f90 bubble_blockdata.f90 -o bubble.out
```

Unterprogramme

FORMELFUNKTION

- eine Formelfunktion ist eine Funktion, die nur aus einer einzigen Anweisung besteht und
- sie ist nur innerhalb einer Geltungseinheit aufrufbar.
- Formelfunktionsdefinition:

```
Syntax:ffname() = ausdruck oder
ffname(formalparameter[,formalparameter]...) =
                                  ausdruck
```

- Sowohl ’ausdruck’ als auch ’formalparameter’ sind skalare Datenobjekte.
- Formelfunktionsaufruf:

```
Syntax:ffname() oder
ffname(formalparameter[,formalparameter]...)
```

- Der Typ der Formelfunktion wird separat (wie bei der Typvereinbarung einer Variablen) vereinbart; dieses gilt auch für die Formalparameter.

Unterprogramme

- Formelfunktion - Beispiel (~uelinste/tmp/capslock_fkt.f90)

```
PROGRAM capslock
IMPLICIT NONE
!
! Das Programm liest Kleinbuchstaben ein, wandelt sie
! in Grossbuchstaben um und druckt diese aus.
!  Variable
!
      CHARACTER, DIMENSION(20) :: zeichenin, zeichenout
      INTEGER i

   CHARACTER gross, klein

!  Formelfunktion:

   gross(klein) = CHAR ( ICHAR(klein) + ICHAR('A') - ICHAR('a'))
```

Unterprogramme

```
!  Einlesen der Zeichen

      PRINT *, ' '
   PRINT *, 'Kleinbuchstaben; max. 20; entweder durch "blank"
getrennt -'
   PRINT *, 'Abschluss <r> oder einzeln durch <r> getrennt!'
   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Eingabe: ')
      READ *, (zeichenin(i), i=1,20)

   DO i = 1,20
      zeichenout(i) = gross(zeichenin(i))
   END DO

   PRINT *, 'Ausgabe:',zeichenout

END PROGRAM capslock
```


Unterprogramme

- Formelfunktion - Beispiel (~uelinste/tmp/kredit_fkt.f90)

```
PROGRAM kredit
IMPLICIT NONE
!
!  Das Programm berechnet die Hoehe der monatlichen Raten fuer
! einen vorgebbaren Kaufpreis bei einzugebender Laufzeit. Raten
!  und Zinsen werden von zwei Jahren bis zu einer angebbaren
!  maximalen Laufzeit in Jahresschritten berechnet. Untere und
!  oebere Grenze des Zinssatzes werden ebenfalls eingelesen.
! ------------------------------------------------------------
!  Variable
!
      REAL preis, mrate, p
      INTEGER n, zmin, zmax, zsatz, zs, year, yearmax
!
! Berechnung der Monatsrate fuer die verschiedenen Zinssaetze
!          formel: r*p/(1-(1+r)**(-n)) mit r = zinssatz/12/100
!  hier als Formelfunktion realisiert:

   mrate(zs,p,n) = ((zs/100./12.) * p) / &
               (1.-(1. + (zs/100./12.))**(-n))
```

Unterprogramme

```
!        preis      Kaufpreis (Eingabe)
!       yearmax    maximale Laufzeit der Ratenzahlung (Eingabe)
!        zmin       minimaler Zinssatz (Eingabe)
!        zmax       maximaler Zinssatz (Eingabe)
!        year       Laufvariable fuer die Laufzeit
!        n          Zahl der Raten
!        zsatz      Laufvariable fuer den Zinssatz
! ------------------------------------------------------------
! Initialisierung
!
!  Einlesen von Kaufpreis, maximaler Laufzeit,
!  minimalem und maximalem Zinssatz
!
      PRINT *, ' '
   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Kaufpreis (REAL): ')
      READ *, preis
   WRITE (*, 2000, ADVANCE='NO')
2000FORMAT ('Laufzeit in Jahren (groesser als 2): ')
      READ *, yearmax
     WRITE (*, 3000, ADVANCE='NO')
3000FORMAT ('minimaler und maximaler Zins (in Prozent
(INTEGER)): ')
      READ *, zmin, zmax; PRINT *, ' '
```


Unterprogramme

```
! Aeussere Schleife

    DO year = 2, yearmax

!  Tabellenueberschrift
        PRINT *, ' '
        PRINT *, ' '
      PRINT *, ' Zinsen fuer eine Laufzeit von ', year, ' Jahren'
        PRINT *, ' '
       PRINT *, ' Zins-  Zahl der    Rate         Gesamtzins'
        PRINT *, ' satz    Raten       (DM)             (DM)'

! Innere Schleife
      DO zsatz = zmin, zmax
         n = 12.*year
          PRINT *, ' ', zsatz, '     ', n, ' ', &
               mrate (zsatz, preis, n) &
               ,' ', n * (mrate (zsatz, preis, n)) -preis
      END DO
    END DO
!
!        Abschlussausgabe
      PRINT *, ' ' ;PRINT *, ' Berechnung abgeschlossen.'
END PROGRAM kredit
```

Unterprogramme

Rücksprung

- Es existieren drei alternative Methoden
  - der “normale” Rücksprung bei Erreichung des END-Statements
  - der “Ausnahme” - Rücksprung, wenn aufgrund einer Bedingung die Aufgabe des Unterprogramms erfüllt ist - etwa analog zum EXIT-Aussprung aus einer DO-Schleife

```
Syntax:RETURN
```

Bei beiden Methoden wird mit dem 1. ausführbaren Statement fortgefahren, das dem Unterprogrammaufruf folgt.

  - der “Ausnahme” - Rücksprung mit wählbarer Fortsetzungsadresse (label).

```
Syntax:RETURN ganzzahliger_ausdruck
```

  - Diese Möglichkeit wird nur der Vollständigkeit erwähnt (Im Falle, daß man mit vorhandenen Programmsystemen umgehen muß). Die Verwendung solcher Konstrukte führt zu undurchsichtigen, nicht mehr wartbaren Programmsystemen.

Unterprogramme

SAVE-Anweisung

- Mit dem SAVE-Attribut wird der Wert einer **lokalen** Variablen in einem Geltungsbereich über die “Lebensdauer” eines Aufrufs hinaus “gesichert”. D. h., daß der Wert der Variablen auch beim nächsten Aufruf wieder verfügbar ist. Man spricht auch von “*Static”* - Variablen.
- Das SAVE-Attribut ist nur erlaubt bei
  - lokalen Variablen
  - benannten COMMON-Blöcken
    (hier ist Vorsicht geboten, ist doch ein COMMON-Block ein globaler Speicherbereich!?!)
- Das SAVE-Attribut ist insbesondere nicht erlaubt bei
  - einzelnen Variablen, die in einem COMMON-Bereich enthalten sind
  - Formalparametern
  - Funktionsvariablen
  - Namensgruppen (Siehe Kapitel 1.10)

Unterprogramme

SAVE-Anweisung

Attributspezifikation

- gleichzeitig mit der Typvereinbarung

```
REAL, SAVE :: skonto, summe
INTEGER, SAVE :: max
```

- oder getrennt mit einer SAVE-Anweisung

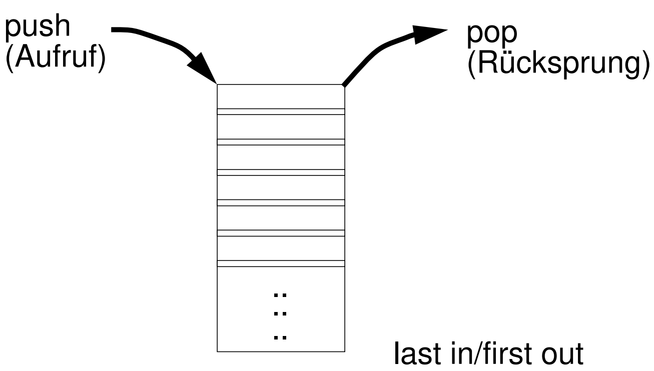
```
REAL skonto, summe
SAVE skonto, summe

INTEGER max
SAVE max
```

Eine SAVE-Anweisung ohne Angabe von Variablennamen bedeutet, das sämtliche lokale Variable in diesem Geltungsbereich diese Spezifikation erhalten.

Rekursive Unterprogramme

Unterprogramme (SUBROUTINE oder FUNCTION) sind dann rekursiv, wenn sie sich selbst direkt oder indirekt aufrufen.

Wenn Unterprogramme das Attribut RECURSION besitzen, muß der Compiler dafür Sorge tragen, daß jeweils bei einem rekursiven Aufruf die lokalen Variablen gerettet und nach dem Rücksprung aus einer tieferen Aufrufebene wieder bereitgestellt werden. Dies ist eine typische Anwendung für einen Kellerspeicher (Stack)!



Rekursive Unterprogramme

- Wann verwendet man rekursive Unterprogramme?
  - Es gibt eine Anzahl von Algorithmen, die "in sich" rekursiv definiert sind. Bei der iterativen Implementierung solcher Algorithmen ist der Programmieraufwand in der Regel beträchtlich und mit einer großen Anzahl von Abfragen behaftet, wie einige Beispiele des Kapitels noch zeigen werden.
  - Als einfaches Beispiel - zum Verständnis - wurde die Berechnung von Fakultäten gewählt, tatsächlich ist es in diesem Fall allerdings effizienter, eine Schleife zu implementieren!

```
PROGRAM recursion
IMPLICIT NONE
INTEGER n, erg
   WRITE (*, FMT = "('n! ; Welches n? : ')", ADVANCE = 'NO')
   READ *, n
   CALL fak (n, erg)
   PRINT *, 'Fakultaet von: ',n ,' = ', erg
   erg = fakfunc (n)
   PRINT *, 'Fakultaet von: ',n ,' = ', erg
CONTAINS
```


Rekursive Unterprogramme

- Rekursive Subroutine

```
RECURSIVE SUBROUTINE fak (fn, ferg)

INTEGER, INTENT (IN) :: fn
INTEGER, INTENT (OUT) :: ferg

   IF (fn == 0) THEN
         ferg = 1
   ELSE
         CALL fak (fn - 1, ferg)
         IF (ferg < 0) THEN
               PRINT *, 'underflow'
               STOP
         END IF
         ferg = fn * ferg
   END IF

END SUBROUTINE fak
```

Rekursive Unterprogramme

▲ Rekursive Funktion

```
RECURSIVE INTEGER FUNCTION fakfunc (fn) RESULT (ferg)

INTEGER, INTENT (IN) :: fn

   IF (fn == 0) THEN
         ferg = 1
   ELSE
         ferg = fn * fakfunc (fn - 1)
         IF (ferg < 0) THEN
               PRINT *, 'underflow'
               STOP
         END IF
   END IF

END FUNCTION fakfunc

END PROGRAM recursion
```


Unterprogramme

■ (Noch) nicht behandelt

- Parameterübergabe bei nicht skalaren Datengrößen (Siehe Kapitel 1.7- Feldverarbeitung)
- Module (Siehe Kapitel 1.11 - Module)
- Externe Unterprogramme (Siehe Kapitel n.b.)
- Nebeneinganspunkte - ENTRY-Anweisungen (Siehe Kapitel n.b.)
- Optionale Formalparameter (Siehe Kapitel n.b.)

Programmierung Paralleler Systeme

3. Programmiermodell “Single Programm Multiple Data”

3.1 Fortran95

3.1 .4 Selbstdef. Datentypen (Strukturen)

3.1.5 Zeiger(Pointer)

3.1.6 Unterprogramme

Claus-Uwe Linster

IMMD IV, Lehrstuhl für Betriebssysteme
Universität Erlangen-Nürnberg

linster@informatik.uni-erlangen.de

3.1.4 Selbstdefinierte Typen

▲ Typdefinition

```
TYPE [,attribut]... [::] typname
   komponentendefinition
   [komponentendefinition]
   .
END TYPE [typname]
```

Die Komponentendefinition entspricht weitgehend den Typvereinbarungsanweisungen; jedoch

- keine Initialisierung (es ist eine Definition; keine Deklaration!)
- als Attribute sind nur
  - DIMENSION (mit expliziter Gestalt) und
  - POINTER (Siehe Kapitel 1.5)

  zugelassen.

Selbstdefinierte Typen

Als Attribute der Typdefinition (dann gültig für alle Deklarationen dieses Typs) sind zulässig:

- PRIVATE (Siehe Kapitel 1.11)
- PUBLIC (Siehe Kapitel 1.11)
- SEQUENCE
  - erzwingt eine Speicherordnung gemäß der Komponentenreihenfolge; dies kann hinsichtlich der Speichernutzung Verschnitt bedeuten.
  - Verschiedene Typdefinitionen mit gleichen Komponenten (z. B. in verschiedenen Gültigkeitsbereichen)sind nur dann wirklich gleich, wenn sie das SEQUENCE-Attribut besitzen.
  - Wenn ein selbstdefinierter Typ das SEQUENCE-Attribut besitzt, dann müssen auch alle ggf. vorhandenen Typ-Komponenten, die ihrerseits selbstdefinierte Typen sind, das SEQUENCE-Attribut besitzen.
- Type - Vereinbarungsanweisung

```
Syntax:TYPE (TYPENAME) [,attribut]... [::] dg [,dg]...
```


Selbstdefinierte Typen

▲ Beispiele für Typdefinitionen

```
TYPE:: DATUM
  SEQUENCE
  INTEGER :: tag
  CHARCATER (LEN=3) :: monat
  INTEGER :: jahr
END TYPE DATUM
TYPE :: MITARBEITER
  CHARACTER *20 :: name, vorname
  TYPE (DATUM) geburtsdatum
  REAL gehalt
  REAL lohnsteuer
  LOGICAL verheiratet
  TYPE (MITARBEITER), POINTER :: alphanext ! (Siehe Kapitel 7)
END TYPE MITARBEITER
```

▲ Beispiel für Typvereinbarungen

```
TYPE (MITARBEITER), DIMENSION (100) :: gehaltsliste
TYPE (DATUM), DIMENSION (365) :: kalender
```

Selbstdefinierte Typen

▲ Strukturkonstante

```
Syntax: TYPNAME (komponentenliste)
  z. B. DATUM (1, 'Jan', 1996)
```

▲ Zuweisungen

- Komponentenweise Zuweisung

```
variablenname%komponentenname = wert
```

- Objektweise Zuweisung

```
variablenname = strukturkonstante
variablenname = variablenname     !gleichen Typs!
```


Selbstdefinierte Typen

■ Beispiele

```
kalender(1)  = DATUM(1,'Jan',1996)
kalender(41)%tag = 10
kalender(41)%monat = 'Feb'
kalender(41)%Jahr = 1941

gehaltsliste(35) = MITARBEITER ('Mueller', 'Horst',&
     DATUM(10,'Feb',1941),4523.12,1212.00,.TRUE.,&
     gehaltsliste(25))

gehaltsliste(42)  = gehaltsliste(35)
gehaltsliste(42)%vorname = 'Anton'
gehaltsliste(42)%alphanext = gehaltsliste(35)
! Kapitel 1.5
```

Selbstdefinierte Typen

weitere Beispiele

- Typdefinition

```
TYPE :: ERSATZTEIL
  SEQUENCE
  CHARACTER *10 pkw_typ
  CHARACTER *10 part_nb
END ERSATZTEIL

TYPE :: PKW
  SEQUENCE
  CHARACTER *10 farbe
  INTEGER fahrgestellnummer
  INTEGER zulassungsjahr
  TYPE (ERSATZTEIL), DIMENSION (100) :: vorderachse
END TYPE PKW
```


Selbstdefinierte Typen

weitere Beispiele

- Typvereinbarungen

```
TYPE (PKW), DIMENSION (200) :: lager
```

- Zuweisungen

```
lager(110)%vorderachse(12)%part_nb = '1000G1019'
lager(1)%farbe = 'RAL312'
```

Selbstdefinierte Typen

```
PROGRAM personal
IMPLICIT NONE
INTEGER, PARAMETER :: max = 10
INTEGER, PARAMETER :: fd = 10
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
!  CHARACTER, DIMENSION (:), POINTER :: beurteilung! Kap. 1.5
END TYPE MITARBEITER

TYPE (MITARBEITER), DIMENSION (max) :: pa! Personal_akte
CHARACTER *4 funktion
CHARACTER zeichen
INTEGER i
LOGICAL :: ende = .FALSE.
```


Selbstdefinierte Typen

```
OPEN(fd,FORM='UNFORMATTED',file = 'database.personal')
   ! Oeffnen der Datei
DO WHILE (.NOT. ende)      !Funktionen

   PRINT *, ' '
   PRINT *, 'PRINT *, 'Eingabe der gewünschten Fkt:'; PRINT *,
' '
   PRINT *, 'init = Initialisiere Datenbasis'
   PRINT *, 'read = Lesen der Datenbasis'
   PRINT *, 'einz = Einzel-Eingabe'
   PRINT *, 'eing = Gruppeneingabe'
   PRINT *, 'korr = Korrektur;'
   PRINT *, 'ausg = Ausgabe;'
   PRINT *, 'sumg = Summe Gehaelter;'
   PRINT *, 'suml = Summe Lohnsteuer;'
   PRINT *, 'pers = Personalliste;'
   PRINT *, 'ende = Beenden des Programms'
   PRINT *, ' '
   PRINT *, 'Funktion?'
   READ *, funktion
   PRINT *, ' '
```

Selbstdefinierte Typen

```
SELECT CASE (funktion)
   CASE ('init')
      PRINT *,'Sind Sie sicher? (y)'
      READ *,zeichen
      if (zeichen /= 'y') EXIT
      DO i = 1,max
            pa(i)%name = 'frei'
            pa(i)%vorname = 'frei'
      END DO
   CASE ('read')
      READ(fd) (pa(i), i=1,max)
   CASE ('einz')
      DO i = 1,max
            IF (pa(i)%name == 'frei') THEN
                  Print *,'Name:'
                  READ *, pa(i)%name
                  PRINT *, 'Vorname'
                  READ * , pa(i)%vorname
                  EXIT
            ELSE
                  CYCLE
            END IF
      END DO
```


Selbstdefinierte Typen

```
      CASE ('eing')
         PRINT *, 'Not yet implemented'
      CASE ('korr')
         PRINT *, 'Not yet implemented'
      CASE ('ausg')
         PRINT *, 'Not yet implemented'
      CASE ('sumg')
         PRINT *, 'Not yet implemented'
      CASE ('suml')
         PRINT *, 'Not yet implemented'
      CASE ('pers')
         DO i= 1,max
               PRINT *, pa(i)%name, ';', pa(i)%vorname
         END DO
      CASE ('ende')
         PRINT *, 'Programmende'
         WRITE(fd) (pa(i), i = 1,max)
         ende = .TRUE.
      CASE DEFAULT
         PRINT *, 'Falsche Eingabe!'
      END SELECT
END DO
END PROGRAM personal
```

3.1.5 Zeiger (Pointer)

- Pointer Definition
  - Der Inhalt einer Pointervariablen ist die Adresse eines Objekts auf das der Pointer zeigt.
  - Konsequenterweise müßten Pointer - wie in anderen Sprachen (z. B. in C) - ein spezieller Datentyp sein. Da man in Fortran90 nicht direkt auf einen Pointer zugreifen kann (keine Adreßarithmetik!), handelt es sich nur um ein Attribut eines Datentyps.
  - In Fortran90 finden Pointer hauptsächlich Verwendung bei der dynamischen Speicherverwaltung (während der Laufzeit des Programms; detailiert in Kapitel 1.8 dargestellt) und beim Aufbau von Datenstrukturen wie verketteten Listen, Baumstrukturen, etc.

Zeiger (Pointer)

- Pointer-Vereinbarung

```
REAL, POINTER :: zeiger, next, current !Typ und Attribut
! oder
REAL zeiger, next, current                   !Typ
Pointer zeiger, next, current                !Attribut
```

- Aus der Pointerdefinition folgt:
  - das Pointer-Attribut darf nur für Variablen oder Funktionen spezifiziert werden
  - folgende weitere Attribute sind unzulässig:
    ALLOCATABLE, EXTERNAL, INTENT, INTRINSIC, PARAMETER, TARGET
  - ein Pointer darf weder bei einer Typvereinbarung noch mit Hilfe einer DATA-Anweisung oder E/A initialisiert werden

Zeiger (Pointer)

Pointer-Zuweisung/Freigabe

- Will man einem Pointer eine Variable (oder Funktion) zuweisen, so muß für diese Datengröße das TARGET-Attribut spezifiziert sein.
  - Spezifikation des TARGET-Attributs

```
INTEGER, TARGET :: feld, person      ! Typ und Attribut
! oder
INTEGER feld, person    ! Typ
TARGET feld, person     ! Attribut
```

  - Zuweisung

```
pointer => zielvariable
! oder
ALLOCATE (pointer [,pointer]...) [, STAT=statusvariable]
```

  - Freigabe von Pointerzielen

```
pointer => neuezielvariable          !Überschreiben
! oder
DEALLOCATE (pointer [,pointer]...) [, STAT=statusvariable]
```


Zeiger (Pointer)

Pointer-Zuordnungsstati

- zugeordnet:
  - dem Pointer wurde eine Variable mit dem TARGET-Attribut zugewiesen.
  - dem Pointer wurde ein Pointer zugewiesen, der selbst den Status “zugeordnet” besitzt.
  - dem Pointer wurde mit der ALLOCATE-Anweisung ein Speicherbereich zugeordnet
- nicht zugeordnet:
  - dem Pointer wird mit der NULLIFY-Anweisung eine Zuordnung entzogen (Keine damit verbundene Speicherfreigabe!).
  - NULLIFY (pointer [,pointer]...)
  - dem Pointer wird mit der DEALLOCATE-Anweisung ein Speicherbereich entzogen, der ihm mit einer ALLOCATE-Anweisung zugeordnet wurde.
  - dem Pointer wird ein Pointer mit dem Status “nicht zugeordnet” zugewiesen.

Zeiger (Pointer)

Pointer-Zuordnungsstati(cont)

- undefiniert:
  - dem Pointer wurde noch kein Ziel zugewiesen (weder '=>' noch **`ALLOCATE`**)
  - das zugeordnete Ziel wurde auf "anderer Weise" entfernt;
    z. B. ein **`DEALLOCATE`** auf einen Zeiger mit dem gleichen Ziel
  - dem Pointer wird ein Pointer zugewiesen mit dem Staus "undefiniert"
  - wenn das Pointerziel bei Ausführung einer **`END`**- oder **`RETURN`**-Anweisung undefiniert wird. (Kapitel 1.6)

Zeiger (Pointer)

ASSOCIATED-Function

- Intrinsic (vordefinierte) Funktion

  `Syntax: ASSOCIATED (pointervar [,targetvar]`

  - liefert den Wert **`.TRUE.`** beim Aufruf

    `ASSOCIATED (pointervar)`

  wenn **`pointervar`** irgendeinem Ziel zugeordnet ist und beim Aufruf

    `ASSOCIATED (pointervar, targetvar)`

  wenn **`pointervar targetvar`** zugeordnet ist.

  - sonst den Wert **`.FALSE.`**

Beispiele

- zu Datenstrukturen, verkettete Listen, Baumstrukturen, etc
  siehe auch Vorl. mit Übungen : "Programm- und Datenstrukturen"
- zu **`ALLOCATE, DEALLOCATE, NULLIFY`**
  siehe auch Feldverarbeitung (Dynamische Speicherverwaltung) Kap. 1.8

Zeiger (Pointer)

```
PROGRAM queue

! Warteschlangen-Demo-Programm adaptiert aus:
! Cooper Redwine; "Upgrading to FORTRAN 90", Springer-Verlag,
! 1995
!
! Queue - Policy: first in - first out (Hier zunaechst nur
! Eingabe)

IMPLICIT NONE
TYPE :: QUEUE_ELEMENT
   CHARACTER (20) :: data
   TYPE (QUEUE_ELEMENT), POINTER :: next
END TYPE

! Zeiger auf des erste, letzte, aktuelle Element: first, last,
! this

TYPE (QUEUE_ELEMENT), POINTER :: first, last, this

CHARACTER(20) :: buffer
CHARACTER *4 :: funktion
LOGICAL :: ende = .FALSE.
INTEGER :: io_stat = 1
```


Zeiger (Pointer)

```
   DO WHILE (.NOT. ende)         !Funktionen
   PRINT *, ' '
   PRINT *, 'Eingabe der gewünschten Fkt:'
   PRINT *, ' '
   PRINT *, 'init = Initialisiere Queue'
   PRINT *, 'read = Lese Queue'
   PRINT *, 'erze = Erzeuge Elemente'
   PRINT *, 'dele = Loesche 1. Element'
   PRINT *, 'ende = Beenden des Programms'
   PRINT *, ' '
   PRINT *, 'Funktion:'
   READ *, funktion
   PRINT *, ' '
```

Zeiger (Pointer)

```
SELECT CASE (funktion)
   CASE ('init')
      NULLIFY (first, last)! Initialisiert: leere Queue
   CASE ('read')

      ! Durchlaufe die Warteschlange und drucke Inhalte als
      ! Liste aus.

      this => first           ! aktuelles Element gleich
                              ! 1. Element
      PRINT *, 'Ausgabe'; PRINT *, ' '
      DO WHILE (ASSOCIATED (this))
            PRINT *, this%data
            this => this%next ! aktuelles Element gleich
                              ! naechstes Element
      END DO
```


Zeiger (Pointer)

```
CASE ('erze')
   PRINT *, 'Eingabe:'
   DO WHILE (io_stat >= 0)
      READ (*, "(A)", IOSTAT = io_stat) buffer

      IF (.NOT. ASSOCIATED (first)) THEN

      ! Queue ist leer; fordere den Speicher fuer das
      ! 1. Element an

            ALLOCATE (first)
            last => first     ! Beide Zeiger zeigen aufs
                              ! gleiche(erste)Element
      ELSE
      ! Queue ist nicht leer, neues Element gleich
      ! letztes Element

            ALLOCATE (last%next)
            last => last%next
      END IF
      last%data = buffer
      NULLIFY (last%next)     ! Letztes Element hat
                              ! keinen Nachfolger
   END DO
```

Zeiger (Pointer)

```
      CASE ('dele')
         PRINT *, 'Not yet implemented'
      CASE ('ende')
         PRINT *, 'Programmende'
         ende = .TRUE.
      CASE DEFAULT
         PRINT *, 'Falsche Eingabe!'
      END SELECT
END DO

END PROGRAM queue
```


3.1.6 Unterprogramme

- Definitionen/Terminologie
  - Programmeinheiten (programunits) sind die Bausteine eines ausführbaren Programms
    - Hauptprogramm (genau einmal)
    - ext. Unterprogramme
    - Module
    - Blockdatenprogrammeinheiten
  - Sie enden mit der END-Anweisung, liegen in separaten Dateien, sind getrennt übersetzbar, jedoch bis auf das Hauptprogramm nicht direkt ausführbar.
  - Ein Hauptprogramm ist ausführbar, wenn es keine offenen 'Externreferenzen' gibt - ausgenommen Bibliotheksreferenzen, wie z.b. SIN(x), COS(x), etc.
    Diese Eigenschaft haben unsere bisher geschriebenen Programme.

Unterprogramme

- Die weiteren Programmeinheiten sind nur dann ausführbar (getrennt testbar), wenn ein entsprechendes Hauptprogramm als “Testrahmen” zugebunden wird.
- Nach der Übersetzung sind diese Einheiten zunächst separate Bindemodule; sogenannte ".o-files"

- Unterprogramme werden unterschieden
  - nach ihrem “Definitionsort”
    - extern
    - eingebettet
    - vordefiniert in Bibliothek
  - und ihrer Art
    - Subroutine
    - Funktion
    - Formelfunktion

Unterprogramme

- Subroutine
  - Subroutinendefinition

```
[...] SUBROUTINE sname [(...)]        !(SUBROUTINE-Anweisung)
    [Spezifikationsteil]
    ...
        Subroutinenrumpf
    ...
[CONTAINS]
    [interne Unterprogramme]
    ...
END [SUBROUTINE [sname]]
```

  - Beispiele:

```
SUBROUTINE sname()                      !(leere Parameterliste)
SUBROUTINE sname (formalparameter[,formalparameter] ...)
```

  - Subroutinenaufruf

```
CALL sname
CALL sname()
CALL sname(Aktualparameter[,Aktualparameter]...)
```

Unterprogramme

- Beispiel - eingebettetes Unterprogramm (CONTAINS)

```
PROGRAM summentest
IMPLICIT NONE
REAL a, b, ergebnis

   Print *, 'Eingabe: summand1 summand2 <r>'
   READ *, a,b

   call summe (a,b,ergebnis)

   PRINT *, 'Ergebnis = ',ergebnis

CONTAINS

      SUBROUTINE summe (summanda, summandb, erg)
      REAL summanda, summandb, erg
         erg = summanda + summandb
      END SUBROUTINE summe

END PROGRAM summentest
```


Unterprogramme

- Geltungsbereich/Gültigkeitsbereich
  - sagt aus, in welchen Bereichen eine Variable zugreifbar ist
    - lokal Variable sind nur gültig innerhalb eines Geltungsbereichs z. B. in einem Unterprogramm
    - globale Variable sind in der gesamten Programmeinheit, in der sie definiert wurden, zugreifbar, falls sie nicht durch eine lokale Definition “verdeckt” werden.
      (“Blickt” man aus der Aufrufschachtelung zurück in Richtung Aufrufer, so sind sämtliche globale Daten der aufrufenden Ebenen zugreifbar, es sei denn, sie sind durch lokale Vereinbarungen verdeckt.)

Unterprogramme

- Geltungsbereich/Gültigkeitsbereich(cont)
  - Transfer von Informationen zwischen Geltungsbereichen durch:
    - Parameter
      - Eingangsparameter
      - Ausgangsparameter; Ergebnisparameter
      - beide Richtungen
    - globale Daten
      - im äußersten Geltungsbereich definiert;
        also programmeinheitsglobal
      - COMMON-Bereiche
        Überall dort gültig, wo definiert; auch über Programmeinheiten hinweg

Unterprogramme

- Parameter
  - Formalparameter (bei der Definition)
  - Aktualparameter (beim Aufruf)

    Zum Zeitpunkt des Aufrufs werden die Formalparameter und die Aktualparameter einander zugeordnet, d.h. eine “Formal-Variable” im Unterprogramm referenziert den gleichen Speicher wie die “Aktual-Variable” im aufrufenden Programm.

  Man unterscheidet

    - positionsgerechte Schreibweise
    - Schlüsselwortschreibweise z. B.

      formalparameter = aktualparameter

  Die gemischte Schreibweise ist möglich. Beginnt man jedoch mit der Schlüsselwortschreibweise, so müssen auch alle folgenden Parameter in Schlüsselwortschreibweise folgen.

Unterprogramme

INTENT-Attribut/Anweisung

- Mit dem INTENT-Attribut kann die Art der Verwendung von Formalparametern spezifiziert werden:
  - Das INTENT(IN)-Attribut besagt, daß der Formalparameter ein Eingabeparameter ist. D.h. dieser Formalparameter darf innerhalb des Unterprogramms nicht redefiniert werden, er darf also z. B. nicht auf der linken Seite einer Zuweisung auftauchen oder Variable einer READ-Anweisung sein.
    Der Aktualparameter kann also sowohl eine Variable als auch eine Konstante sein.
  - Das INTENT(OUT)-Attribut besagt, daß der Formalparameter ein Ausgabeparameter ist. D. h. der Formalparameter muß erst definiert (ein Wert zugewiesen) werden, ehe im Unterprogramm auf ihn zugegriffen wird. Hier **muß** es sich also um eine Variable handeln.

Unterprogramme

INTENT-Attribut/Anweisung

- Mit dem INTENT-Attribut kann die Art der Verwendung von Formalparametern spezifiziert werden (cont):
  - Das INTENT(INOUT)-Attribut besagt, daß der Parameter sowohl Eingabe- als auch Ausgabeparameter sein soll. Auch hier **muß** es sich also um eine Variable handeln.

Wir werden im Rahmen dieses Kurses die Formalparameter mit INTENT-Attributen versehen denn: Keine Angabe = INTENT(INOUT), es sei denn, der Aktual-Parameter ist eine Konstante.

© Claus-Uwe Linster – pps3.1-2_spmd_f90: 19/3/98

Unterprogramme

- Beispiel:

```
PROGRAM summentest
IMPLICIT NONE
REAL a, b, ergebnis

   Print *, 'Eingabe: summand1 summand2 <r>'
   READ *, a,b

   call summe (a,b,ergebnis)

   PRINT *, 'Ergebnis = ',ergebnis

CONTAINS

      SUBROUTINE summe (summanda, summandb, erg)
      REAL, INTENT(IN) :: summanda, summandb
      REAL, INTENT(out) :: erg
         erg = summanda + summandb
      END SUBROUTINE summe

END PROGRAM summentest
```


Unterprogramme

- FUNKTION

```
[...] FUNCTION fname [(...)]!(FUNCTION-Anweisung)
   [Spezifikationsteil]
   ...
      Funktionsrumpf
      ...
[CONTAINS]
   [interne Unterprogramme]
   ...
END [FUNCTION [fname]]
```

  - der wesentliche Unterschied einer Funktion zu einer Subroutine ist, daß der Funktionsname gleichzeitig auch der Name der (lokalen) Ergebnisvariablen ist. Deshalb hat die Funktion einen Typ!
  - Der Aufruf einer Funktion kann überall dort stehen, wo auch eine Variable stehen darf; nur auf der linken Seite einer Zuweisung (natürlich) nicht.
  - Der Funktionsaufruf darf auch ein separates Statement sein - analog zur Subroutine - allerdings ohne “CALL”; jedoch mit “()”.

Unterprogramme

```
( ~uelinste/tmp/newton_fkt.f90 )

PROGRAM newton
REAL eingabe
! Author: Uwe Linster
! Anfordern und einlesen der Zahl, deren Quadratwurzel
! berechnet werden soll

   PRINT *, ' '

   ! Ausgabe auf das Sichtgerät ohne Zeilenvorschub
   ! Nichtvorrueckende Ausgabe: Kapitel 1.10

   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Zu berechnen ist die Quadratwurzel aus: ')

   READ *, eingabe
   PRINT *, ' '; PRINT *, ' '

   DO WHILE(eingabe <= 0)
      PRINT *, ' Die Eingabedaten entsprechen nicht den &
               &Anforderungen des Programms (Zahl > 0!):'
      READ *, eingabe
   END DO
```


Unterprogramme

```
!Berechnung

   ergebnis = sqroot(eingabe)

!Ergebnisausgabe

   PRINT *, ' Die Quadratwurzel aus ', eingabe ;PRINT *, ' '
   PRINT *, ' ist ', ergebnis
! Oder auch gleich:
!  PRINT *, ' ist ', sqroot(eingabe)

PRINT *, ' '; PRINT *, ' Ende des Programms'
```

Unterprogramme

```
CONTAINS

REAL FUNCTION sqroot (a)
!Berechnung der Quadratwurzel nach dem Newton'schen Verfahren
!Eingabeparameter:a; Startwert der Iteration: 1
!Iterationsformel:x(i+1) = (x(i)*x(i) + a) / (2*x(i))

   REAL , INTENT(IN) :: a
   REAL x
      !a Größe, deren Quadratwurzel zu berechnen ist
      !x Berechnete Approximation der Quadratwurzel

   !Anfangsapproximation
   x = 1
   !Iteration
   DO WHILE (.NOT. (ABS(x*x - a) <= 1E-4))
      x = (x*x + a) / (2.*x)
   END DO
   sqroot = x
END FUNCTION sqroot

END PROGRAM newton
```


Unterprogramme

- COMMON-BEREICHE
  - Durch COMMON-Bereiche werden globale Speicherbereiche spezifiziert. Variable sind in den Geltungsbereichen direkt zugreifbar, in denen die COMMON-Bereiche spezifiziert wurden; auch über verschiedene Programmeinheiten hinweg.
    Man unterscheidet:
    - einen unbenannten COMMON-Block

```
Syntax:COMMON variablenliste
```

    - und beliebig viele benannte COMMON-Blöcke

```
Syntax:COMMON /blockname/ variablenliste [, /blockname/]
              variablenliste]...
```

    - auch die Mischschreibweise ist möglich:

```
Syntax:COMMON  variablenliste [[,]/[blockname]/
              variablenliste]...
```

Unterprogramme

- COMMON-BEREICHE(cont)
  - Eigenschaften
    - Durch Angabe der Variablen wird eine Speicherfolge festgelegt; um Fehler zu vermeiden sollte man mit “copy” und “paste” arbeiten.
    - Variablennamen in verschiedenen Geltungseinheiten dürfen verschieden sein; das sollte man jedoch nur tun, wenn es dafür auch einen plausiblen Grund gibt. Man greift auf dieselbe globale Speicheradresse zu, das sollte man durch den gleiche Variablennamen auch dokumentieren!
    - Initialisiert werden COMMON-Bereiche mit einer BLOCK DATA-Programmeinheit; jedoch nur die benannten Blöcke.
    - Selbstdefinierte Datentypen müssen das Attribut SEQUENCE besitzen. (Speicherfolge!)

Unterprogramme

- COMMON-Bereich - Beispiel (~uelinste/tmp/common_demo_bubble.f90

```
PROGRAM bubble
IMPLICIT NONE

! Zweck des Programms, Loesungsverfahren:
!
! Die Eintraege in der Datei titel sollen in lexikographisch
! sortierter Reihenfolge gedruckt werden. Das Programm geht
! davon aus, dass die Datei maximal 200 Eintraege besitzt und
! jeder Eintrag aus hoechstens 20 Zeichen besteht.
! Als Sortierverfahren wird 'bubble sort' verwendet.
!------------------------------------------------------------
! Globale Variable; in COMMON-Bereiche spezifiziert
!    name Feld zur Aufnahme der Eintraege aus der Dateri titel
!    max Anzahl der Eintraege
!  fd Kanalnummer (Filedescriptor; Unit)
!------------------------------------------------------------
! Initialisierung
!
! Ausgabe der Gesamtueberschrift
!
PRINT *, 'Eintraege der Datei titel lexikograpgisch sortiert:'
```

Unterprogramme

```
! Oeffnen der Datei
!
   CALL oeffnen_datei()

! Einlesen der Daten
!
   CALL einlesen()
!
! Sortieren
!
   CALL bubblesort()
!
! Ausgabe des Ergebnisses
!
   CALL ausgeben_screen()
   CALL ausgeben_datei()
```


Unterprogramme

```
! Die Unterprogramme:

CONTAINS

SUBROUTINE oeffnen_datei()

! Oeffnen der Datei "titel" mit Kanalnummer 10
! Global Data:

INTEGER fd
COMMON /datei/ fd
   fd = 10      ! Entfaelt bei BLOCK-DATA-INIT!!
! Local DATA: none

   open(fd, FORM = 'FORMATTED', file = 'titel')

END SUBROUTINE oeffnen_datei
```

Unterprogramme

```
SUBROUTINE einlesen()

! Einlesen der Eintraege aus der Datei "titel" in das Feld !
"name".
! Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

!Local Data:
INTEGER i

   DO i = 1, 200
      READ(fd, 1000, END = 1010) name(i)
1000  FORMAT (a)
   END DO

1010max = i - 1
   PRINT *, 'Gelesene Datensaetze: ',max

END SUBROUTINE einlesen
```


Unterprogramme

```
SUBROUTINE bubblesort()
! Sortiert nach dem "bubble-sort"-Verfahren
! Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

!Local Data

CHARACTER *20 :: hilf
LOGICAL flag
INTEGER i, j, dl

!  flag   zeigt an, ob bei einem Durchlauf Aenderungen in der
!              Anordnung vorgenommen wurden
!              (.FALSE.: Keine Umordnungen
!               .TRUE. : Es wurden Umordnungen vorgenommen)
!  hilf   Hilfsfeld waehrend des Vertauschens zweier Eintraege
!  dl Anzahl der Sortierschleifen
!
```

Unterprogramme

```
! Der Sortiervorgang

   dl = 0
   DO  i = max, 2, -1
      flag = .FALSE.

      DO j = 1, i - 1
          IF (name(j) > name(j + 1)) THEN
               hilf = name(j)
               name(j) = name(j + 1)
               name(j + 1) = hilf
               flag = .TRUE.
               dl = dl + 1
          END IF
        END DO

        if (.NOT. flag) THEN
         PRINT *, 'Anzahl der Durchlaeufe: ', dl
         EXIT
      END IF

   END DO

END SUBROUTINE bubblesort
```


Unterprogramme

```
SUBROUTINE ausgeben_screen()

!Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max
COMMON name, max

! Local Data:

INTEGER i

   DO i=1,max
      Print *, name(i)
   END DO

END SUBROUTINE ausgeben_screen
```

Unterprogramme

```
SUBROUTINE ausgeben_datei()

!Global Data

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

! Local Data

INTEGER i

   REWIND (fd)
   WRITE ( fd, 1030 )(name(i), i=1, max)
1030FORMAT (a)

END SUBROUTINE ausgeben_datei

END PROGRAM bubble
```


Unterprogramme

- BLOCK DATA
  - Eine BLOCK-DATA-Programmeinheit
    - ist eine nichtausführbare Programmeinheit, sie dient der Initialisierung von Variablen in **benannten** COMMON-Bereichen.
    - COMMON-Blöcke sind globale Speicherbereiche - gültig auch über Programm-Unit-Grenzen hinweg. Eine Initialisierung darf also Programmsystemweit nur einmal erfolgen. Dies könnte innerhalb einer “ausgezeichneten” Programmeinheit erfolgen oder wie hier in einer speziellen Programmeinheit

```
BLOCKDATA [name]
     [Spezifikationsteil]
     .
     .
     .
  END BLOCKDATA [name]
```

Unterprogramme

- BLOCK DATA(cont)

```
Beispiel
BLOCKDATA car

CHARACTER vorderachse *14, hinterachse *10, motor *11
INTEGER anzahl, seriennummer, partnb

COMMON /ersatzteile/ vorderachse, hinterachse, motor
COMMON /attribute/ anzahl, seriennummer, partnb

DATA vorderachse, hinterachse /'Schwingachsen', `Inkaachsen'/
DATA motor /'Dieselmotor'/
DATA anzahl, partnb /250, 4711/ ! seriennummer bleibt
uninitialisiert

END BLOCKDATA car
```

- Da diese Programm-Unit in einer eigenen Datei liegt muß beim Aufruf des Compilers diese Datei als weiterer Parameter angegeben werden:; z. B.

```
f90 hauptprogramm.f90 blockdata.f90 -o programm.out
```


Unterprogramme

- BLOCK DATA
  - Beispiel für Programm “bubble”:

Datei: bubble_blockdata.f90

```
BLOCKDATA bubble

   INTEGER fd

   COMMON /datei/ fd

   DATA fd /10/

END BLOCKDATA bubble
```

- Aufruf des Compilers:

```
f90 common_demo_bubble.f90 bubble_blockdata.f90 -o bubble.out
```

Unterprogramme

FORMELFUNKTION

- eine Formelfunktion ist eine Funktion, die nur aus einer einzigen Anweisung besteht und
- sie ist nur innerhalb einer Geltungseinheit aufrufbar.
- Formelfunktionsdefinition:

```
Syntax:ffname() = ausdruck oder
ffname(formalparameter[,formalparameter]...) =
                          ausdruck
```

- Sowohl ’ausdruck’ als auch ’formalparameter’ sind skalare Datenobjekte.
- Formelfunktionsaufruf:

```
Syntax:ffname() oder
ffname(formalparameter[,formalparameter]...)
```

- Der Typ der Formelfunktion wird separat (wie bei der Typvereinbarung einer Variablen) vereinbart; dieses gilt auch für die Formalparameter.

Unterprogramme

- Formelfunktion - Beispiel (~uelinste/tmp/capslock_fkt.f90)

```
PROGRAM capslock
IMPLICIT NONE
!
! Das Programm liest Kleinbuchstaben ein, wandelt sie
! in Grossbuchstaben um und druckt diese aus.
!  Variable
!
      CHARACTER, DIMENSION(20) :: zeichenin, zeichenout
      INTEGER i

   CHARACTER gross, klein

!  Formelfunktion:

   gross(klein) = CHAR ( ICHAR(klein) + ICHAR('A') - ICHAR('a'))
```

Unterprogramme

```
!  Einlesen der Zeichen

      PRINT *, ' '
   PRINT *, 'Kleinbuchstaben; max. 20; entweder durch "blank"
getrennt -'
   PRINT *, 'Abschluss <r> oder einzeln durch <r> getrennt!'
   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Eingabe: ')
      READ *, (zeichenin(i), i=1,20)

   DO i = 1,20
      zeichenout(i) = gross(zeichenin(i))
   END DO

   PRINT *, 'Ausgabe:',zeichenout

END PROGRAM capslock
```


Unterprogramme

- Formelfunktion - Beispiel (~uelinste/tmp/kredit_fkt.f90)

```
PROGRAM kredit
IMPLICIT NONE
!
!  Das Programm berechnet die Hoehe der monatlichen Raten fuer
! einen vorgebbaren Kaufpreis bei einzugebender Laufzeit. Raten
!  und Zinsen werden von zwei Jahren bis zu einer angebbaren
!  maximalen Laufzeit in Jahresschritten berechnet. Untere und
!  oebere Grenze des Zinssatzes werden ebenfalls eingelesen.
! ------------------------------------------------------------
!  Variable
!
      REAL preis, mrate, p
      INTEGER n, zmin, zmax, zsatz, zs, year, yearmax
!
! Berechnung der Monatsrate fuer die verschiedenen Zinssaetze
!          formel: r*p/(1-(1+r)**(-n)) mit r = zinssatz/12/100
!  hier als Formelfunktion realisiert:

   mrate(zs,p,n) = ((zs/100./12.) * p) / &
            (1.-(1. + (zs/100./12.))**(-n))
```

Unterprogramme

```
!        preis      Kaufpreis (Eingabe)
!       yearmax   maximale Laufzeit der Ratenzahlung (Eingabe)
!        zmin       minimaler Zinssatz (Eingabe)
!        zmax       maximaler Zinssatz (Eingabe)
!        year       Laufvariable fuer die Laufzeit
!        n          Zahl der Raten
!        zsatz      Laufvariable fuer den Zinssatz
! ------------------------------------------------------------
! Initialisierung
!
!  Einlesen von Kaufpreis, maximaler Laufzeit,
!  minimalem und maximalem Zinssatz
!
      PRINT *, ' '
   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Kaufpreis (REAL): ')
      READ *, preis
   WRITE (*, 2000, ADVANCE='NO')
2000FORMAT ('Laufzeit in Jahren (groesser als 2): ')
      READ *, yearmax
     WRITE (*, 3000, ADVANCE='NO')
3000FORMAT ('minimaler und maximaler Zins (in Prozent
(INTEGER)): ')
      READ *, zmin, zmax; PRINT *, ' '
```


Unterprogramme

```
! Aeussere Schleife

    DO year = 2, yearmax

!  Tabellenueberschrift
        PRINT *, ' '
        PRINT *, ' '
      PRINT *, ' Zinsen fuer eine Laufzeit von ', year, ' Jahren'
        PRINT *, ' '
       PRINT *, ' Zins-  Zahl der     Rate          Gesamtzins'
        PRINT *, ' satz    Raten       (DM)              (DM)'

! Innere Schleife
      DO zsatz = zmin, zmax
         n = 12.*year
          PRINT *, ' ', zsatz, '     ', n, ' ', &
               mrate (zsatz, preis, n) &
               ,' ', n * (mrate (zsatz, preis, n)) -preis
      END DO
    END DO
!
!        Abschlussausgabe
      PRINT *, ' ' ;PRINT *, ' Berechnung abgeschlossen.'
END PROGRAM kredit
```

Unterprogramme

- Rücksprung
  - Es existieren drei alternative Methoden
    - der “normale” Rücksprung bei Erreichung des END-Statements
    - der “Ausnahme” - Rücksprung, wenn aufgrund einer Bedingung die Aufgabe des Unterprogramms erfüllt ist - etwa analog zum EXIT-Aussprung aus einer DO-Schleife

```
Syntax:RETURN
```

Bei beiden Methoden wird mit dem 1. ausführbaren Statement fortgefahren, das dem Unterprogrammaufruf folgt.

- der “Ausnahme” - Rücksprung mit wählbarer Fortsetzungsadresse (label).

```
Syntax:RETURN ganzzahliger_ausdruck
```

- Diese Möglichkeit wird nur der Vollständigkeit erwähnt (Im Falle, daß man mit vorhandenen Programmsystemen umgehen muß). Die Verwendung solcher Konstrukte führt zu undurchsichtigen, nicht mehr wartbaren Programmsystemen.

Unterprogramme

- SAVE-Anweisung
  - Mit dem SAVE-Attribut wird der Wert einer **lokalen** Variablen in einem Geltungsbereich über die “Lebensdauer” eines Aufrufs hinaus “gesichert”. D. h., daß der Wert der Variablen auch beim nächsten Aufruf wieder verfügbar ist. Man spricht auch von “*Static”* - Variablen.
  - Das SAVE-Attribut ist nur erlaubt bei
    - lokalen Variablen
    - benannten COMMON-Blöcken (hier ist Vorsicht geboten, ist doch ein COMMON-Block ein globaler Speicherbereich!?!)
  - Das SAVE-Attribut ist insbesondere nicht erlaubt bei
    - einzelnen Variablen, die in einem COMMON-Bereich enthalten sind
    - Formalparametern
    - Funktionsvariablen
    - Namensgruppen (Siehe Kapitel 1.10)

Unterprogramme

SAVE-Anweisung

Attributspezifikation

- gleichzeitig mit der Typvereinbarung

```
REAL, SAVE :: skonto, summe
INTEGER, SAVE :: max
```

- oder getrennt mit einer SAVE-Anweisung

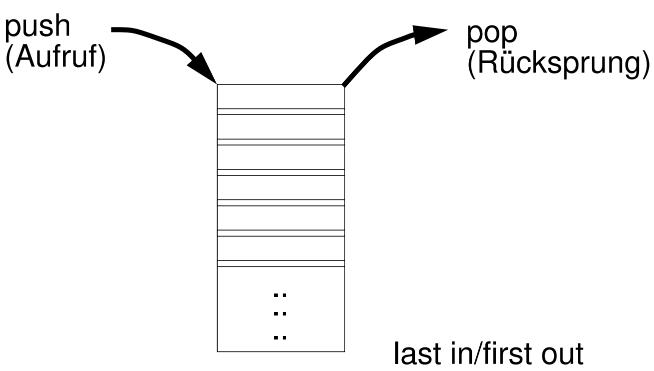
```
REAL skonto, summe
SAVE skonto, summe

INTEGER max
SAVE max
```

Eine SAVE-Anweisung ohne Angabe von Variablennamen bedeutet, das sämtliche lokale Variable in diesem Geltungsbereich diese Spezifikation erhalten.

Rekursive Unterprogramme

Unterprogramme (SUBROUTINE oder FUNCTION) sind dann rekursiv, wenn sie sich selbst direkt oder indirekt aufrufen.

Wenn Unterprogramme das Attribut RECURSION besitzen, muß der Compiler dafür Sorge tragen, daß jeweils bei einem rekursiven Aufruf die lokalen Variablen gerettet und nach dem Rücksprung aus einer tieferen Aufrufebene wieder bereitgestellt werden. Dies ist eine typische Anwendung für einen Kellerspeicher (Stack)!



© Claus-Uwe Linster – pps3.1-2_spmd_f90: 19/3/98

Rekursive Unterprogramme

- Wann verwendet man rekursive Unterprogramme?
  - Es gibt eine Anzahl von Algorithmen, die "in sich" rekursiv definiert sind. Bei der iterativen Implementierung solcher Algorithmen ist der Programmieraufwand in der Regel beträchtlich und mit einer großen Anzahl von Abfragen behaftet, wie einige Beispiele des Kapitels noch zeigen werden.
  - Als einfaches Beispiel - zum Verständnis - wurde die Berechnung von Fakultäten gewählt, tatsächlich ist es in diesem Fall allerdings effizienter, eine Schleife zu implementieren!

```
PROGRAM recursion
IMPLICIT NONE
INTEGER n, erg
   WRITE (*, FMT = "('n! ; Welches n? : ')", ADVANCE = 'NO')
   READ *, n
   CALL fak (n, erg)
   PRINT *, 'Fakultaet von: ',n ,' = ', erg
   erg = fakfunc (n)
   PRINT *, 'Fakultaet von: ',n ,' = ', erg
CONTAINS
```


Rekursive Unterprogramme

- Rekursive Subroutine

```
RECURSIVE SUBROUTINE fak (fn, ferg)

INTEGER, INTENT (IN) :: fn
INTEGER, INTENT (OUT) :: ferg

   IF (fn == 0) THEN
        ferg = 1
   ELSE
        CALL fak (fn – 1, ferg)
        IF (ferg < 0) THEN
             PRINT *, 'underflow'
             STOP
        END IF
        ferg = fn * ferg
   END IF

END SUBROUTINE fak
```

Rekursive Unterprogramme

▲ Rekursive Funktion

```
RECURSIVE INTEGER FUNCTION fakfunc (fn) RESULT (ferg)

INTEGER, INTENT (IN) :: fn

   IF (fn == 0) THEN
        ferg = 1
   ELSE
        ferg = fn * fakfunc (fn - 1)
        IF (ferg < 0) THEN
             PRINT *, 'underflow'
             STOP
        END IF
   END IF

END FUNCTION fakfunc

END PROGRAM recursion
```


Unterprogramme

■ (Noch) nicht behandelt

- Parameterübergabe bei nicht skalaren Datengrößen (Siehe Kapitel 1.7- Feldverarbeitung)
- Module (Siehe Kapitel 1.11 - Module)
- Externe Unterprogramme (Siehe Kapitel n.b.)
- Nebeneinganspunkte - ENTRY-Anweisungen (Siehe Kapitel n.b.)
- Optionale Formalparameter (Siehe Kapitel n.b.)

Programmierung Paralleler Systeme

3. Programmiermodell “Single Programm Multiple Data”

3.1 Fortran95

3.1 .4 Selbstdef. Datentypen (Strukturen)

3.1.5 Zeiger(Pointer)

3.1.6 Unterprogramme

Claus-Uwe Linster

IMMD IV, Lehrstuhl für Betriebssysteme
Universität Erlangen-Nürnberg

linster@informatik.uni-erlangen.de

3.1.4 Selbstdefinierte Typen

▲ Typdefinition

```
TYPE [,attribut]... [::] typname
   komponentendefinition
   [komponentendefinition]
   .
END TYPE [typname]
```

Die Komponentendefinition entspricht weitgehend den Typvereinbarungsanweisungen; jedoch

- keine Initialisierung (es ist eine Definition; keine Deklaration!)
- als Attribute sind nur
  - DIMENSION (mit expliziter Gestalt) und
  - POINTER (Siehe Kapitel 1.5)

  zugelassen.

Selbstdefinierte Typen

Als Attribute der Typdefinition (dann gültig für alle Deklarationen dieses Typs) sind zulässig:

- PRIVATE (Siehe Kapitel 1.11)
- PUBLIC (Siehe Kapitel 1.11)
- SEQUENCE
  - erzwingt eine Speicherordnung gemäß der Komponentenreihenfolge; dies kann hinsichtlich der Speichernutzung Verschnitt bedeuten.
  - Verschiedene Typdefinitionen mit gleichen Komponenten (z. B. in verschiedenen Gültigkeitsbereichen)sind nur dann wirklich gleich, wenn sie das SEQUENCE-Attribut besitzen.
  - Wenn ein selbstdefinierter Typ das SEQUENCE-Attribut besitzt, dann müssen auch alle ggf. vorhandenen Typ-Komponenten, die ihrerseits selbstdefinierte Typen sind, das SEQUENCE-Attribut besitzen.
- Type - Vereinbarungsanweisung

```
Syntax:TYPE (TYPENAME) [,attribut]... [::] dg [,dg]...
```


Selbstdefinierte Typen

▲ Beispiele für Typdefinitionen

```
TYPE:: DATUM
  SEQUENCE
  INTEGER :: tag
  CHARCATER (LEN=3) :: monat
  INTEGER :: jahr
END TYPE DATUM
TYPE :: MITARBEITER
  CHARACTER *20 :: name, vorname
  TYPE (DATUM) geburtsdatum
  REAL gehalt
  REAL lohnsteuer
  LOGICAL verheiratet
  TYPE (MITARBEITER), POINTER :: alphanext ! (Siehe Kapitel 7)
END TYPE MITARBEITER
```

▲ Beispiel für Typvereinbarungen

```
TYPE (MITARBEITER), DIMENSION (100) :: gehaltsliste
TYPE (DATUM), DIMENSION (365) :: kalender
```

Selbstdefinierte Typen

- Strukturkonstante

```
Syntax: TYPNAME (komponentenliste)
  z. B. DATUM (1, 'Jan', 1996)
```

- Zuweisungen
  - Komponentenweise Zuweisung

```
variablenname%komponentenname = wert
```

  - Objektweise Zuweisung

```
variablenname = strukturkonstante
variablenname = variablenname     !gleichen Typs!
```


Selbstdefinierte Typen

- Beispiele

```
kalender(1)  = DATUM(1,'Jan',1996)
kalender(41)%tag = 10
kalender(41)%monat = 'Feb'
kalender(41)%Jahr = 1941

gehaltsliste(35) = MITARBEITER ('Mueller', 'Horst',&
     DATUM(10,'Feb',1941),4523.12,1212.00,.TRUE.,&
     gehaltsliste(25))

gehaltsliste(42)  = gehaltsliste(35)
gehaltsliste(42)%vorname = 'Anton'
gehaltsliste(42)%alphanext = gehaltsliste(35)
! Kapitel 1.5
```

Selbstdefinierte Typen

weitere Beispiele

- Typdefinition

```
TYPE :: ERSATZTEIL
  SEQUENCE
  CHARACTER *10 pkw_typ
  CHARACTER *10 part_nb
END ERSATZTEIL

TYPE :: PKW
  SEQUENCE
  CHARACTER *10 farbe
  INTEGER fahrgestellnummer
  INTEGER zulassungsjahr
  TYPE (ERSATZTEIL), DIMENSION (100) :: vorderachse
END TYPE PKW
```


Selbstdefinierte Typen

weitere Beispiele

- Typvereinbarungen

```
TYPE (PKW), DIMENSION (200) :: lager
```

- Zuweisungen

```
lager(110)%vorderachse(12)%part_nb = '1000G1019'
lager(1)%farbe = 'RAL312'
```

Selbstdefinierte Typen

```
PROGRAM personal
IMPLICIT NONE
INTEGER, PARAMETER :: max = 10
INTEGER, PARAMETER :: fd = 10
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
!  CHARACTER, DIMENSION (:), POINTER :: beurteilung! Kap. 1.5
END TYPE MITARBEITER

TYPE (MITARBEITER), DIMENSION (max) :: pa! Personal_akte
CHARACTER *4 funktion
CHARACTER zeichen
INTEGER i
LOGICAL :: ende = .FALSE.
```


Selbstdefinierte Typen

```
OPEN(fd,FORM='UNFORMATTED',file = 'database.personal')
   ! Oeffnen der Datei
DO WHILE (.NOT. ende)      !Funktionen

   PRINT *, ' '
   PRINT *, 'PRINT *, 'Eingabe der gewünschten Fkt:'; PRINT *,
' '
   PRINT *, 'init = Initialisiere Datenbasis'
   PRINT *, 'read = Lesen der Datenbasis'
   PRINT *, 'einz = Einzel-Eingabe'
   PRINT *, 'eing = Gruppeneingabe'
   PRINT *, 'korr = Korrektur;'
   PRINT *, 'ausg = Ausgabe;'
   PRINT *, 'sumg = Summe Gehaelter;'
   PRINT *, 'suml = Summe Lohnsteuer;'
   PRINT *, 'pers = Personalliste;'
   PRINT *, 'ende = Beenden des Programms'
   PRINT *, ' '
   PRINT *, 'Funktion?'
   READ *, funktion
   PRINT *, ' '
```

Selbstdefinierte Typen

```
   SELECT CASE (funktion)
      CASE ('init')
         PRINT *,'Sind Sie sicher? (y)'
         READ *,zeichen
         if (zeichen /= 'y') EXIT
         DO i = 1,max
               pa(i)%name = 'frei'
               pa(i)%vorname = 'frei'
         END DO
      CASE ('read')
         READ(fd) (pa(i), i=1,max)
      CASE ('einz')
         DO i = 1,max
               IF (pa(i)%name == 'frei') THEN
                     Print *,'Name:'
                     READ *, pa(i)%name
                     PRINT *, 'Vorname'
                     READ * , pa(i)%vorname
                     EXIT
               ELSE
                     CYCLE
               END IF
         END DO
```


Selbstdefinierte Typen

```
      CASE ('eing')
         PRINT *, 'Not yet implemented'
      CASE ('korr')
         PRINT *, 'Not yet implemented'
      CASE ('ausg')
         PRINT *, 'Not yet implemented'
      CASE ('sumg')
         PRINT *, 'Not yet implemented'
      CASE ('suml')
         PRINT *, 'Not yet implemented'
      CASE ('pers')
         DO i= 1,max
               PRINT *, pa(i)%name, ';', pa(i)%vorname
         END DO
      CASE ('ende')
         PRINT *, 'Programmende'
         WRITE(fd) (pa(i), i = 1,max)
         ende = .TRUE.
      CASE DEFAULT
         PRINT *, 'Falsche Eingabe!'
      END SELECT
END DO
END PROGRAM personal
```

3.1.5 Zeiger (Pointer)

- Pointer Definition
  - Der Inhalt einer Pointervariablen ist die Adresse eines Objekts auf das der Pointer zeigt.
  - Konsequenterweise müßten Pointer - wie in anderen Sprachen (z. B. in C) - ein spezieller Datentyp sein. Da man in Fortran90 nicht direkt auf einen Pointer zugreifen kann (keine Adreßarithmetik!), handelt es sich nur um ein Attribut eines Datentyps.
  - In Fortran90 finden Pointer hauptsächlich Verwendung bei der dynamischen Speicherverwaltung (während der Laufzeit des Programms; detailiert in Kapitel 1.8 dargestellt) und beim Aufbau von Datenstrukturen wie verketteten Listen, Baumstrukturen, etc.

Zeiger (Pointer)

- Pointer-Vereinbarung

```
REAL, POINTER :: zeiger, next, current !Typ und Attribut
! oder
REAL zeiger, next, current                  !Typ
Pointer zeiger, next, current               !Attribut
```

- Aus der Pointerdefinition folgt:
  - das Pointer-Attribut darf nur für Variablen oder Funktionen spezifiziert werden
  - folgende weitere Attribute sind unzulässig:
    ALLOCATABLE, EXTERNAL, INTENT, INTRINSIC, PARAMETER, TARGET
  - ein Pointer darf weder bei einer Typvereinbarung noch mit Hilfe einer DATA-Anweisung oder E/A initialisiert werden

Zeiger (Pointer)

Pointer-Zuweisung/Freigabe

- Will man einem Pointer eine Variable (oder Funktion) zuweisen, so muß für diese Datengröße das TARGET-Attribut spezifiziert sein.
  - Spezifikation des TARGET-Attributs

```
INTEGER, TARGET :: feld, person      ! Typ und Attribut
! oder
INTEGER feld, person    ! Typ
TARGET feld, person     ! Attribut
```

  - Zuweisung

```
pointer => zielvariable
! oder
ALLOCATE (pointer [,pointer]...) [, STAT=statusvariable]
```

  - Freigabe von Pointerzielen

```
pointer => neuezielvariable          !Überschreiben
! oder
DEALLOCATE (pointer [,pointer]...) [, STAT=statusvariable]
```


Zeiger (Pointer)

Pointer-Zuordnungsstati

- zugeordnet:
  - dem Pointer wurde eine Variable mit dem TARGET-Attribut zugewiesen.
  - dem Pointer wurde ein Pointer zugewiesen, der selbst den Status “zugeordnet” besitzt.
  - dem Pointer wurde mit der ALLOCATE-Anweisung ein Speicherbereich zugeordnet
- nicht zugeordnet:
  - dem Pointer wird mit der NULLIFY-Anweisung eine Zuordnung entzogen (Keine damit verbundene Speicherfreigabe!).
  - NULLIFY (pointer [,pointer]...)
  - dem Pointer wird mit der DEALLOCATE-Anweisung ein Speicherbereich entzogen, der ihm mit einer ALLOCATE-Anweisung zugeordnet wurde.
  - dem Pointer wird ein Pointer mit dem Status “nicht zugeordnet” zugewiesen.

Zeiger (Pointer)

Pointer-Zuordnungsstati(cont)

- undefiniert:
  - dem Pointer wurde noch kein Ziel zugewiesen (weder '=>' noch **ALLOCATE**)
  - das zugeordnete Ziel wurde auf "anderer Weise" entfernt;
    z. B. ein **DEALLOCATE** auf einen Zeiger mit dem gleichen Ziel
  - dem Pointer wird ein Pointer zugewiesen mit dem Staus "undefiniert"
  - wenn das Pointerziel bei Ausführung einer **END**- oder **RETURN**-Anweisung undefiniert wird. (Kapitel 1.6)

Zeiger (Pointer)

ASSOCIATED-Function

- Intrinsic (vordefinierte) Funktion

  `Syntax: ASSOCIATED (pointervar [,targetvar]`

  - liefert den Wert **.TRUE.** beim Aufruf

    `ASSOCIATED (pointervar)`

    wenn **pointervar** irgendeinem Ziel zugeordnet ist und beim Aufruf

    `ASSOCIATED (pointervar, targetvar)`

    wenn **pointervar targetvar** zugeordnet ist.
  - sonst den Wert **.FALSE.**

Beispiele

- zu Datenstrukturen, verkettete Listen, Baumstrukturen, etc
  siehe auch Vorl. mit Übungen : "Programm- und Datenstrukturen"
- zu **ALLOCATE, DEALLOCATE, NULLIFY**
  siehe auch Feldverarbeitung (Dynamische Speicherverwaltung) Kap. 1.8

Zeiger (Pointer)

```
PROGRAM queue

! Warteschlangen-Demo-Programm adaptiert aus:
! Cooper Redwine; "Upgrading to FORTRAN 90", Springer-Verlag,
! 1995
!
! Queue - Policy: first in - first out (Hier zunaechst nur
! Eingabe)

IMPLICIT NONE
TYPE :: QUEUE_ELEMENT
   CHARACTER (20) :: data
   TYPE (QUEUE_ELEMENT), POINTER :: next
END TYPE

! Zeiger auf des erste, letzte, aktuelle Element: first, last,
! this

TYPE (QUEUE_ELEMENT), POINTER :: first, last, this

CHARACTER(20) :: buffer
CHARACTER *4 :: funktion
LOGICAL :: ende = .FALSE.
INTEGER :: io_stat = 1
```


Zeiger (Pointer)

```
   DO WHILE (.NOT. ende)         !Funktionen
   PRINT *, ' '
   PRINT *, 'Eingabe der gewünschten Fkt:'
   PRINT *, ' '
   PRINT *, 'init = Initialisiere Queue'
   PRINT *, 'read = Lese Queue'
   PRINT *, 'erze = Erzeuge Elemente'
   PRINT *, 'dele = Loesche 1. Element'
   PRINT *, 'ende = Beenden des Programms'
   PRINT *, ' '
   PRINT *, 'Funktion:'
   READ *, funktion
   PRINT *, ' '
```

Zeiger (Pointer)

```
SELECT CASE (funktion)
   CASE ('init')
      NULLIFY (first, last)! Initialisiert: leere Queue
   CASE ('read')

      ! Durchlaufe die Warteschlange und drucke Inhalte als
      ! Liste aus.

      this => first           ! aktuelles Element gleich
                              ! 1. Element
      PRINT *, 'Ausgabe'; PRINT *, ' '
      DO WHILE (ASSOCIATED (this))
            PRINT *, this%data
            this => this%next ! aktuelles Element gleich
                              ! naechstes Element
      END DO
```


Zeiger (Pointer)

```
CASE ('erze')
   PRINT *, 'Eingabe:'
   DO WHILE (io_stat >= 0)
      READ (*, "(A)", IOSTAT = io_stat) buffer

      IF (.NOT. ASSOCIATED (first)) THEN

      ! Queue ist leer; fordere den Speicher fuer das
      ! 1. Element an

            ALLOCATE (first)
            last => first     ! Beide Zeiger zeigen aufs
                              ! gleiche(erste)Element
      ELSE
      ! Queue ist nicht leer, neues Element gleich
      ! letztes Element

            ALLOCATE (last%next)
            last => last%next
      END IF
      last%data = buffer
      NULLIFY (last%next)     ! Letztes Element hat
                              ! keinen Nachfolger
   END DO
```

Zeiger (Pointer)

```
      CASE ('dele')
         PRINT *, 'Not yet implemented'
      CASE ('ende')
         PRINT *, 'Programmende'
         ende = .TRUE.
      CASE DEFAULT
         PRINT *, 'Falsche Eingabe!'
      END SELECT
END DO

END PROGRAM queue
```


3.1.6 Unterprogramme

- Definitionen/Terminologie
  - Programmeinheiten (programunits) sind die Bausteine eines ausführbaren Programms
    - Hauptprogramm (genau einmal)
    - ext. Unterprogramme
    - Module
    - Blockdatenprogrammeinheiten
  - Sie enden mit der END-Anweisung, liegen in separaten Dateien, sind getrennt übersetzbar, jedoch bis auf das Hauptprogramm nicht direkt ausführbar.
  - Ein Hauptprogramm ist ausführbar, wenn es keine offenen ’Externreferenzen’ gibt - ausgenommen Bibliotheksreferenzen, wie z.b. SIN(x), COS(x), etc.
    Diese Eigenschaft haben unsere bisher geschriebenen Programme.

Unterprogramme

- Die weiteren Programmeinheiten sind nur dann ausführbar (getrennt testbar), wenn ein entsprechendes Hauptprogramm als “Testrahmen” zugebunden wird.
- Nach der Übersetzung sind diese Einheiten zunächst separate Bindemodule; sogenannte ".o-files"

■ Unterprogramme werden unterschieden

- nach ihrem “Definitionsort”
  - extern
  - eingebettet
  - vordefiniert in Bibliothek
- und ihrer Art
  - Subroutine
  - Funktion
  - Formelfunktion

Unterprogramme

■ Subroutine

- Subroutinendefinition

```
[...] SUBROUTINE sname [(...)]        !(SUBROUTINE-Anweisung)
    [Spezifikationsteil]
    ...
        Subroutinenrumpf
    ...
[CONTAINS]
    [interne Unterprogramme]
    ...
END [SUBROUTINE [sname]]
```

- Beispiele:

```
SUBROUTINE sname()                       !(leere Parameterliste)
SUBROUTINE sname (formalparameter[,formalparameter] ...)
```

- Subroutinenaufruf

```
CALL sname
CALL sname()
CALL sname(Aktualparameter[,Aktualparameter]...)
```

Unterprogramme

- Beispiel - eingebettetes Unterprogramm (CONTAINS)

```
PROGRAM summentest
IMPLICIT NONE
REAL a, b, ergebnis

   Print *, 'Eingabe: summand1 summand2 <r>'
   READ *, a,b

   call summe (a,b,ergebnis)

   PRINT *, 'Ergebnis = ',ergebnis

CONTAINS

      SUBROUTINE summe (summanda, summandb, erg)
      REAL summanda, summandb, erg
         erg = summanda + summandb
      END SUBROUTINE summe

END PROGRAM summentest
```


Unterprogramme

- Geltungsbereich/Gültigkeitsbereich
  - sagt aus, in welchen Bereichen eine Variable zugreifbar ist
    - lokal Variable sind nur gültig innerhalb eines Geltungsbereichs z. B. in einem Unterprogramm
    - globale Variable sind in der gesamten Programmeinheit, in der sie definiert wurden, zugreifbar, falls sie nicht durch eine lokale Definition “verdeckt” werden.
      (“Blickt” man aus der Aufrufschachtelung zurück in Richtung Aufrufer, so sind sämtliche globale Daten der aufrufenden Ebenen zugreifbar, es sei denn, sie sind durch lokale Vereinbarungen verdeckt.)

Unterprogramme

- Geltungsbereich/Gültigkeitsbereich(cont)
  - Transfer von Informationen zwischen Geltungsbereichen durch:
    - Parameter
      - Eingangsparameter
      - Ausgangsparameter; Ergebnisparameter
      - beide Richtungen
    - globale Daten
      - im äußersten Geltungsbereich definiert; also programmeinheitsglobal
      - COMMON-Bereiche
        Überall dort gültig, wo definiert; auch über Programmeinheiten hinweg

Unterprogramme

- Parameter
  - Formalparameter (bei der Definition)
  - Aktualparameter (beim Aufruf)

    Zum Zeitpunkt des Aufrufs werden die Formalparameter und die Aktualparameter einander zugeordnet, d.h. eine “Formal-Variable” im Unterprogramm referenziert den gleichen Speicher wie die “Aktual-Variable” im aufrufenden Programm.

  Man unterscheidet
    - positionsgerechte Schreibweise
    - Schlüsselwortschreibweise z. B.

      formalparameter = aktualparameter

    Die gemischte Schreibweise ist möglich. Beginnt man jedoch mit der Schlüsselwortschreibweise, so müssen auch alle folgenden Parameter in Schlüsselwortschreibweise folgen.

Unterprogramme

INTENT-Attribut/Anweisung

- Mit dem INTENT-Attribut kann die Art der Verwendung von Formalparametern spezifiziert werden:
  - Das INTENT(IN)-Attribut besagt, daß der Formalparameter ein Eingabeparameter ist. D.h. dieser Formalparameter darf innerhalb des Unterprogramms nicht redefiniert werden, er darf also z. B. nicht auf der linken Seite einer Zuweisung auftauchen oder Variable einer READ-Anweisung sein.
    Der Aktualparameter kann also sowohl eine Variable als auch eine Konstante sein.
  - Das INTENT(OUT)-Attribut besagt, daß der Formalparameter ein Ausgabeparameter ist. D. h. der Formalparameter muß erst definiert (ein Wert zugewiesen) werden, ehe im Unterprogramm auf ihn zugegriffen wird. Hier **muß** es sich also um eine Variable handeln.

Unterprogramme

INTENT-Attribut/Anweisung

- Mit dem INTENT-Attribut kann die Art der Verwendung von Formalparametern spezifiziert werden (cont):
  - Das INTENT(INOUT)-Attribut besagt, daß der Parameter sowohl Eingabe- als auch Ausgabeparameter sein soll. Auch hier **muß** es sich also um eine Variable handeln.

Wir werden im Rahmen dieses Kurses die Formalparameter mit INTENT-Attributen versehen denn: Keine Angabe = INTENT(INOUT), es sei denn, der Aktual-Parameter ist eine Konstante.

© Claus-Uwe Linster – pps3.1-2_spmd_f90: 19/3/98

Unterprogramme

- Beispiel:

```
PROGRAM summentest
IMPLICIT NONE
REAL a, b, ergebnis

   Print *, 'Eingabe: summand1 summand2 <r>'
   READ *, a,b

   call summe (a,b,ergebnis)

   PRINT *, 'Ergebnis = ',ergebnis

CONTAINS

      SUBROUTINE summe (summanda, summandb, erg)
      REAL, INTENT(IN) :: summanda, summandb
      REAL, INTENT(out) :: erg
         erg = summanda + summandb
      END SUBROUTINE summe

END PROGRAM summentest
```


Unterprogramme

- FUNKTION

```
[...] FUNCTION fname [(...)]!(FUNCTION-Anweisung)
   [Spezifikationsteil]
   ...
      Funktionsrumpf
      ...
[CONTAINS]
   [interne Unterprogramme]
   ...
END [FUNCTION [fname]]
```

- der wesentliche Unterschied einer Funktion zu einer Subroutine ist, daß der Funktionsname gleichzeitig auch der Name der (lokalen) Ergebnisvariablen ist. Deshalb hat die Funktion einen Typ!
- Der Aufruf einer Funktion kann überall dort stehen, wo auch eine Variable stehen darf; nur auf der linken Seite einer Zuweisung (natürlich) nicht.
- Der Funktionsaufruf darf auch ein separates Statement sein - analog zur Subroutine - allerdings ohne “CALL”; jedoch mit “()”.

Unterprogramme

```
( ~uelinste/tmp/newton_fkt.f90 )

PROGRAM newton
REAL eingabe
! Author: Uwe Linster
! Anfordern und einlesen der Zahl, deren Quadratwurzel
! berechnet werden soll

   PRINT *, ' '

   ! Ausgabe auf das Sichtgerät ohne Zeilenvorschub
   ! Nichtvorrueckende Ausgabe: Kapitel 1.10

   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Zu berechnen ist die Quadratwurzel aus: ')

   READ *, eingabe
   PRINT *, ' '; PRINT *, ' '

   DO WHILE(eingabe <= 0)
      PRINT *, ' Die Eingabedaten entsprechen nicht den &
                &Anforderungen des Programms (Zahl > 0!):'
      READ *, eingabe
   END DO
```


Unterprogramme

```
!Berechnung

   ergebnis = sqroot(eingabe)

!Ergebnisausgabe

   PRINT *, ' Die Quadratwurzel aus ', eingabe ;PRINT *, ' '
   PRINT *, ' ist ', ergebnis
! Oder auch gleich:
!  PRINT *, ' ist ', sqroot(eingabe)

PRINT *, ' '; PRINT *, ' Ende des Programms'
```

Unterprogramme

```
CONTAINS

REAL FUNCTION sqroot (a)
!Berechnung der Quadratwurzel nach dem Newton'schen Verfahren
!Eingabeparameter:a; Startwert der Iteration: 1
!Iterationsformel:x(i+1) = (x(i)*x(i) + a) / (2*x(i))

   REAL , INTENT(IN) :: a
   REAL x
      !a Größe, deren Quadratwurzel zu berechnen ist
      !x Berechnete Approximation der Quadratwurzel

   !Anfangsapproximation
   x = 1
   !Iteration
   DO WHILE (.NOT. (ABS(x*x - a) <= 1E-4))
      x = (x*x + a) / (2.*x)
   END DO
   sqroot = x
END FUNCTION sqroot

END PROGRAM newton
```


Unterprogramme

- COMMON-BEREICHE
  - Durch COMMON-Bereiche werden globale Speicherbereiche spezifiziert. Variable sind in den Geltungsbereichen direkt zugreifbar, in denen die COMMON-Bereiche spezifiziert wurden; auch über verschiedene Programmeinheiten hinweg.
    Man unterscheidet:
    - einen unbenannten COMMON-Block

```
Syntax:COMMON variablenliste
```

    - und beliebig viele benannte COMMON-Blöcke

```
Syntax:COMMON /blockname/ variablenliste [, /blockname/]
              variablenliste]...
```

    - auch die Mischschreibweise ist möglich:

```
Syntax:COMMON  variablenliste [[,]/[blockname]/
              variablenliste]...
```

Unterprogramme

- COMMON-BEREICHE(cont)
  - Eigenschaften
    - Durch Angabe der Variablen wird eine Speicherfolge festgelegt; um Fehler zu vermeiden sollte man mit “copy” und “paste” arbeiten.
    - Variablennamen in verschiedenen Geltungseinheiten dürfen verschieden sein; das sollte man jedoch nur tun, wenn es dafür auch einen plausiblen Grund gibt. Man greift auf dieselbe globale Speicheradresse zu, das sollte man durch den gleiche Variablennamen auch dokumentieren!
    - Initialisiert werden COMMON-Bereiche mit einer BLOCK DATA-Programmeinheit; jedoch nur die benannten Blöcke.
    - Selbstdefinierte Datentypen müssen das Attribut SEQUENCE besitzen. (Speicherfolge!)

Unterprogramme

- COMMON-Bereich - Beispiel (~uelinste/tmp/common_demo_bubble.f90

```
PROGRAM bubble
IMPLICIT NONE

! Zweck des Programms, Loesungsverfahren:
!
! Die Eintraege in der Datei titel sollen in lexikographisch
! sortierter Reihenfolge gedruckt werden. Das Programm geht
! davon aus, dass die Datei maximal 200 Eintraege besitzt und
! jeder Eintrag aus hoechstens 20 Zeichen besteht.
! Als Sortierverfahren wird 'bubble sort' verwendet.
!-------------------------------------------------------------
! Globale Variable; in COMMON-Bereiche spezifiziert
!    name Feld zur Aufnahme der Eintraege aus der Dateri titel
!    max Anzahl der Eintraege
!  fd Kanalnummer (Filedescriptor; Unit)
!-------------------------------------------------------------
! Initialisierung
!
! Ausgabe der Gesamtueberschrift
!
PRINT *, 'Eintraege der Datei titel lexikograpgisch sortiert:'
```

Unterprogramme

```
! Oeffnen der Datei
!
   CALL oeffnen_datei()

! Einlesen der Daten
!
   CALL einlesen()
!
! Sortieren
!
   CALL bubblesort()
!
! Ausgabe des Ergebnisses
!
   CALL ausgeben_screen()
   CALL ausgeben_datei()
```


Unterprogramme

```
! Die Unterprogramme:

CONTAINS

SUBROUTINE oeffnen_datei()

! Oeffnen der Datei "titel" mit Kanalnummer 10
! Global Data:

INTEGER fd
COMMON /datei/ fd
   fd = 10     ! Entfaelt bei BLOCK-DATA-INIT!!
! Local DATA: none

   open(fd, FORM = 'FORMATTED', file = 'titel')

END SUBROUTINE oeffnen_datei
```

Unterprogramme

```
SUBROUTINE einlesen()

! Einlesen der Eintraege aus der Datei "titel" in das Feld !
"name".
! Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

!Local Data:
INTEGER i

   DO i = 1, 200
      READ(fd, 1000, END = 1010) name(i)
1000  FORMAT (a)
   END DO

1010max = i - 1
   PRINT *, 'Gelesene Datensaetze: ',max

END SUBROUTINE einlesen
```


Unterprogramme

```
SUBROUTINE bubblesort()
! Sortiert nach dem "bubble-sort"-Verfahren
! Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

!Local Data

CHARACTER *20 :: hilf
LOGICAL flag
INTEGER i, j, dl

!  flag   zeigt an, ob bei einem Durchlauf Aenderungen in der
!              Anordnung vorgenommen wurden
!              (.FALSE.: Keine Umordnungen
!               .TRUE. : Es wurden Umordnungen vorgenommen)
!  hilf   Hilfsfeld waehrend des Vertauschens zweier Eintraege
!  dl Anzahl der Sortierschleifen
!
```

Unterprogramme

```
! Der Sortiervorgang

   dl = 0
   DO  i = max, 2, -1
      flag = .FALSE.

      DO j = 1, i - 1
          IF (name(j) > name(j + 1)) THEN
               hilf = name(j)
               name(j) = name(j + 1)
               name(j + 1) = hilf
               flag = .TRUE.
               dl = dl + 1
          END IF
        END DO

        if (.NOT. flag) THEN
         PRINT *, 'Anzahl der Durchlaeufe: ', dl
         EXIT
      END IF

   END DO

END SUBROUTINE bubblesort
```


Unterprogramme

```
SUBROUTINE ausgeben_screen()

!Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max
COMMON name, max

! Local Data:

INTEGER i

   DO i=1,max
      Print *, name(i)
   END DO

END SUBROUTINE ausgeben_screen
```

Unterprogramme

```
SUBROUTINE ausgeben_datei()

!Global Data

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

! Local Data

INTEGER i

   REWIND (fd)
   WRITE ( fd, 1030 )(name(i), i=1, max)
1030FORMAT (a)

END SUBROUTINE ausgeben_datei

END PROGRAM bubble
```


Unterprogramme

- BLOCK DATA
  - Eine BLOCK-DATA-Programmeinheit
    - ist eine nichtausführbare Programmeinheit, sie dient der Initialisierung von Variablen in **benannten** COMMON-Bereichen.
    - COMMON-Blöcke sind globale Speicherbereiche - gültig auch über Programm-Unit-Grenzen hinweg. Eine Initialisierung darf also Programmsystemweit nur einmal erfolgen. Dies könnte innerhalb einer “ausgezeichneten” Programmeinheit erfolgen oder wie hier in einer speziellen Programmeinheit

```
BLOCKDATA [name]
     [Spezifikationsteil]
     .
     .
     .
  END BLOCKDATA [name]
```

Unterprogramme

- BLOCK DATA(cont)

```
Beispiel
BLOCKDATA car

CHARACTER vorderachse *14, hinterachse *10, motor *11
INTEGER anzahl, seriennummer, partnb

COMMON /ersatzteile/ vorderachse, hinterachse, motor
COMMON /attribute/ anzahl, seriennummer, partnb

DATA vorderachse, hinterachse /'Schwingachsen', `Inkaachsen'/
DATA motor /'Dieselmotor'/
DATA anzahl, partnb /250, 4711/ ! seriennummer bleibt
uninitialisiert

END BLOCKDATA car
```

- Da diese Programm-Unit in einer eigenen Datei liegt muß beim Aufruf des Compilers diese Datei als weiterer Parameter angegeben werden:; z. B.

```
f90 hauptprogramm.f90 blockdata.f90 -o programm.out
```


Unterprogramme

- BLOCK DATA
  - Beispiel für Programm “bubble”:

Datei: bubble_blockdata.f90

```
BLOCKDATA bubble

   INTEGER fd

   COMMON /datei/ fd

   DATA fd /10/

END BLOCKDATA bubble
```

- Aufruf des Compilers:

```
f90 common_demo_bubble.f90 bubble_blockdata.f90 -o bubble.out
```

Unterprogramme

FORMELFUNKTION

- eine Formelfunktion ist eine Funktion, die nur aus einer einzigen Anweisung besteht und
- sie ist nur innerhalb einer Geltungseinheit aufrufbar.
- Formelfunktionsdefinition:

```
Syntax:ffname() = ausdruck oder
ffname(formalparameter[,formalparameter]...) =
                              ausdruck
```

- Sowohl 'ausdruck' als auch 'formalparameter' sind skalare Datenobjekte.
- Formelfunktionsaufruf:

```
Syntax:ffname() oder
ffname(formalparameter[,formalparameter]...)
```

- Der Typ der Formelfunktion wird separat (wie bei der Typvereinbarung einer Variablen) vereinbart; dieses gilt auch für die Formalparameter.

Unterprogramme

- Formelfunktion - Beispiel (~uelinste/tmp/capslock_fkt.f90)

```
PROGRAM capslock
IMPLICIT NONE
!
! Das Programm liest Kleinbuchstaben ein, wandelt sie
! in Grossbuchstaben um und druckt diese aus.
!  Variable
!
      CHARACTER, DIMENSION(20) :: zeichenin, zeichenout
      INTEGER i

   CHARACTER gross, klein

!  Formelfunktion:

   gross(klein) = CHAR ( ICHAR(klein) + ICHAR('A') - ICHAR('a'))
```

Unterprogramme

```
!  Einlesen der Zeichen

      PRINT *, ' '
   PRINT *, 'Kleinbuchstaben; max. 20; entweder durch "blank"
getrennt -'
   PRINT *, 'Abschluss <r> oder einzeln durch <r> getrennt!'
   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Eingabe: ')
      READ *, (zeichenin(i), i=1,20)

   DO i = 1,20
      zeichenout(i) = gross(zeichenin(i))
   END DO

   PRINT *, 'Ausgabe:',zeichenout

END PROGRAM capslock
```


Unterprogramme

- Formelfunktion - Beispiel (~uelinste/tmp/kredit_fkt.f90)

```
PROGRAM kredit
IMPLICIT NONE
!
!  Das Programm berechnet die Hoehe der monatlichen Raten fuer
!  einen vorgebbaren Kaufpreis bei einzugebender Laufzeit. Raten
!  und Zinsen werden von zwei Jahren bis zu einer angebbaren
!  maximalen Laufzeit in Jahresschritten berechnet. Untere und
!  oebere Grenze des Zinssatzes werden ebenfalls eingelesen.
! ------------------------------------------------------------
!  Variable
!
      REAL preis, mrate, p
      INTEGER n, zmin, zmax, zsatz, zs, year, yearmax
!
! Berechnung der Monatsrate fuer die verschiedenen Zinssaetze
!          formel: r*p/(1-(1+r)**(-n)) mit r = zinssatz/12/100
!  hier als Formelfunktion realisiert:

   mrate(zs,p,n) = ((zs/100./12.) * p) / &
               (1.-(1. + (zs/100./12.))**(-n))
```

Unterprogramme

```
!        preis      Kaufpreis (Eingabe)
!       yearmax   maximale Laufzeit der Ratenzahlung (Eingabe)
!        zmin       minimaler Zinssatz (Eingabe)
!        zmax       maximaler Zinssatz (Eingabe)
!        year       Laufvariable fuer die Laufzeit
!        n          Zahl der Raten
!        zsatz      Laufvariable fuer den Zinssatz
! ------------------------------------------------------------
! Initialisierung
!
!  Einlesen von Kaufpreis, maximaler Laufzeit,
!  minimalem und maximalem Zinssatz
!
      PRINT *, ' '
   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Kaufpreis (REAL): ')
      READ *, preis
   WRITE (*, 2000, ADVANCE='NO')
2000FORMAT ('Laufzeit in Jahren (groesser als 2): ')
      READ *, yearmax
     WRITE (*, 3000, ADVANCE='NO')
3000FORMAT ('minimaler und maximaler Zins (in Prozent
(INTEGER)): ')
      READ *, zmin, zmax; PRINT *, ' '
```


Unterprogramme

```
! Aeussere Schleife

    DO year = 2, yearmax

!  Tabellenueberschrift
        PRINT *, ' '
        PRINT *, ' '
      PRINT *, ' Zinsen fuer eine Laufzeit von ', year, ' Jahren'
        PRINT *, ' '
       PRINT *, ' Zins-  Zahl der    Rate         Gesamtzins'
        PRINT *, ' satz    Raten       (DM)              (DM)'

! Innere Schleife
      DO zsatz = zmin, zmax
         n = 12.*year
          PRINT *, ' ', zsatz, '     ', n, ' ', &
               mrate (zsatz, preis, n) &
               ,' ', n * (mrate (zsatz, preis, n)) -preis
      END DO
    END DO
!
!        Abschlussausgabe
      PRINT *, ' ' ;PRINT *, ' Berechnung abgeschlossen.'
END PROGRAM kredit
```

Unterprogramme

- Rücksprung
  - Es existieren drei alternative Methoden
    - der “normale” Rücksprung bei Erreichung des END-Statements
    - der “Ausnahme” - Rücksprung, wenn aufgrund einer Bedingung die Aufgabe des Unterprogramms erfüllt ist - etwa analog zum EXIT-Aussprung aus einer DO-Schleife

```
Syntax:RETURN
```

Bei beiden Methoden wird mit dem 1. ausführbaren Statement fortgefahren, das dem Unterprogrammaufruf folgt.

- der “Ausnahme” - Rücksprung mit wählbarer Fortsetzungsadresse (label).

```
Syntax:RETURN ganzzahliger_ausdruck
```

- Diese Möglichkeit wird nur der Vollständigkeit erwähnt (Im Falle, daß man mit vorhandenen Programmsystemen umgehen muß). Die Verwendung solcher Konstrukte führt zu undurchsichtigen, nicht mehr wartbaren Programmsystemen.

Unterprogramme

- SAVE-Anweisung
  - Mit dem SAVE-Attribut wird der Wert einer **lokalen** Variablen in einem Geltungsbereich über die “Lebensdauer” eines Aufrufs hinaus “gesichert”. D. h., daß der Wert der Variablen auch beim nächsten Aufruf wieder verfügbar ist. Man spricht auch von “*Static*” - Variablen.
  - Das SAVE-Attribut ist nur erlaubt bei
    - lokalen Variablen
    - benannten COMMON-Blöcken
      (hier ist Vorsicht geboten, ist doch ein COMMON-Block ein globaler Speicherbereich!?!)
  - Das SAVE-Attribut ist insbesondere nicht erlaubt bei
    - einzelnen Variablen, die in einem COMMON-Bereich enthalten sind
    - Formalparametern
    - Funktionsvariablen
    - Namensgruppen (Siehe Kapitel 1.10)

Unterprogramme

SAVE-Anweisung

Attributspezifikation

- gleichzeitig mit der Typvereinbarung

```
REAL, SAVE :: skonto, summe
INTEGER, SAVE :: max
```

- oder getrennt mit einer SAVE-Anweisung

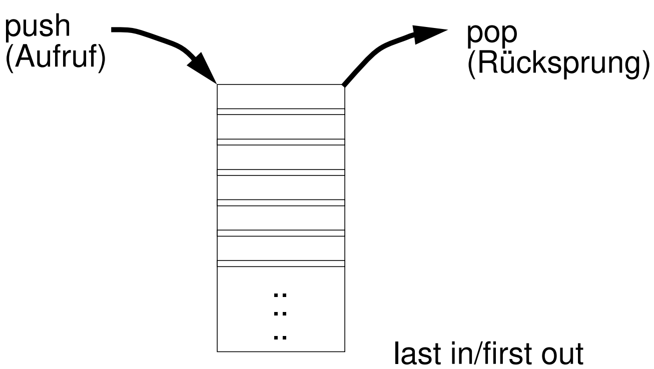
```
REAL skonto, summe
SAVE skonto, summe

INTEGER max
SAVE max
```

Eine SAVE-Anweisung ohne Angabe von Variablennamen bedeutet, das sämtliche lokale Variable in diesem Geltungsbereich diese Spezifikation erhalten.

Rekursive Unterprogramme

Unterprogramme (SUBROUTINE oder FUNCTION) sind dann rekursiv, wenn sie sich selbst direkt oder indirekt aufrufen.

Wenn Unterprogramme das Attribut RECURSION besitzen, muß der Compiler dafür Sorge tragen, daß jeweils bei einem rekursiven Aufruf die lokalen Variablen gerettet und nach dem Rücksprung aus einer tieferen Aufrufebene wieder bereitgestellt werden. Dies ist eine typische Anwendung für einen Kellerspeicher (Stack)!



last in/first out

Rekursive Unterprogramme

- Wann verwendet man rekursive Unterprogramme?
  - Es gibt eine Anzahl von Algorithmen, die "in sich" rekursiv definiert sind. Bei der iterativen Implementierung solcher Algorithmen ist der Programmieraufwand in der Regel beträchtlich und mit einer großen Anzahl von Abfragen behaftet, wie einige Beispiele des Kapitels noch zeigen werden.
  - Als einfaches Beispiel - zum Verständnis - wurde die Berechnung von Fakultäten gewählt, tatsächlich ist es in diesem Fall allerdings effizienter, eine Schleife zu implementieren!

```
PROGRAM recursion
IMPLICIT NONE
INTEGER n, erg
   WRITE (*, FMT = "('n! ; Welches n? : ')", ADVANCE = 'NO')
   READ *, n
   CALL fak (n, erg)
   PRINT *, 'Fakultaet von: ',n ,' = ', erg
   erg = fakfunc (n)
   PRINT *, 'Fakultaet von: ',n ,' = ', erg
CONTAINS
```


Rekursive Unterprogramme

- Rekursive Subroutine

```
RECURSIVE SUBROUTINE fak (fn, ferg)

INTEGER, INTENT (IN) :: fn
INTEGER, INTENT (OUT) :: ferg

   IF (fn == 0) THEN
         ferg = 1
   ELSE
         CALL fak (fn - 1, ferg)
         IF (ferg < 0) THEN
               PRINT *, 'underflow'
               STOP
         END IF
         ferg = fn * ferg
   END IF

END SUBROUTINE fak
```

Rekursive Unterprogramme

▲ Rekursive Funktion

```
RECURSIVE INTEGER FUNCTION fakfunc (fn) RESULT (ferg)

INTEGER, INTENT (IN) :: fn

   IF (fn == 0) THEN
         ferg = 1
   ELSE
         ferg = fn * fakfunc (fn - 1)
         IF (ferg < 0) THEN
               PRINT *, 'underflow'
               STOP
         END IF
   END IF

END FUNCTION fakfunc

END PROGRAM recursion
```


Unterprogramme

■ (Noch) nicht behandelt

- Parameterübergabe bei nicht skalaren Datengrößen (Siehe Kapitel 1.7- Feldverarbeitung)
- Module (Siehe Kapitel 1.11 - Module)
- Externe Unterprogramme (Siehe Kapitel n.b.)
- Nebeneinganspunkte - ENTRY-Anweisungen (Siehe Kapitel n.b.)
- Optionale Formalparameter (Siehe Kapitel n.b.)

Programmierung Paralleler Systeme

3. Programmiermodell “Single Programm Multiple Data”

3.1 Fortran95

3.1 .4 Selbstdef. Datentypen (Strukturen)

3.1.5 Zeiger(Pointer)

3.1.6 Unterprogramme

Claus-Uwe Linster

IMMD IV, Lehrstuhl für Betriebssysteme
Universität Erlangen-Nürnberg

linster@informatik.uni-erlangen.de

3.1.4 Selbstdefinierte Typen

▲ Typdefinition

```
TYPE [,attribut]... [::] typname
   komponentendefinition
   [komponentendefinition]
   .
END TYPE [typname]
```

Die Komponentendefinition entspricht weitgehend den Typvereinbarungsanweisungen; jedoch

- keine Initialisierung (es ist eine Definition; keine Deklaration!)
- als Attribute sind nur
  - DIMENSION (mit expliziter Gestalt) und
  - POINTER (Siehe Kapitel 1.5)

  zugelassen.

Selbstdefinierte Typen

Als Attribute der Typdefinition (dann gültig für alle Deklarationen dieses Typs) sind zulässig:

- PRIVATE (Siehe Kapitel 1.11)
- PUBLIC (Siehe Kapitel 1.11)
- SEQUENCE
  - erzwingt eine Speicherordnung gemäß der Komponentenreihenfolge; dies kann hinsichtlich der Speichernutzung Verschnitt bedeuten.
  - Verschiedene Typdefinitionen mit gleichen Komponenten (z. B. in verschiedenen Gültigkeitsbereichen)sind nur dann wirklich gleich, wenn sie das SEQUENCE-Attribut besitzen.
  - Wenn ein selbstdefinierter Typ das SEQUENCE-Attribut besitzt, dann müssen auch alle ggf. vorhandenen Typ-Komponenten, die ihrerseits selbstdefinierte Typen sind, das SEQUENCE-Attribut besitzen.
- Type - Vereinbarungsanweisung

```
Syntax:TYPE (TYPENAME) [,attribut]... [::] dg [,dg]...
```


Selbstdefinierte Typen

▲ Beispiele für Typdefinitionen

```
TYPE:: DATUM
  SEQUENCE
  INTEGER :: tag
  CHARCATER (LEN=3) :: monat
  INTEGER :: jahr
END TYPE DATUM
TYPE :: MITARBEITER
  CHARACTER *20 :: name, vorname
  TYPE (DATUM) geburtsdatum
  REAL gehalt
  REAL lohnsteuer
  LOGICAL verheiratet
  TYPE (MITARBEITER), POINTER :: alphanext ! (Siehe Kapitel 7)
END TYPE MITARBEITER
```

▲ Beispiel für Typvereinbarungen

```
TYPE (MITARBEITER), DIMENSION (100) :: gehaltsliste
TYPE (DATUM), DIMENSION (365) :: kalender
```

Selbstdefinierte Typen

- Strukturkonstante

```
Syntax: TYPNAME (komponentenliste)
  z. B. DATUM (1, 'Jan', 1996)
```

- Zuweisungen
  - Komponentenweise Zuweisung

```
variablenname%komponentenname = wert
```

  - Objektweise Zuweisung

```
variablenname = strukturkonstante
variablenname = variablenname     !gleichen Typs!
```


Selbstdefinierte Typen

- Beispiele

```
kalender(1)  = DATUM(1,'Jan',1996)
kalender(41)%tag = 10
kalender(41)%monat = 'Feb'
kalender(41)%Jahr = 1941

gehaltsliste(35) = MITARBEITER ('Mueller', 'Horst',&
     DATUM(10,'Feb',1941),4523.12,1212.00,.TRUE.,&
     gehaltsliste(25))

gehaltsliste(42)  = gehaltsliste(35)
gehaltsliste(42)%vorname = 'Anton'
gehaltsliste(42)%alphanext = gehaltsliste(35)
! Kapitel 1.5
```

Selbstdefinierte Typen

weitere Beispiele

- Typdefinition

```
TYPE :: ERSATZTEIL
  SEQUENCE
  CHARACTER *10 pkw_typ
  CHARACTER *10 part_nb
END ERSATZTEIL

TYPE :: PKW
  SEQUENCE
  CHARACTER *10 farbe
  INTEGER fahrgestellnummer
  INTEGER zulassungsjahr
  TYPE (ERSATZTEIL), DIMENSION (100) :: vorderachse
END TYPE PKW
```


Selbstdefinierte Typen

weitere Beispiele

- Typvereinbarungen

```
TYPE (PKW), DIMENSION (200) :: lager
```

- Zuweisungen

```
lager(110)%vorderachse(12)%part_nb = '1000G1019'
lager(1)%farbe = 'RAL312'
```

Selbstdefinierte Typen

```
PROGRAM personal
IMPLICIT NONE
INTEGER, PARAMETER :: max = 10
INTEGER, PARAMETER :: fd = 10
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
!  CHARACTER, DIMENSION (:), POINTER :: beurteilung! Kap. 1.5
END TYPE MITARBEITER

TYPE (MITARBEITER), DIMENSION (max) :: pa! Personal_akte
CHARACTER *4 funktion
CHARACTER zeichen
INTEGER i
LOGICAL :: ende = .FALSE.
```


Selbstdefinierte Typen

```
OPEN(fd,FORM='UNFORMATTED',file = 'database.personal')
   ! Oeffnen der Datei
DO WHILE (.NOT. ende)      !Funktionen

   PRINT *, ' '
   PRINT *, 'PRINT *, 'Eingabe der gewünschten Fkt:'; PRINT *,
' '
   PRINT *, 'init = Initialisiere Datenbasis'
   PRINT *, 'read = Lesen der Datenbasis'
   PRINT *, 'einz = Einzel-Eingabe'
   PRINT *, 'eing = Gruppeneingabe'
   PRINT *, 'korr = Korrektur;'
   PRINT *, 'ausg = Ausgabe;'
   PRINT *, 'sumg = Summe Gehaelter;'
   PRINT *, 'suml = Summe Lohnsteuer;'
   PRINT *, 'pers = Personalliste;'
   PRINT *, 'ende = Beenden des Programms'
   PRINT *, ' '
   PRINT *, 'Funktion?'
   READ *, funktion
   PRINT *, ' '
```

Selbstdefinierte Typen

```
SELECT CASE (funktion)
   CASE ('init')
      PRINT *,'Sind Sie sicher? (y)'
      READ *,zeichen
      if (zeichen /= 'y') EXIT
      DO i = 1,max
            pa(i)%name = 'frei'
            pa(i)%vorname = 'frei'
      END DO
   CASE ('read')
      READ(fd) (pa(i), i=1,max)
   CASE ('einz')
      DO i = 1,max
            IF (pa(i)%name == 'frei') THEN
                  Print *,'Name:'
                  READ *, pa(i)%name
                  PRINT *, 'Vorname'
                  READ * , pa(i)%vorname
                  EXIT
            ELSE
                  CYCLE
            END IF
      END DO
```


Selbstdefinierte Typen

```
   CASE ('eing')
      PRINT *, 'Not yet implemented'
   CASE ('korr')
      PRINT *, 'Not yet implemented'
   CASE ('ausg')
      PRINT *, 'Not yet implemented'
   CASE ('sumg')
      PRINT *, 'Not yet implemented'
   CASE ('suml')
      PRINT *, 'Not yet implemented'
   CASE ('pers')
      DO i= 1,max
            PRINT *, pa(i)%name, ';', pa(i)%vorname
      END DO
   CASE ('ende')
      PRINT *, 'Programmende'
      WRITE(fd) (pa(i), i = 1,max)
      ende = .TRUE.
   CASE DEFAULT
      PRINT *, 'Falsche Eingabe!'
   END SELECT
END DO
END PROGRAM personal
```

3.1.5 Zeiger (Pointer)

- Pointer Definition
  - Der Inhalt einer Pointervariablen ist die Adresse eines Objekts auf das der Pointer zeigt.
  - Konsequenterweise müßten Pointer - wie in anderen Sprachen (z. B. in C) - ein spezieller Datentyp sein. Da man in Fortran90 nicht direkt auf einen Pointer zugreifen kann (keine Adreßarithmetik!), handelt es sich nur um ein Attribut eines Datentyps.
  - In Fortran90 finden Pointer hauptsächlich Verwendung bei der dynamischen Speicherverwaltung (während der Laufzeit des Programms; detailiert in Kapitel 1.8 dargestellt) und beim Aufbau von Datenstrukturen wie verketteten Listen, Baumstrukturen, etc.

Zeiger (Pointer)

- Pointer-Vereinbarung

```
REAL, POINTER :: zeiger, next, current !Typ und Attribut
! oder
REAL zeiger, next, current                  !Typ
Pointer zeiger, next, current               !Attribut
```

- Aus der Pointerdefinition folgt:
  - das Pointer-Attribut darf nur für Variablen oder Funktionen spezifiziert werden
  - folgende weitere Attribute sind unzulässig:
    `ALLOCATABLE, EXTERNAL, INTENT, INTRINSIC, PARAMETER, TARGET`
  - ein Pointer darf weder bei einer Typvereinbarung noch mit Hilfe einer DATA-Anweisung oder E/A initialisiert werden

Zeiger (Pointer)

Pointer-Zuweisung/Freigabe

- Will man einem Pointer eine Variable (oder Funktion) zuweisen, so muß für diese Datengröße das TARGET-Attribut spezifiziert sein.
  - Spezifikation des TARGET-Attributs

```
INTEGER, TARGET :: feld, person       ! Typ und Attribut
! oder
INTEGER feld, person    ! Typ
TARGET feld, person     ! Attribut
```

  - Zuweisung

```
pointer => zielvariable
! oder
ALLOCATE (pointer [,pointer]...) [, STAT=statusvariable]
```

  - Freigabe von Pointerzielen

```
pointer => neuezielvariable          !Überschreiben
! oder
DEALLOCATE (pointer [,pointer]...) [, STAT=statusvariable]
```


Zeiger (Pointer)

Pointer-Zuordnungsstati

- zugeordnet:
  - dem Pointer wurde eine Variable mit dem TARGET-Attribut zugewiesen.
  - dem Pointer wurde ein Pointer zugewiesen, der selbst den Status “zugeordnet” besitzt.
  - dem Pointer wurde mit der ALLOCATE-Anweisung ein Speicherbereich zugeordnet
- nicht zugeordnet:
  - dem Pointer wird mit der NULLIFY-Anweisung eine Zuordnung entzogen (Keine damit verbundene Speicherfreigabe!).
  - NULLIFY (pointer [,pointer]...)
  - dem Pointer wird mit der DEALLOCATE-Anweisung ein Speicherbereich entzogen, der ihm mit einer ALLOCATE-Anweisung zugeordnet wurde.
  - dem Pointer wird ein Pointer mit dem Status “nicht zugeordnet” zugewiesen.

Zeiger (Pointer)

Pointer-Zuordnungsstati(cont)

- undefiniert:
  - dem Pointer wurde noch kein Ziel zugewiesen (weder '=>' noch **ALLOCATE**)
  - das zugeordnete Ziel wurde auf "anderer Weise" entfernt;
    z. B. ein **DEALLOCATE** auf einen Zeiger mit dem gleichen Ziel
  - dem Pointer wird ein Pointer zugewiesen mit dem Staus "undefiniert"
  - wenn das Pointerziel bei Ausführung einer **END**- oder **RETURN**-Anweisung undefiniert wird. (Kapitel 1.6)

Zeiger (Pointer)

ASSOCIATED-Function

- Intrinsic (vordefinierte) Funktion

  `Syntax: ASSOCIATED (pointervar [,targetvar]`

  - liefert den Wert **.TRUE.** beim Aufruf

    `ASSOCIATED (pointervar)`

  wenn **pointervar** irgendeinem Ziel zugeordnet ist und beim Aufruf

    `ASSOCIATED (pointervar, targetvar)`

  wenn **pointervar targetvar** zugeordnet ist.

  - sonst den Wert **.FALSE.**

Beispiele

- zu Datenstrukturen, verkettete Listen, Baumstrukturen, etc
  siehe auch Vorl. mit Übungen : "Programm- und Datenstrukturen"
- zu **ALLOCATE, DEALLOCATE, NULLIFY**
  siehe auch Feldverarbeitung (Dynamische Speicherverwaltung) Kap. 1.8

Zeiger (Pointer)

```
PROGRAM queue

! Warteschlangen-Demo-Programm adaptiert aus:
! Cooper Redwine; "Upgrading to FORTRAN 90", Springer-Verlag,
! 1995
!
! Queue - Policy: first in - first out (Hier zunaechst nur
! Eingabe)

IMPLICIT NONE
TYPE :: QUEUE_ELEMENT
   CHARACTER (20) :: data
   TYPE (QUEUE_ELEMENT), POINTER :: next
END TYPE

! Zeiger auf des erste, letzte, aktuelle Element: first, last,
! this

TYPE (QUEUE_ELEMENT), POINTER :: first, last, this

CHARACTER(20) :: buffer
CHARACTER *4 :: funktion
LOGICAL :: ende = .FALSE.
INTEGER :: io_stat = 1
```


Zeiger (Pointer)

```
   DO WHILE (.NOT. ende)         !Funktionen
   PRINT *, ' '
   PRINT *, 'Eingabe der gewünschten Fkt:'
   PRINT *, ' '
   PRINT *, 'init = Initialisiere Queue'
   PRINT *, 'read = Lese Queue'
   PRINT *, 'erze = Erzeuge Elemente'
   PRINT *, 'dele = Loesche 1. Element'
   PRINT *, 'ende = Beenden des Programms'
   PRINT *, ' '
   PRINT *, 'Funktion:'
   READ *, funktion
   PRINT *, ' '
```

Zeiger (Pointer)

```
SELECT CASE (funktion)
   CASE ('init')
      NULLIFY (first, last)! Initialisiert: leere Queue
   CASE ('read')

      ! Durchlaufe die Warteschlange und drucke Inhalte als
      ! Liste aus.

      this => first           ! aktuelles Element gleich
                              ! 1. Element
      PRINT *, 'Ausgabe'; PRINT *, ' '
      DO WHILE (ASSOCIATED (this))
            PRINT *, this%data
            this => this%next ! aktuelles Element gleich
                              ! naechstes Element
      END DO
```


Zeiger (Pointer)

```
CASE ('erze')
   PRINT *, 'Eingabe:'
   DO WHILE (io_stat >= 0)
      READ (*, "(A)", IOSTAT = io_stat) buffer

      IF (.NOT. ASSOCIATED (first)) THEN

      ! Queue ist leer; fordere den Speicher fuer das
      ! 1. Element an

            ALLOCATE (first)
            last => first     ! Beide Zeiger zeigen aufs
                              ! gleiche(erste)Element
      ELSE
      ! Queue ist nicht leer, neues Element gleich
      ! letztes Element

            ALLOCATE (last%next)
            last => last%next
      END IF
      last%data = buffer
      NULLIFY (last%next)     ! Letztes Element hat
                              ! keinen Nachfolger
   END DO
```

Zeiger (Pointer)

```
      CASE ('dele')
         PRINT *, 'Not yet implemented'
      CASE ('ende')
         PRINT *, 'Programmende'
         ende = .TRUE.
      CASE DEFAULT
         PRINT *, 'Falsche Eingabe!'
      END SELECT
END DO

END PROGRAM queue
```


3.1.6 Unterprogramme

- Definitionen/Terminologie
  - Programmeinheiten (programunits) sind die Bausteine eines ausführbaren Programms
    - Hauptprogramm (genau einmal)
    - ext. Unterprogramme
    - Module
    - Blockdatenprogrammeinheiten
  - Sie enden mit der END-Anweisung, liegen in separaten Dateien, sind getrennt übersetzbar, jedoch bis auf das Hauptprogramm nicht direkt ausführbar.
  - Ein Hauptprogramm ist ausführbar, wenn es keine offenen 'Externreferenzen' gibt - ausgenommen Bibliotheksreferenzen, wie z.b. SIN(x), COS(x), etc.
    Diese Eigenschaft haben unsere bisher geschriebenen Programme.

Unterprogramme

- Die weiteren Programmeinheiten sind nur dann ausführbar (getrennt testbar), wenn ein entsprechendes Hauptprogramm als “Testrahmen” zugebunden wird.
- Nach der Übersetzung sind diese Einheiten zunächst separate Bindemodule; sogenannte ".o-files"

■ Unterprogramme werden unterschieden

- nach ihrem “Definitionsort”
  - extern
  - eingebettet
  - vordefiniert in Bibliothek
- und ihrer Art
  - Subroutine
  - Funktion
  - Formelfunktion

Unterprogramme

■ Subroutine

- Subroutinendefinition

```
[...] SUBROUTINE sname [(...)]        !(SUBROUTINE-Anweisung)
    [Spezifikationsteil]
    ...
        Subroutinenrumpf
    ...
[CONTAINS]
    [interne Unterprogramme]
    ...
END [SUBROUTINE [sname]]
```

- Beispiele:

```
SUBROUTINE sname()                       !(leere Parameterliste)
SUBROUTINE sname (formalparameter[,formalparameter] ...)
```

- Subroutinenaufruf

```
CALL sname
CALL sname()
CALL sname(Aktualparameter[,Aktualparameter]...)
```

Unterprogramme

- Beispiel - eingebettetes Unterprogramm (CONTAINS)

```
PROGRAM summentest
IMPLICIT NONE
REAL a, b, ergebnis

   Print *, 'Eingabe: summand1 summand2 <r>'
   READ *, a,b

   call summe (a,b,ergebnis)

   PRINT *, 'Ergebnis = ',ergebnis

CONTAINS

      SUBROUTINE summe (summanda, summandb, erg)
      REAL summanda, summandb, erg
         erg = summanda + summandb
      END SUBROUTINE summe

END PROGRAM summentest
```


Unterprogramme

- Geltungsbereich/Gültigkeitsbereich
  - sagt aus, in welchen Bereichen eine Variable zugreifbar ist
    - lokal Variable sind nur gültig innerhalb eines Geltungsbereichs z. B. in einem Unterprogramm
    - globale Variable sind in der gesamten Programmeinheit, in der sie definiert wurden, zugreifbar, falls sie nicht durch eine lokale Definition “verdeckt” werden.
      (“Blickt” man aus der Aufrufschachtelung zurück in Richtung Aufrufer, so sind sämtliche globale Daten der aufrufenden Ebenen zugreifbar, es sei denn, sie sind durch lokale Vereinbarungen verdeckt.)

Unterprogramme

- Geltungsbereich/Gültigkeitsbereich(cont)
  - Transfer von Informationen zwischen Geltungsbereichen durch:
    - Parameter
      - Eingangsparameter
      - Ausgangsparameter; Ergebnisparameter
      - beide Richtungen
    - globale Daten
      - im äußersten Geltungsbereich definiert;
        also programmeinheitsglobal
      - COMMON-Bereiche
        Überall dort gültig, wo definiert; auch über Programmeinheiten hinweg

Unterprogramme

- Parameter
  - Formalparameter (bei der Definition)
  - Aktualparameter (beim Aufruf)

    Zum Zeitpunkt des Aufrufs werden die Formalparameter und die Aktualparameter einander zugeordnet, d.h. eine “Formal-Variable” im Unterprogramm referenziert den gleichen Speicher wie die “Aktual-Variable” im aufrufenden Programm.

  Man unterscheidet

    - positionsgerechte Schreibweise
    - Schlüsselwortschreibweise z. B.

      formalparameter = aktualparameter

    Die gemischte Schreibweise ist möglich. Beginnt man jedoch mit der Schlüsselwortschreibweise, so müssen auch alle folgenden Parameter in Schlüsselwortschreibweise folgen.

Unterprogramme

INTENT-Attribut/Anweisung

- Mit dem INTENT-Attribut kann die Art der Verwendung von Formalparametern spezifiziert werden:
  - Das INTENT(IN)-Attribut besagt, daß der Formalparameter ein Eingabeparameter ist. D.h. dieser Formalparameter darf innerhalb des Unterprogramms nicht redefiniert werden, er darf also z. B. nicht auf der linken Seite einer Zuweisung auftauchen oder Variable einer READ-Anweisung sein.
    Der Aktualparameter kann also sowohl eine Variable als auch eine Konstante sein.
  - Das INTENT(OUT)-Attribut besagt, daß der Formalparameter ein Ausgabeparameter ist. D. h. der Formalparameter muß erst definiert (ein Wert zugewiesen) werden, ehe im Unterprogramm auf ihn zugegriffen wird. Hier **muß** es sich also um eine Variable handeln.

Unterprogramme

INTENT-Attribut/Anweisung

- Mit dem INTENT-Attribut kann die Art der Verwendung von Formalparametern spezifiziert werden (cont):
  - Das INTENT(INOUT)-Attribut besagt, daß der Parameter sowohl Eingabe- als auch Ausgabeparameter sein soll. Auch hier **muß** es sich also um eine Variable handeln.

Wir werden im Rahmen dieses Kurses die Formalparameter mit INTENT-Attributen versehen denn: Keine Angabe = INTENT(INOUT), es sei denn, der Aktual-Parameter ist eine Konstante.

Unterprogramme

- Beispiel:

```
PROGRAM summentest
IMPLICIT NONE
REAL a, b, ergebnis

   Print *, 'Eingabe: summand1 summand2 <r>'
   READ *, a,b

   call summe (a,b,ergebnis)

   PRINT *, 'Ergebnis = ',ergebnis

CONTAINS

      SUBROUTINE summe (summanda, summandb, erg)
      REAL, INTENT(IN) :: summanda, summandb
      REAL, INTENT(out) :: erg
         erg = summanda + summandb
      END SUBROUTINE summe

END PROGRAM summentest
```


Unterprogramme

- FUNKTION

```
[...] FUNCTION fname [(...)]!(FUNCTION-Anweisung)
   [Spezifikationsteil]
   ...
      Funktionsrumpf
      ...
[CONTAINS]
   [interne Unterprogramme]
   ...
END [FUNCTION [fname]]
```

- der wesentliche Unterschied einer Funktion zu einer Subroutine ist, daß der Funktionsname gleichzeitig auch der Name der (lokalen) Ergebnisvariablen ist. Deshalb hat die Funktion einen Typ!
- Der Aufruf einer Funktion kann überall dort stehen, wo auch eine Variable stehen darf; nur auf der linken Seite einer Zuweisung (natürlich) nicht.
- Der Funktionsaufruf darf auch ein separates Statement sein - analog zur Subroutine - allerdings ohne “CALL”; jedoch mit “()”.

Unterprogramme

```
( ~uelinste/tmp/newton_fkt.f90 )

PROGRAM newton
REAL eingabe
! Author: Uwe Linster
! Anfordern und einlesen der Zahl, deren Quadratwurzel
! berechnet werden soll

   PRINT *, ' '

   ! Ausgabe auf das Sichtgerät ohne Zeilenvorschub
   ! Nichtvorrueckende Ausgabe: Kapitel 1.10

   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Zu berechnen ist die Quadratwurzel aus: ')

   READ *, eingabe
   PRINT *, ' '; PRINT *, ' '

   DO WHILE(eingabe <= 0)
      PRINT *, ' Die Eingabedaten entsprechen nicht den &
               &Anforderungen des Programms (Zahl > 0!):'
      READ *, eingabe
   END DO
```


Unterprogramme

```
!Berechnung

   ergebnis = sqroot(eingabe)

!Ergebnisausgabe

   PRINT *, ' Die Quadratwurzel aus ', eingabe ;PRINT *, ' '
   PRINT *, ' ist ', ergebnis
! Oder auch gleich:
!  PRINT *, ' ist ', sqroot(eingabe)

PRINT *, ' '; PRINT *, ' Ende des Programms'
```

Unterprogramme

```
CONTAINS

REAL FUNCTION sqroot (a)
!Berechnung der Quadratwurzel nach dem Newton'schen Verfahren
!Eingabeparameter:a; Startwert der Iteration: 1
!Iterationsformel:x(i+1) = (x(i)*x(i) + a) / (2*x(i))

   REAL , INTENT(IN) :: a
   REAL x
      !a Größe, deren Quadratwurzel zu berechnen ist
      !x Berechnete Approximation der Quadratwurzel

   !Anfangsapproximation
   x = 1
   !Iteration
   DO WHILE (.NOT. (ABS(x*x - a) <= 1E-4))
      x = (x*x + a) / (2.*x)
   END DO
   sqroot = x
END FUNCTION sqroot

END PROGRAM newton
```


Unterprogramme

COMMON-BEREICHE

- Durch COMMON-Bereiche werden globale Speicherbereiche spezifiziert. Variable sind in den Geltungsbereichen direkt zugreifbar, in denen die COMMON-Bereiche spezifiziert wurden; auch über verschiedene Programmeinheiten hinweg.
  Man unterscheidet:

  - einen unbenannten COMMON-Block

```
Syntax:COMMON variablenliste
```

  - und beliebig viele benannte COMMON-Blöcke

```
Syntax:COMMON /blockname/ variablenliste [, /blockname/]
                variablenliste]...
```

  - auch die Mischschreibweise ist möglich:

```
Syntax:COMMON   variablenliste [[,]/[blockname]/
                variablenliste]...
```

Unterprogramme

- COMMON-BEREICHE(cont)
  - Eigenschaften
    - Durch Angabe der Variablen wird eine Speicherfolge festgelegt; um Fehler zu vermeiden sollte man mit “copy” und “paste” arbeiten.
    - Variablennamen in verschiedenen Geltungseinheiten dürfen verschieden sein; das sollte man jedoch nur tun, wenn es dafür auch einen plausiblen Grund gibt. Man greift auf dieselbe globale Speicheradresse zu, das sollte man durch den gleiche Variablennamen auch dokumentieren!
    - Initialisiert werden COMMON-Bereiche mit einer BLOCK DATA-Programmeinheit; jedoch nur die benannten Blöcke.
    - Selbstdefinierte Datentypen müssen das Attribut SEQUENCE besitzen. (Speicherfolge!)

Unterprogramme

- COMMON-Bereich - Beispiel (~uelinste/tmp/common_demo_bubble.f90

```
PROGRAM bubble
IMPLICIT NONE

! Zweck des Programms, Loesungsverfahren:
!
! Die Eintraege in der Datei titel sollen in lexikographisch
! sortierter Reihenfolge gedruckt werden. Das Programm geht
! davon aus, dass die Datei maximal 200 Eintraege besitzt und
! jeder Eintrag aus hoechstens 20 Zeichen besteht.
! Als Sortierverfahren wird 'bubble sort' verwendet.
!------------------------------------------------------------
! Globale Variable; in COMMON-Bereiche spezifiziert
!    name Feld zur Aufnahme der Eintraege aus der Dateri titel
!    max Anzahl der Eintraege
!  fd Kanalnummer (Filedescriptor; Unit)
!------------------------------------------------------------
! Initialisierung
!
! Ausgabe der Gesamtueberschrift
!
PRINT *, 'Eintraege der Datei titel lexikograpgisch sortiert:'
```

Unterprogramme

```
! Oeffnen der Datei
!
   CALL oeffnen_datei()

! Einlesen der Daten
!
   CALL einlesen()
!
! Sortieren
!
   CALL bubblesort()
!
! Ausgabe des Ergebnisses
!
   CALL ausgeben_screen()
   CALL ausgeben_datei()
```


Unterprogramme

```
! Die Unterprogramme:

CONTAINS

SUBROUTINE oeffnen_datei()

! Oeffnen der Datei "titel" mit Kanalnummer 10
! Global Data:

INTEGER fd
COMMON /datei/ fd
   fd = 10      ! Entfaelt bei BLOCK-DATA-INIT!!
! Local DATA: none

   open(fd, FORM = 'FORMATTED', file = 'titel')

END SUBROUTINE oeffnen_datei
```

Unterprogramme

```
SUBROUTINE einlesen()

! Einlesen der Eintraege aus der Datei "titel" in das Feld !
"name".
! Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

!Local Data:
INTEGER i

   DO i = 1, 200
      READ(fd, 1000, END = 1010) name(i)
1000  FORMAT (a)
   END DO

1010max = i - 1
   PRINT *, 'Gelesene Datensaetze: ',max

END SUBROUTINE einlesen
```


Unterprogramme

```
SUBROUTINE bubblesort()
! Sortiert nach dem "bubble-sort"-Verfahren
! Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

!Local Data

CHARACTER *20 :: hilf
LOGICAL flag
INTEGER i, j, dl

!  flag   zeigt an, ob bei einem Durchlauf Aenderungen in der
!              Anordnung vorgenommen wurden
!              (.FALSE.: Keine Umordnungen
!               .TRUE. : Es wurden Umordnungen vorgenommen)
!  hilf   Hilfsfeld waehrend des Vertauschens zweier Eintraege
!  dl Anzahl der Sortierschleifen
!
```

Unterprogramme

```
! Der Sortiervorgang

   dl = 0
   DO  i = max, 2, -1
      flag = .FALSE.

      DO j = 1, i - 1
          IF (name(j) > name(j + 1)) THEN
               hilf = name(j)
               name(j) = name(j + 1)
               name(j + 1) = hilf
               flag = .TRUE.
               dl = dl + 1
          END IF
        END DO

        if (.NOT. flag) THEN
         PRINT *, 'Anzahl der Durchlauefe: ', dl
         EXIT
      END IF

   END DO

END SUBROUTINE bubblesort
```


Unterprogramme

```
SUBROUTINE ausgeben_screen()

!Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max
COMMON name, max

! Local Data:

INTEGER i

   DO i=1,max
      Print *, name(i)
   END DO

END SUBROUTINE ausgeben_screen
```

Unterprogramme

```
SUBROUTINE ausgeben_datei()

!Global Data

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

! Local Data

INTEGER i

   REWIND (fd)
   WRITE ( fd, 1030 )(name(i), i=1, max)
1030FORMAT (a)

END SUBROUTINE ausgeben_datei

END PROGRAM bubble
```


Unterprogramme

- BLOCK DATA
  - Eine BLOCK-DATA-Programmeinheit
    - ist eine nichtausführbare Programmeinheit, sie dient der Initialisierung von Variablen in **benannten** COMMON-Bereichen.
    - COMMON-Blöcke sind globale Speicherbereiche - gültig auch über Programm-Unit-Grenzen hinweg. Eine Initialisierung darf also Programmsystemweit nur einmal erfolgen. Dies könnte innerhalb einer “ausgezeichneten” Programmeinheit erfolgen oder wie hier in einer speziellen Programmeinheit

```
BLOCKDATA [name]
     [Spezifikationsteil]
     .
     .
     .
  END BLOCKDATA [name]
```

Unterprogramme

- BLOCK DATA(cont)

```
Beispiel
BLOCKDATA car

CHARACTER vorderachse *14, hinterachse *10, motor *11
INTEGER anzahl, seriennummer, partnb

COMMON /ersatzteile/ vorderachse, hinterachse, motor
COMMON /attribute/ anzahl, seriennummer, partnb

DATA vorderachse, hinterachse /'Schwingachsen', `Inkaachsen'/
DATA motor /'Dieselmotor'/
DATA anzahl, partnb /250, 4711/ ! seriennummer bleibt
uninitialisiert

END BLOCKDATA car
```

- Da diese Programm-Unit in einer eigenen Datei liegt muß beim Aufruf des Compilers diese Datei als weiterer Parameter angegeben werden:; z. B.

```
f90 hauptprogramm.f90 blockdata.f90 -o programm.out
```


Unterprogramme

- BLOCK DATA
  - Beispiel für Programm “bubble”:

Datei: bubble_blockdata.f90

```
BLOCKDATA bubble

   INTEGER fd

   COMMON /datei/ fd

   DATA fd /10/

END BLOCKDATA bubble
```

- Aufruf des Compilers:

```
f90 common_demo_bubble.f90 bubble_blockdata.f90 -o bubble.out
```

Unterprogramme

FORMELFUNKTION

- eine Formelfunktion ist eine Funktion, die nur aus einer einzigen Anweisung besteht und
- sie ist nur innerhalb einer Geltungseinheit aufrufbar.
- Formelfunktionsdefinition:

```
Syntax:ffname() = ausdruck oder
ffname(formalparameter[,formalparameter]...) =
                              ausdruck
```

- Sowohl 'ausdruck' als auch 'formalparameter' sind skalare Datenobjekte.
- Formelfunktionsaufruf:

```
Syntax:ffname() oder
ffname(formalparameter[,formalparameter]...)
```

- Der Typ der Formelfunktion wird separat (wie bei der Typvereinbarung einer Variablen) vereinbart; dieses gilt auch für die Formalparameter.

Unterprogramme

- Formelfunktion - Beispiel (~uelinste/tmp/capslock_fkt.f90)

```
PROGRAM capslock
IMPLICIT NONE
!
! Das Programm liest Kleinbuchstaben ein, wandelt sie
! in Grossbuchstaben um und druckt diese aus.
!  Variable
!
      CHARACTER, DIMENSION(20) :: zeichenin, zeichenout
      INTEGER i

   CHARACTER gross, klein

!  Formelfunktion:

   gross(klein) = CHAR ( ICHAR(klein) + ICHAR('A') - ICHAR('a'))
```

Unterprogramme

```
!  Einlesen der Zeichen

      PRINT *, ' '
   PRINT *, 'Kleinbuchstaben; max. 20; entweder durch "blank"
getrennt -'
   PRINT *, 'Abschluss <r> oder einzeln durch <r> getrennt!'
   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Eingabe: ')
      READ *, (zeichenin(i), i=1,20)

   DO i = 1,20
      zeichenout(i) = gross(zeichenin(i))
   END DO

   PRINT *, 'Ausgabe:',zeichenout

END PROGRAM capslock
```


Unterprogramme

- Formelfunktion - Beispiel (~uelinste/tmp/kredit_fkt.f90)

```
PROGRAM kredit
IMPLICIT NONE
!
!  Das Programm berechnet die Hoehe der monatlichen Raten fuer
! einen vorgebbaren Kaufpreis bei einzugebender Laufzeit. Raten
!  und Zinsen werden von zwei Jahren bis zu einer angebbaren
!  maximalen Laufzeit in Jahresschritten berechnet. Untere und
!  oebere Grenze des Zinssatzes werden ebenfalls eingelesen.
! ------------------------------------------------------------
!  Variable
!
      REAL preis, mrate, p
      INTEGER n, zmin, zmax, zsatz, zs, year, yearmax
!
! Berechnung der Monatsrate fuer die verschiedenen Zinssaetze
!          formel: r*p/(1-(1+r)**(-n)) mit r = zinssatz/12/100
!  hier als Formelfunktion realisiert:

   mrate(zs,p,n) = ((zs/100./12.) * p) / &
             (1.-(1. + (zs/100./12.))**(-n))
```

Unterprogramme

```
!        preis      Kaufpreis (Eingabe)
!       yearmax   maximale Laufzeit der Ratenzahlung (Eingabe)
!        zmin       minimaler Zinssatz (Eingabe)
!        zmax       maximaler Zinssatz (Eingabe)
!        year       Laufvariable fuer die Laufzeit
!        n          Zahl der Raten
!        zsatz      Laufvariable fuer den Zinssatz
! ------------------------------------------------------------
! Initialisierung
!
!  Einlesen von Kaufpreis, maximaler Laufzeit,
!  minimalem und maximalem Zinssatz
!
      PRINT *, ' '
   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Kaufpreis (REAL): ')
      READ *, preis
   WRITE (*, 2000, ADVANCE='NO')
2000FORMAT ('Laufzeit in Jahren (groesser als 2): ')
      READ *, yearmax
     WRITE (*, 3000, ADVANCE='NO')
3000FORMAT ('minimaler und maximaler Zins (in Prozent
(INTEGER)): ')
      READ *, zmin, zmax; PRINT *, ' '
```


Unterprogramme

```
! Aeussere Schleife

    DO year = 2, yearmax

!  Tabellenueberschrift
        PRINT *, ' '
        PRINT *, ' '
      PRINT *, ' Zinsen fuer eine Laufzeit von ', year, ' Jahren'
        PRINT *, ' '
       PRINT *, ' Zins-  Zahl der     Rate          Gesamtzins'
        PRINT *, ' satz    Raten       (DM)              (DM)'

! Innere Schleife
      DO zsatz = zmin, zmax
         n = 12.*year
          PRINT *, ' ', zsatz, '      ', n, ' ', &
               mrate (zsatz, preis, n) &
               ,' ', n * (mrate (zsatz, preis, n)) -preis
      END DO
    END DO
!
!        Abschlussausgabe
      PRINT *, ' ' ;PRINT *, ' Berechnung abgeschlossen.'
END PROGRAM kredit
```

Unterprogramme

■ Rücksprung

▲ Es existieren drei alternative Methoden

- der “normale” Rücksprung bei Erreichung des END-Statements
- der “Ausnahme” - Rücksprung, wenn aufgrund einer Bedingung die Aufgabe des Unterprogramms erfüllt ist - etwa analog zum EXIT-Aussprung aus einer DO-Schleife

```
Syntax:RETURN
```

Bei beiden Methoden wird mit dem 1. ausführbaren Statement fortgefahren, das dem Unterprogrammaufruf folgt.

- der “Ausnahme” - Rücksprung mit wählbarer Fortsetzungsadresse (label).

```
Syntax:RETURN ganzzahliger_ausdruck
```

- Diese Möglichkeit wird nur der Vollständigkeit erwähnt (Im Falle, daß man mit vorhandenen Programmsystemen umgehen muß). Die Verwendung solcher Konstrukte führt zu undurchsichtigen, nicht mehr wartbaren Programmsystemen.

Unterprogramme

■ SAVE-Anweisung

▲ Mit dem SAVE-Attribut wird der Wert einer **lokalen** Variablen in einem Geltungsbereich über die “Lebensdauer” eines Aufrufs hinaus “gesichert”. D. h., daß der Wert der Variablen auch beim nächsten Aufruf wieder verfügbar ist. Man spricht auch von “*Static”* - Variablen.

▲ Das SAVE-Attribut ist nur erlaubt bei

- lokalen Variablen
- benannten COMMON-Blöcken
  (hier ist Vorsicht geboten, ist doch ein COMMON-Block ein globaler Speicherbereich!?!)

▲ Das SAVE-Attribut ist insbesondere nicht erlaubt bei

- einzelnen Variablen, die in einem COMMON-Bereich enthalten sind
- Formalparametern
- Funktionsvariablen
- Namensgruppen (Siehe Kapitel 1.10)

Unterprogramme

SAVE-Anweisung

Attributspezifikation

- gleichzeitig mit der Typvereinbarung

```
REAL, SAVE :: skonto, summe
INTEGER, SAVE :: max
```

- oder getrennt mit einer SAVE-Anweisung

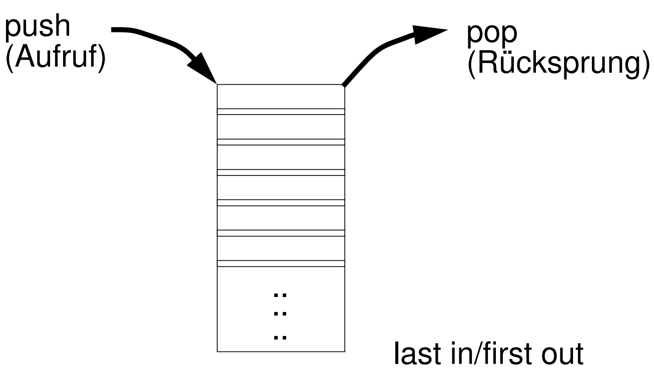
```
REAL skonto, summe
SAVE skonto, summe

INTEGER max
SAVE max
```

Eine SAVE-Anweisung ohne Angabe von Variablennamen bedeutet, das sämtliche lokale Variable in diesem Geltungsbereich diese Spezifikation erhalten.

Rekursive Unterprogramme

Unterprogramme (SUBROUTINE oder FUNCTION) sind dann rekursiv, wenn sie sich selbst direkt oder indirekt aufrufen.

Wenn Unterprogramme das Attribut RECURSION besitzen, muß der Compiler dafür Sorge tragen, daß jeweils bei einem rekursiven Aufruf die lokalen Variablen gerettet und nach dem Rücksprung aus einer tieferen Aufrufebene wieder bereitgestellt werden. Dies ist eine typische Anwendung für einen Kellerspeicher (Stack)!



Rekursive Unterprogramme

- Wann verwendet man rekursive Unterprogramme?
  - Es gibt eine Anzahl von Algorithmen, die "in sich" rekursiv definiert sind. Bei der iterativen Implementierung solcher Algorithmen ist der Programmieraufwand in der Regel beträchtlich und mit einer großen Anzahl von Abfragen behaftet, wie einige Beispiele des Kapitels noch zeigen werden.
  - Als einfaches Beispiel - zum Verständnis - wurde die Berechnung von Fakultäten gewählt, tatsächlich ist es in diesem Fall allerdings effizienter, eine Schleife zu implementieren!

```
PROGRAM recursion
IMPLICIT NONE
INTEGER n, erg
   WRITE (*, FMT = "('n! ; Welches n? : ')", ADVANCE = 'NO')
   READ *, n
   CALL fak (n, erg)
   PRINT *, 'Fakultaet von: ',n ,' = ', erg
   erg = fakfunc (n)
   PRINT *, 'Fakultaet von: ',n ,' = ', erg
CONTAINS
```


Rekursive Unterprogramme

- Rekursive Subroutine

```
RECURSIVE SUBROUTINE fak (fn, ferg)

INTEGER, INTENT (IN) :: fn
INTEGER, INTENT (OUT) :: ferg

   IF (fn == 0) THEN
         ferg = 1
   ELSE
         CALL fak (fn - 1, ferg)
         IF (ferg < 0) THEN
               PRINT *, 'underflow'
               STOP
         END IF
         ferg = fn * ferg
   END IF

END SUBROUTINE fak
```

Rekursive Unterprogramme

▲ Rekursive Funktion

```
RECURSIVE INTEGER FUNCTION fakfunc (fn) RESULT (ferg)

INTEGER, INTENT (IN) :: fn

   IF (fn == 0) THEN
         ferg = 1
   ELSE
         ferg = fn * fakfunc (fn - 1)
         IF (ferg < 0) THEN
               PRINT *, 'underflow'
               STOP
         END IF
   END IF

END FUNCTION fakfunc

END PROGRAM recursion
```


Unterprogramme

■ (Noch) nicht behandelt

- Parameterübergabe bei nicht skalaren Datengrößen (Siehe Kapitel 1.7- Feldverarbeitung)
- Module (Siehe Kapitel 1.11 - Module)
- Externe Unterprogramme (Siehe Kapitel n.b.)
- Nebeneinganspunkte - ENTRY-Anweisungen (Siehe Kapitel n.b.)
- Optionale Formalparameter (Siehe Kapitel n.b.)

Programmierung Paralleler Systeme

3. Programmiermodell “Single Programm Multiple Data”

3.1 Fortran95

3.1 .4 Selbstdef. Datentypen (Strukturen)

3.1.5 Zeiger(Pointer)

3.1.6 Unterprogramme

Claus-Uwe Linster

IMMD IV, Lehrstuhl für Betriebssysteme
Universität Erlangen-Nürnberg

linster@informatik.uni-erlangen.de

3.1.4 Selbstdefinierte Typen

▲ Typdefinition

```
TYPE [,attribut]... [::] typname
   komponentendefinition
   [komponentendefinition]
   .
END TYPE [typname]
```

Die Komponentendefinition entspricht weitgehend den Typvereinbarungsanweisungen; jedoch

- keine Initialisierung (es ist eine Definition; keine Deklaration!)
- als Attribute sind nur
  - DIMENSION (mit expliziter Gestalt) und
  - POINTER (Siehe Kapitel 1.5)

  zugelassen.

Selbstdefinierte Typen

Als Attribute der Typdefinition (dann gültig für alle Deklarationen dieses Typs) sind zulässig:

- PRIVATE (Siehe Kapitel 1.11)
- PUBLIC (Siehe Kapitel 1.11)
- SEQUENCE
  - erzwingt eine Speicherordnung gemäß der Komponentenreihenfolge; dies kann hinsichtlich der Speichernutzung Verschnitt bedeuten.
  - Verschiedene Typdefinitionen mit gleichen Komponenten (z. B. in verschiedenen Gültigkeitsbereichen)sind nur dann wirklich gleich, wenn sie das SEQUENCE-Attribut besitzen.
  - Wenn ein selbstdefinierter Typ das SEQUENCE-Attribut besitzt, dann müssen auch alle ggf. vorhandenen Typ-Komponenten, die ihrerseits selbstdefinierte Typen sind, das SEQUENCE-Attribut besitzen.
- Type - Vereinbarungsanweisung

```
Syntax:TYPE (TYPENAME) [,attribut]... [::] dg [,dg]...
```


Selbstdefinierte Typen

▲ Beispiele für Typdefinitionen

```
TYPE:: DATUM
  SEQUENCE
  INTEGER :: tag
  CHARCATER (LEN=3) :: monat
  INTEGER :: jahr
END TYPE DATUM
TYPE :: MITARBEITER
  CHARACTER *20 :: name, vorname
  TYPE (DATUM) geburtsdatum
  REAL gehalt
  REAL lohnsteuer
  LOGICAL verheiratet
  TYPE (MITARBEITER), POINTER :: alphanext ! (Siehe Kapitel 7)
END TYPE MITARBEITER
```

▲ Beispiel für Typvereinbarungen

```
TYPE (MITARBEITER), DIMENSION (100) :: gehaltsliste
TYPE (DATUM), DIMENSION (365) :: kalender
```

Selbstdefinierte Typen

- Strukturkonstante

```
Syntax: TYPNAME (komponentenliste)
  z. B. DATUM (1, 'Jan', 1996)
```

- Zuweisungen
  - Komponentenweise Zuweisung

```
variablenname%komponentenname = wert
```

  - Objektweise Zuweisung

```
variablenname = strukturkonstante
variablenname = variablenname      !gleichen Typs!
```


Selbstdefinierte Typen

- Beispiele

```
kalender(1)  = DATUM(1,'Jan',1996)
kalender(41)%tag = 10
kalender(41)%monat = 'Feb'
kalender(41)%Jahr = 1941

gehaltsliste(35) = MITARBEITER ('Mueller', 'Horst',&
     DATUM(10,'Feb',1941),4523.12,1212.00,.TRUE.,&
     gehaltsliste(25))

gehaltsliste(42)  = gehaltsliste(35)
gehaltsliste(42)%vorname = 'Anton'
gehaltsliste(42)%alphanext = gehaltsliste(35)
! Kapitel 1.5
```

© Claus-Uwe Linster – pps3.1-2_spmd_f90: 19/3/98

Selbstdefinierte Typen

weitere Beispiele

- Typdefinition

```
TYPE :: ERSATZTEIL
  SEQUENCE
  CHARACTER *10 pkw_typ
  CHARACTER *10 part_nb
END ERSATZTEIL

TYPE :: PKW
  SEQUENCE
  CHARACTER *10 farbe
  INTEGER fahrgestellnummer
  INTEGER zulassungsjahr
  TYPE (ERSATZTEIL), DIMENSION (100) :: vorderachse
END TYPE PKW
```


Selbstdefinierte Typen

weitere Beispiele

- Typvereinbarungen

```
TYPE (PKW), DIMENSION (200) :: lager
```

- Zuweisungen

```
lager(110)%vorderachse(12)%part_nb = ’1000G1019’
lager(1)%farbe = ’RAL312’
```

Selbstdefinierte Typen

```
PROGRAM personal
IMPLICIT NONE
INTEGER, PARAMETER :: max = 10
INTEGER, PARAMETER :: fd = 10
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
!  CHARACTER, DIMENSION (:), POINTER :: beurteilung! Kap. 1.5
END TYPE MITARBEITER

TYPE (MITARBEITER), DIMENSION (max) :: pa! Personal_akte
CHARACTER *4 funktion
CHARACTER zeichen
INTEGER i
LOGICAL :: ende = .FALSE.
```


Selbstdefinierte Typen

```
OPEN(fd,FORM='UNFORMATTED',file = 'database.personal')
   ! Oeffnen der Datei
DO WHILE (.NOT. ende)      !Funktionen

   PRINT *, ' '
   PRINT *, 'PRINT *, 'Eingabe der gewünschten Fkt:'; PRINT *,
' '
   PRINT *, 'init = Initialisiere Datenbasis'
   PRINT *, 'read = Lesen der Datenbasis'
   PRINT *, 'einz = Einzel-Eingabe'
   PRINT *, 'eing = Gruppeneingabe'
   PRINT *, 'korr = Korrektur;'
   PRINT *, 'ausg = Ausgabe;'
   PRINT *, 'sumg = Summe Gehaelter;'
   PRINT *, 'suml = Summe Lohnsteuer;'
   PRINT *, 'pers = Personalliste;'
   PRINT *, 'ende = Beenden des Programms'
   PRINT *, ' '
   PRINT *, 'Funktion?'
   READ *, funktion
   PRINT *, ' '
```

Selbstdefinierte Typen

```
SELECT CASE (funktion)
   CASE ('init')
      PRINT *,'Sind Sie sicher? (y)'
      READ *,zeichen
      if (zeichen /= 'y') EXIT
      DO i = 1,max
            pa(i)%name = 'frei'
            pa(i)%vorname = 'frei'
      END DO
   CASE ('read')
      READ(fd) (pa(i), i=1,max)
   CASE ('einz')
      DO i = 1,max
            IF (pa(i)%name == 'frei') THEN
                  Print *,'Name:'
                  READ *, pa(i)%name
                  PRINT *, 'Vorname'
                  READ * , pa(i)%vorname
                  EXIT
            ELSE
                  CYCLE
            END IF
      END DO
```


Selbstdefinierte Typen

```
   CASE ('eing')
      PRINT *, 'Not yet implemented'
   CASE ('korr')
      PRINT *, 'Not yet implemented'
   CASE ('ausg')
      PRINT *, 'Not yet implemented'
   CASE ('sumg')
      PRINT *, 'Not yet implemented'
   CASE ('suml')
      PRINT *, 'Not yet implemented'
   CASE ('pers')
      DO i= 1,max
            PRINT *, pa(i)%name, ';', pa(i)%vorname
      END DO
   CASE ('ende')
      PRINT *, 'Programmende'
      WRITE(fd) (pa(i), i = 1,max)
      ende = .TRUE.
   CASE DEFAULT
      PRINT *, 'Falsche Eingabe!'
   END SELECT
END DO
END PROGRAM personal
```

3.1.5 Zeiger (Pointer)

- Pointer Definition
  - Der Inhalt einer Pointervariablen ist die Adresse eines Objekts auf das der Pointer zeigt.
  - Konsequenterweise müßten Pointer - wie in anderen Sprachen (z. B. in C) - ein spezieller Datentyp sein. Da man in Fortran90 nicht direkt auf einen Pointer zugreifen kann (keine Adreßarithmetik!), handelt es sich nur um ein Attribut eines Datentyps.
  - In Fortran90 finden Pointer hauptsächlich Verwendung bei der dynamischen Speicherverwaltung (während der Laufzeit des Programms; detailliert in Kapitel 1.8 dargestellt) und beim Aufbau von Datenstrukturen wie verketteten Listen, Baumstrukturen, etc.

Zeiger (Pointer)

- Pointer-Vereinbarung

```
REAL, POINTER :: zeiger, next, current !Typ und Attribut
! oder
REAL zeiger, next, current                  !Typ
Pointer zeiger, next, current               !Attribut
```

- Aus der Pointerdefinition folgt:
  - das Pointer-Attribut darf nur für Variablen oder Funktionen spezifiziert werden
  - folgende weitere Attribute sind unzulässig:
    ALLOCATABLE, EXTERNAL, INTENT, INTRINSIC, PARAMETER, TARGET
  - ein Pointer darf weder bei einer Typvereinbarung noch mit Hilfe einer DATA-Anweisung oder E/A initialisiert werden

Zeiger (Pointer)

Pointer-Zuweisung/Freigabe

- Will man einem Pointer eine Variable (oder Funktion) zuweisen, so muß für diese Datengröße das TARGET-Attribut spezifiziert sein.
  - Spezifikation des TARGET-Attributs

```
INTEGER, TARGET :: feld, person      ! Typ und Attribut
! oder
INTEGER feld, person    ! Typ
TARGET feld, person     ! Attribut
```

  - Zuweisung

```
pointer => zielvariable
! oder
ALLOCATE (pointer [,pointer]...) [, STAT=statusvariable]
```

  - Freigabe von Pointerzielen

```
pointer => neuezielvariable          !Überschreiben
! oder
DEALLOCATE (pointer [,pointer]...) [, STAT=statusvariable]
```


Zeiger (Pointer)

Pointer-Zuordnungsstati

- zugeordnet:
  - dem Pointer wurde eine Variable mit dem TARGET-Attribut zugewiesen.
  - dem Pointer wurde ein Pointer zugewiesen, der selbst den Status “zugeordnet” besitzt.
  - dem Pointer wurde mit der ALLOCATE-Anweisung ein Speicherbereich zugeordnet
- nicht zugeordnet:
  - dem Pointer wird mit der NULLIFY-Anweisung eine Zuordnung entzogen (Keine damit verbundene Speicherfreigabe!).
  - NULLIFY (pointer [,pointer]...)
  - dem Pointer wird mit der DEALLOCATE-Anweisung ein Speicherbereich entzogen, der ihm mit einer ALLOCATE-Anweisung zugeordnet wurde.
  - dem Pointer wird ein Pointer mit dem Status “nicht zugeordnet” zugewiesen.

Zeiger (Pointer)

Pointer-Zuordnungsstati(cont)

- undefiniert:
  - dem Pointer wurde noch kein Ziel zugewiesen (weder ’=>’ noch **`ALLOCATE`**)
  - das zugeordnete Ziel wurde auf “anderer Weise” entfernt;
    z. B. ein **`DEALLOCATE`** auf einen Zeiger mit dem gleichen Ziel
  - dem Pointer wird ein Pointer zugewiesen mit dem Staus “undefiniert”
  - wenn das Pointerziel bei Ausführung einer **`END`**- oder **`RETURN`**-Anweisung undefiniert wird. (Kapitel 1.6)

Zeiger (Pointer)

ASSOCIATED-Function

- Intrinsic (vordefinierte) Funktion

  `Syntax: ASSOCIATED (pointervar [,targetvar]`

  - liefert den Wert **`.TRUE.`** beim Aufruf

    `ASSOCIATED (pointervar)`

    wenn **`pointervar`** irgendeinem Ziel zugeordnet ist und beim Aufruf

    `ASSOCIATED (pointervar, targetvar)`

    wenn **`pointervar targetvar`** zugeordnet ist.
  - sonst den Wert **`.FALSE.`**

Beispiele

- zu Datenstrukturen, verkettete Listen, Baumstrukturen, etc
  siehe auch Vorl. mit Übungen : "Programm- und Datenstrukturen"
- zu **`ALLOCATE, DEALLOCATE, NULLIFY`**
  siehe auch Feldverarbeitung (Dynamische Speicherverwaltung) Kap. 1.8

Zeiger (Pointer)

```
PROGRAM queue

! Warteschlangen-Demo-Programm adaptiert aus:
! Cooper Redwine; "Upgrading to FORTRAN 90", Springer-Verlag,
! 1995
!
! Queue - Policy: first in - first out (Hier zunaechst nur
! Eingabe)

IMPLICIT NONE
TYPE :: QUEUE_ELEMENT
   CHARACTER (20) :: data
   TYPE (QUEUE_ELEMENT), POINTER :: next
END TYPE

! Zeiger auf des erste, letzte, aktuelle Element: first, last,
! this

TYPE (QUEUE_ELEMENT), POINTER :: first, last, this

CHARACTER(20) :: buffer
CHARACTER *4 :: funktion
LOGICAL :: ende = .FALSE.
INTEGER :: io_stat = 1
```


Zeiger (Pointer)

```
   DO WHILE (.NOT. ende)         !Funktionen
   PRINT *, ' '
   PRINT *, 'Eingabe der gewünschten Fkt:'
   PRINT *, ' '
   PRINT *, 'init = Initialisiere Queue'
   PRINT *, 'read = Lese Queue'
   PRINT *, 'erze = Erzeuge Elemente'
   PRINT *, 'dele = Loesche 1. Element'
   PRINT *, 'ende = Beenden des Programms'
   PRINT *, ' '
   PRINT *, 'Funktion:'
   READ *, funktion
   PRINT *, ' '
```

Zeiger (Pointer)

```
SELECT CASE (funktion)
   CASE ('init')
      NULLIFY (first, last)! Initialisiert: leere Queue
   CASE ('read')

      ! Durchlaufe die Warteschlange und drucke Inhalte als
      ! Liste aus.

      this => first           ! aktuelles Element gleich
                              ! 1. Element
      PRINT *, 'Ausgabe'; PRINT *, ' '
      DO WHILE (ASSOCIATED (this))
            PRINT *, this%data
            this => this%next ! aktuelles Element gleich
                              ! naechstes Element
      END DO
```


Zeiger (Pointer)

```
CASE ('erze')
   PRINT *, 'Eingabe:'
   DO WHILE (io_stat >= 0)
      READ (*, "(A)", IOSTAT = io_stat) buffer

      IF (.NOT. ASSOCIATED (first)) THEN

      ! Queue ist leer; fordere den Speicher fuer das
      ! 1. Element an

            ALLOCATE (first)
            last => first     ! Beide Zeiger zeigen aufs
                              ! gleiche(erste)Element
      ELSE
      ! Queue ist nicht leer, neues Element gleich
      ! letztes Element

            ALLOCATE (last%next)
            last => last%next
      END IF
      last%data = buffer
      NULLIFY (last%next)     ! Letztes Element hat
                              ! keinen Nachfolger
   END DO
```

Zeiger (Pointer)

```
      CASE ('dele')
         PRINT *, 'Not yet implemented'
      CASE ('ende')
         PRINT *, 'Programmende'
         ende = .TRUE.
      CASE DEFAULT
         PRINT *, 'Falsche Eingabe!'
      END SELECT
END DO

END PROGRAM queue
```


3.1.6 Unterprogramme

- Definitionen/Terminologie
  - Programmeinheiten (programunits) sind die Bausteine eines ausführbaren Programms
    - Hauptprogramm (genau einmal)
    - ext. Unterprogramme
    - Module
    - Blockdatenprogrammeinheiten
  - Sie enden mit der END-Anweisung, liegen in separaten Dateien, sind getrennt übersetzbar, jedoch bis auf das Hauptprogramm nicht direkt ausführbar.
  - Ein Hauptprogramm ist ausführbar, wenn es keine offenen ’Externreferenzen’ gibt - ausgenommen Bibliotheksreferenzen, wie z.b. SIN(x), COS(x), etc.
    Diese Eigenschaft haben unsere bisher geschriebenen Programme.

Unterprogramme

- Die weiteren Programmeinheiten sind nur dann ausführbar (getrennt testbar), wenn ein entsprechendes Hauptprogramm als “Testrahmen” zugebunden wird.
- Nach der Übersetzung sind diese Einheiten zunächst separate Bindemodule; sogenannte ".o-files"

- Unterprogramme werden unterschieden
  - nach ihrem “Definitionsort”
    - extern
    - eingebettet
    - vordefiniert in Bibliothek
  - und ihrer Art
    - Subroutine
    - Funktion
    - Formelfunktion

Unterprogramme

Subroutine

- Subroutinendefinition

```
[...] SUBROUTINE sname [(...)]        !(SUBROUTINE-Anweisung)
    [Spezifikationsteil]
    ...
        Subroutinenrumpf
    ...
[CONTAINS]
    [interne Unterprogramme]
    ...
END [SUBROUTINE [sname]]
```

- Beispiele:

```
SUBROUTINE sname()                     !(leere Parameterliste)
SUBROUTINE sname (formalparameter[,formalparameter] ...)
```

- Subroutinenaufruf

```
CALL sname
CALL sname()
CALL sname(Aktualparameter[,Aktualparameter]...)
```

Unterprogramme

- Beispiel - eingebettetes Unterprogramm (CONTAINS)

```
PROGRAM summentest
IMPLICIT NONE
REAL a, b, ergebnis

   Print *, 'Eingabe: summand1 summand2 <r>'
   READ *, a,b

   call summe (a,b,ergebnis)

   PRINT *, 'Ergebnis = ',ergebnis

CONTAINS

      SUBROUTINE summe (summanda, summandb, erg)
      REAL summanda, summandb, erg
         erg = summanda + summandb
      END SUBROUTINE summe

END PROGRAM summentest
```


Unterprogramme

- Geltungsbereich/Gültigkeitsbereich
  - sagt aus, in welchen Bereichen eine Variable zugreifbar ist
    - lokal Variable sind nur gültig innerhalb eines Geltungsbereichs z. B. in einem Unterprogramm
    - globale Variable sind in der gesamten Programmeinheit, in der sie definiert wurden, zugreifbar, falls sie nicht durch eine lokale Definition “verdeckt” werden.
      (“Blickt” man aus der Aufrufschachtelung zurück in Richtung Aufrufer, so sind sämtliche globale Daten der aufrufenden Ebenen zugreifbar, es sei denn, sie sind durch lokale Vereinbarungen verdeckt.)

Unterprogramme

- Geltungsbereich/Gültigkeitsbereich(cont)
  - Transfer von Informationen zwischen Geltungsbereichen durch:
    - Parameter
      - Eingangsparameter
      - Ausgangsparameter; Ergebnisparameter
      - beide Richtungen
    - globale Daten
      - im äußersten Geltungsbereich definiert; also programmeinheitsglobal
      - COMMON-Bereiche
        Überall dort gültig, wo definiert; auch über Programmeinheiten hinweg

Unterprogramme

- Parameter
  - Formalparameter (bei der Definition)
  - Aktualparameter (beim Aufruf)

    Zum Zeitpunkt des Aufrufs werden die Formalparameter und die Aktualparameter einander zugeordnet, d.h. eine “Formal-Variable” im Unterprogramm referenziert den gleichen Speicher wie die “Aktual-Variable” im aufrufenden Programm.

  Man unterscheidet
    - positionsgerechte Schreibweise
    - Schlüsselwortschreibweise z. B.

      formalparameter = aktualparameter

    Die gemischte Schreibweise ist möglich. Beginnt man jedoch mit der Schlüsselwortschreibweise, so müssen auch alle folgenden Parameter in Schlüsselwortschreibweise folgen.

Unterprogramme

INTENT-Attribut/Anweisung

- Mit dem INTENT-Attribut kann die Art der Verwendung von Formalparametern spezifiziert werden:
  - Das INTENT(IN)-Attribut besagt, daß der Formalparameter ein Eingabeparameter ist. D.h. dieser Formalparameter darf innerhalb des Unterprogramms nicht redefiniert werden, er darf also z. B. nicht auf der linken Seite einer Zuweisung auftauchen oder Variable einer READ-Anweisung sein.
    Der Aktualparameter kann also sowohl eine Variable als auch eine Konstante sein.
  - Das INTENT(OUT)-Attribut besagt, daß der Formalparameter ein Ausgabeparameter ist. D. h. der Formalparameter muß erst definiert (ein Wert zugewiesen) werden, ehe im Unterprogramm auf ihn zugegriffen wird. Hier **muß** es sich also um eine Variable handeln.

Unterprogramme

INTENT-Attribut/Anweisung

- Mit dem INTENT-Attribut kann die Art der Verwendung von Formalparametern spezifiziert werden (cont):
  - Das INTENT(INOUT)-Attribut besagt, daß der Parameter sowohl Eingabe- als auch Ausgabeparameter sein soll. Auch hier **muß** es sich also um eine Variable handeln.

Wir werden im Rahmen dieses Kurses die Formalparameter mit INTENT-Attributen versehen denn: Keine Angabe = INTENT(INOUT), es sei denn, der Aktual-Parameter ist eine Konstante.

Unterprogramme

- Beispiel:

```
PROGRAM summentest
IMPLICIT NONE
REAL a, b, ergebnis

   Print *, 'Eingabe: summand1 summand2 <r>'
   READ *, a,b

   call summe (a,b,ergebnis)

   PRINT *, 'Ergebnis = ',ergebnis

CONTAINS

      SUBROUTINE summe (summanda, summandb, erg)
      REAL, INTENT(IN) :: summanda, summandb
      REAL, INTENT(out) :: erg
         erg = summanda + summandb
      END SUBROUTINE summe

END PROGRAM summentest
```


Unterprogramme

- FUNKTION

```
[...] FUNCTION fname [(...)]!(FUNCTION-Anweisung)
    [Spezifikationsteil]
    ...
        Funktionsrumpf
        ...
[CONTAINS]
    [interne Unterprogramme]
    ...
END [FUNCTION [fname]]
```

- der wesentliche Unterschied einer Funktion zu einer Subroutine ist, daß der Funktionsname gleichzeitig auch der Name der (lokalen) Ergebnisvariablen ist. Deshalb hat die Funktion einen Typ!
- Der Aufruf einer Funktion kann überall dort stehen, wo auch eine Variable stehen darf; nur auf der linken Seite einer Zuweisung (natürlich) nicht.
- Der Funktionsaufruf darf auch ein separates Statement sein - analog zur Subroutine - allerdings ohne “CALL”; jedoch mit “()”.

Unterprogramme

```
( ~uelinste/tmp/newton_fkt.f90 )

PROGRAM newton
REAL eingabe
! Author: Uwe Linster
! Anfordern und einlesen der Zahl, deren Quadratwurzel
! berechnet werden soll

   PRINT *, ' '

   ! Ausgabe auf das Sichtgerät ohne Zeilenvorschub
   ! Nichtvorrueckende Ausgabe: Kapitel 1.10

   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Zu berechnen ist die Quadratwurzel aus: ')

   READ *, eingabe
   PRINT *, ' '; PRINT *, ' '

   DO WHILE(eingabe <= 0)
      PRINT *, ' Die Eingabedaten entsprechen nicht den &
                 &Anforderungen des Programms (Zahl > 0!):'
      READ *, eingabe
   END DO
```


Unterprogramme

```
!Berechnung

   ergebnis = sqroot(eingabe)

!Ergebnisausgabe

   PRINT *, ' Die Quadratwurzel aus ', eingabe ;PRINT *, ' '
   PRINT *, ' ist ', ergebnis
! Oder auch gleich:
!  PRINT *, ' ist ', sqroot(eingabe)

PRINT *, ' '; PRINT *, ' Ende des Programms'
```

Unterprogramme

```
CONTAINS

REAL FUNCTION sqroot (a)
!Berechnung der Quadratwurzel nach dem Newton'schen Verfahren
!Eingabeparameter:a; Startwert der Iteration: 1
!Iterationsformel:x(i+1) = (x(i)*x(i) + a) / (2*x(i))

   REAL , INTENT(IN) :: a
   REAL x
      !a Größe, deren Quadratwurzel zu berechnen ist
      !x Berechnete Approximation der Quadratwurzel

   !Anfangsapproximation
   x = 1
   !Iteration
   DO WHILE (.NOT. (ABS(x*x - a) <= 1E-4))
      x = (x*x + a) / (2.*x)
   END DO
   sqroot = x
END FUNCTION sqroot

END PROGRAM newton
```


Unterprogramme

- COMMON-BEREICHE
  - Durch COMMON-Bereiche werden globale Speicherbereiche spezifiziert. Variable sind in den Geltungsbereichen direkt zugreifbar, in denen die COMMON-Bereiche spezifiziert wurden; auch über verschiedene Programmeinheiten hinweg.
    Man unterscheidet:
    - einen unbenannten COMMON-Block

```
Syntax:COMMON variablenliste
```

    - und beliebig viele benannte COMMON-Blöcke

```
Syntax:COMMON /blockname/ variablenliste [, /blockname/]
              variablenliste]...
```

    - auch die Mischschreibweise ist möglich:

```
Syntax:COMMON  variablenliste [[,]/[blockname]/
              variablenliste]...
```

Unterprogramme

- COMMON-BEREICHE(cont)
  - Eigenschaften
    - Durch Angabe der Variablen wird eine Speicherfolge festgelegt; um Fehler zu vermeiden sollte man mit “copy” und “paste” arbeiten.
    - Variablennamen in verschiedenen Geltungseinheiten dürfen verschieden sein; das sollte man jedoch nur tun, wenn es dafür auch einen plausiblen Grund gibt. Man greift auf dieselbe globale Speicheradresse zu, das sollte man durch den gleiche Variablennamen auch dokumentieren!
    - Initialisiert werden COMMON-Bereiche mit einer BLOCK DATA-Programmeinheit; jedoch nur die benannten Blöcke.
    - Selbstdefinierte Datentypen müssen das Attribut SEQUENCE besitzen. (Speicherfolge!)

Unterprogramme

- COMMON-Bereich - Beispiel (~uelinste/tmp/common_demo_bubble.f90

```
PROGRAM bubble
IMPLICIT NONE

! Zweck des Programms, Loesungsverfahren:
!
! Die Eintraege in der Datei titel sollen in lexikographisch
! sortierter Reihenfolge gedruckt werden. Das Programm geht
! davon aus, dass die Datei maximal 200 Eintraege besitzt und
! jeder Eintrag aus hoechstens 20 Zeichen besteht.
! Als Sortierverfahren wird 'bubble sort' verwendet.
!-------------------------------------------------------------
! Globale Variable; in COMMON-Bereiche spezifiziert
!    name Feld zur Aufnahme der Eintraege aus der Dateri titel
!    max Anzahl der Eintraege
!  fd Kanalnummer (Filedescriptor; Unit)
!-------------------------------------------------------------
! Initialisierung
!
! Ausgabe der Gesamtueberschrift
!
PRINT *, 'Eintraege der Datei titel lexikograpgisch sortiert:'
```

Unterprogramme

```
! Oeffnen der Datei
!
   CALL oeffnen_datei()

! Einlesen der Daten
!
   CALL einlesen()
!
! Sortieren
!
   CALL bubblesort()
!
! Ausgabe des Ergebnisses
!
   CALL ausgeben_screen()
   CALL ausgeben_datei()
```


Unterprogramme

```
! Die Unterprogramme:

CONTAINS

SUBROUTINE oeffnen_datei()

! Oeffnen der Datei "titel" mit Kanalnummer 10
! Global Data:

INTEGER fd
COMMON /datei/ fd
   fd = 10      ! Entfaelt bei BLOCK-DATA-INIT!!
! Local DATA: none

   open(fd, FORM = 'FORMATTED', file = 'titel')

END SUBROUTINE oeffnen_datei
```

Unterprogramme

```
SUBROUTINE einlesen()

! Einlesen der Eintraege aus der Datei "titel" in das Feld !
"name".
! Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

!Local Data:
INTEGER i

   DO i = 1, 200
      READ(fd, 1000, END = 1010) name(i)
1000  FORMAT (a)
   END DO

1010max = i - 1
   PRINT *, 'Gelesene Datensaetze: ',max

END SUBROUTINE einlesen
```


Unterprogramme

```
SUBROUTINE bubblesort()
! Sortiert nach dem "bubble-sort"-Verfahren
! Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

!Local Data

CHARACTER *20 :: hilf
LOGICAL flag
INTEGER i, j, dl

!  flag   zeigt an, ob bei einem Durchlauf Aenderungen in der
!              Anordnung vorgenommen wurden
!              (.FALSE.: Keine Umordnungen
!               .TRUE. : Es wurden Umordnungen vorgenommen)
!  hilf   Hilfsfeld waehrend des Vertauschens zweier Eintraege
!  dl Anzahl der Sortierschleifen
!
```

Unterprogramme

```
! Der Sortiervorgang

   dl = 0
   DO  i = max, 2, -1
      flag = .FALSE.

      DO j = 1, i - 1
          IF (name(j) > name(j + 1)) THEN
               hilf = name(j)
               name(j) = name(j + 1)
               name(j + 1) = hilf
               flag = .TRUE.
               dl = dl + 1
          END IF
        END DO

        if (.NOT. flag) THEN
         PRINT *, 'Anzahl der Durchlaeufe: ', dl
         EXIT
      END IF

   END DO

END SUBROUTINE bubblesort
```


Unterprogramme

```
SUBROUTINE ausgeben_screen()

!Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max
COMMON name, max

! Local Data:

INTEGER i

   DO i=1,max
      Print *, name(i)
   END DO

END SUBROUTINE ausgeben_screen
```

Unterprogramme

```
SUBROUTINE ausgeben_datei()

!Global Data

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

! Local Data

INTEGER i

   REWIND (fd)
   WRITE ( fd, 1030 )(name(i), i=1, max)
1030FORMAT (a)

END SUBROUTINE ausgeben_datei

END PROGRAM bubble
```


Unterprogramme

- BLOCK DATA
  - Eine BLOCK-DATA-Programmeinheit
    - ist eine nichtausführbare Programmeinheit, sie dient der Initialisierung von Variablen in **benannten** COMMON-Bereichen.
    - COMMON-Blöcke sind globale Speicherbereiche - gültig auch über Programm-Unit-Grenzen hinweg. Eine Initialisierung darf also Programmsystemweit nur einmal erfolgen. Dies könnte innerhalb einer “ausgezeichneten” Programmeinheit erfolgen oder wie hier in einer speziellen Programmeinheit

```
BLOCKDATA [name]
     [Spezifikationsteil]
     .
     .
     .
  END BLOCKDATA [name]
```

Unterprogramme

- BLOCK DATA(cont)

```
Beispiel
BLOCKDATA car

CHARACTER vorderachse *14, hinterachse *10, motor *11
INTEGER anzahl, seriennummer, partnb

COMMON /ersatzteile/ vorderachse, hinterachse, motor
COMMON /attribute/ anzahl, seriennummer, partnb

DATA vorderachse, hinterachse /'Schwingachsen', `Inkaachsen'/
DATA motor /'Dieselmotor'/
DATA anzahl, partnb /250, 4711/ ! seriennummer bleibt
uninitialisiert

END BLOCKDATA car
```

- Da diese Programm-Unit in einer eigenen Datei liegt muß beim Aufruf des Compilers diese Datei als weiterer Parameter angegeben werden:; z. B.

```
f90 hauptprogramm.f90 blockdata.f90 -o programm.out
```


Unterprogramme

- BLOCK DATA
  - Beispiel für Programm “bubble”:

Datei: bubble_blockdata.f90

```
BLOCKDATA bubble

   INTEGER fd

   COMMON /datei/ fd

   DATA fd /10/

END BLOCKDATA bubble
```

- Aufruf des Compilers:

```
f90 common_demo_bubble.f90 bubble_blockdata.f90 -o bubble.out
```

Unterprogramme

FORMELFUNKTION

- eine Formelfunktion ist eine Funktion, die nur aus einer einzigen Anweisung besteht und
- sie ist nur innerhalb einer Geltungseinheit aufrufbar.
- Formelfunktionsdefinition:

```
Syntax:ffname() = ausdruck oder
ffname(formalparameter[,formalparameter]...) =
                                ausdruck
```

- Sowohl 'ausdruck' als auch 'formalparameter' sind skalare Datenobjekte.
- Formelfunktionsaufruf:

```
Syntax:ffname() oder
ffname(formalparameter[,formalparameter]...)
```

- Der Typ der Formelfunktion wird separat (wie bei der Typvereinbarung einer Variablen) vereinbart; dieses gilt auch für die Formalparameter.

Unterprogramme

- Formelfunktion - Beispiel (~uelinste/tmp/capslock_fkt.f90)

```
PROGRAM capslock
IMPLICIT NONE
!
! Das Programm liest Kleinbuchstaben ein, wandelt sie
! in Grossbuchstaben um und druckt diese aus.
!  Variable
!
      CHARACTER, DIMENSION(20) :: zeichenin, zeichenout
      INTEGER i

   CHARACTER gross, klein

!  Formelfunktion:

   gross(klein) = CHAR ( ICHAR(klein) + ICHAR('A') - ICHAR('a'))
```

Unterprogramme

```
!  Einlesen der Zeichen

      PRINT *, ' '
   PRINT *, 'Kleinbuchstaben; max. 20; entweder durch "blank"
getrennt -'
   PRINT *, 'Abschluss <r> oder einzeln durch <r> getrennt!'
   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Eingabe: ')
      READ *, (zeichenin(i), i=1,20)

   DO i = 1,20
      zeichenout(i) = gross(zeichenin(i))
   END DO

   PRINT *, 'Ausgabe:',zeichenout

END PROGRAM capslock
```


Unterprogramme

- Formelfunktion - Beispiel (~uelinste/tmp/kredit_fkt.f90)

```
PROGRAM kredit
IMPLICIT NONE
!
!  Das Programm berechnet die Hoehe der monatlichen Raten fuer
! einen vorgebbaren Kaufpreis bei einzugebender Laufzeit. Raten
!  und Zinsen werden von zwei Jahren bis zu einer angebbaren
!  maximalen Laufzeit in Jahresschritten berechnet. Untere und
!  oebere Grenze des Zinssatzes werden ebenfalls eingelesen.
! ------------------------------------------------------------
!  Variable
!
      REAL preis, mrate, p
      INTEGER n, zmin, zmax, zsatz, zs, year, yearmax
!
! Berechnung der Monatsrate fuer die verschiedenen Zinssaetze
!          formel: r*p/(1-(1+r)**(-n)) mit r = zinssatz/12/100
!  hier als Formelfunktion realisiert:

   mrate(zs,p,n) = ((zs/100./12.) * p) / &
               (1.-(1. + (zs/100./12.))**(-n))
```

Unterprogramme

```
!        preis      Kaufpreis (Eingabe)
!       yearmax   maximale Laufzeit der Ratenzahlung (Eingabe)
!        zmin       minimaler Zinssatz (Eingabe)
!        zmax       maximaler Zinssatz (Eingabe)
!        year       Laufvariable fuer die Laufzeit
!        n          Zahl der Raten
!        zsatz      Laufvariable fuer den Zinssatz
! ------------------------------------------------------------
! Initialisierung
!
!  Einlesen von Kaufpreis, maximaler Laufzeit,
!  minimalem und maximalem Zinssatz
!
      PRINT *, ' '
   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Kaufpreis (REAL): ')
      READ *, preis
   WRITE (*, 2000, ADVANCE='NO')
2000FORMAT ('Laufzeit in Jahren (groesser als 2): ')
      READ *, yearmax
     WRITE (*, 3000, ADVANCE='NO')
3000FORMAT ('minimaler und maximaler Zins (in Prozent
(INTEGER)): ')
      READ *, zmin, zmax; PRINT *, ' '
```


Unterprogramme

```
! Aeussere Schleife

    DO year = 2, yearmax

!  Tabellenueberschrift
        PRINT *, ' '
        PRINT *, ' '
      PRINT *, ' Zinsen fuer eine Laufzeit von ', year, ' Jahren'
        PRINT *, ' '
       PRINT *, ' Zins-  Zahl der     Rate          Gesamtzins'
        PRINT *, ' satz    Raten       (DM)               (DM)'

! Innere Schleife
      DO zsatz = zmin, zmax
         n = 12.*year
          PRINT *, ' ', zsatz, '     ', n, ' ', &
               mrate (zsatz, preis, n) &
               ,' ', n * (mrate (zsatz, preis, n)) -preis
      END DO
    END DO
!
!        Abschlussausgabe
      PRINT *, ' ' ;PRINT *, ' Berechnung abgeschlossen.'
END PROGRAM kredit
```

Unterprogramme

- Rücksprung
  - Es existieren drei alternative Methoden
    - der “normale” Rücksprung bei Erreichung des END-Statements
    - der “Ausnahme” - Rücksprung, wenn aufgrund einer Bedingung die Aufgabe des Unterprogramms erfüllt ist - etwa analog zum EXIT-Aussprung aus einer DO-Schleife

```
Syntax:RETURN
```

Bei beiden Methoden wird mit dem 1. ausführbaren Statement fortgefahren, das dem Unterprogrammaufruf folgt.

- der “Ausnahme” - Rücksprung mit wählbarer Fortsetzungsadresse (label).

```
Syntax:RETURN ganzzahliger_ausdruck
```

- Diese Möglichkeit wird nur der Vollständigkeit erwähnt (Im Falle, daß man mit vorhandenen Programmsystemen umgehen muß). Die Verwendung solcher Konstrukte führt zu undurchsichtigen, nicht mehr wartbaren Programmsystemen.

Unterprogramme

- SAVE-Anweisung
  - Mit dem SAVE-Attribut wird der Wert einer **lokalen** Variablen in einem Geltungsbereich über die “Lebensdauer” eines Aufrufs hinaus “gesichert”. D. h., daß der Wert der Variablen auch beim nächsten Aufruf wieder verfügbar ist. Man spricht auch von “*Static”* - Variablen.
  - Das SAVE-Attribut ist nur erlaubt bei
    - lokalen Variablen
    - benannten COMMON-Blöcken
      (hier ist Vorsicht geboten, ist doch ein COMMON-Block ein globaler Speicherbereich!?!)
  - Das SAVE-Attribut ist insbesondere nicht erlaubt bei
    - einzelnen Variablen, die in einem COMMON-Bereich enthalten sind
    - Formalparametern
    - Funktionsvariablen
    - Namensgruppen (Siehe Kapitel 1.10)

Unterprogramme

SAVE-Anweisung

Attributspezifikation

- gleichzeitig mit der Typvereinbarung

```
REAL, SAVE :: skonto, summe
INTEGER, SAVE :: max
```

- oder getrennt mit einer SAVE-Anweisung

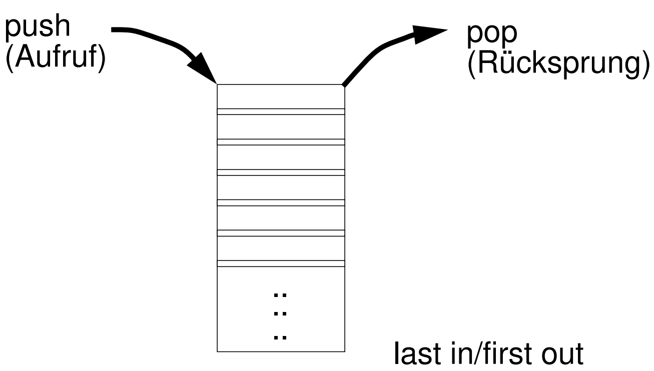
```
REAL skonto, summe
SAVE skonto, summe

INTEGER max
SAVE max
```

Eine SAVE-Anweisung ohne Angabe von Variablennamen bedeutet, das sämtliche lokale Variable in diesem Geltungsbereich diese Spezifikation erhalten.

Rekursive Unterprogramme

Unterprogramme (SUBROUTINE oder FUNCTION) sind dann rekursiv, wenn sie sich selbst direkt oder indirekt aufrufen.

Wenn Unterprogramme das Attribut RECURSION besitzen, muß der Compiler dafür Sorge tragen, daß jeweils bei einem rekursiven Aufruf die lokalen Variablen gerettet und nach dem Rücksprung aus einer tieferen Aufrufebene wieder bereitgestellt werden. Dies ist eine typische Anwendung für einen Kellerspeicher (Stack)!



Rekursive Unterprogramme

- Wann verwendet man rekursive Unterprogramme?
  - Es gibt eine Anzahl von Algorithmen, die "in sich" rekursiv definiert sind. Bei der iterativen Implementierung solcher Algorithmen ist der Programmieraufwand in der Regel beträchtlich und mit einer großen Anzahl von Abfragen behaftet, wie einige Beispiele des Kapitels noch zeigen werden.
  - Als einfaches Beispiel - zum Verständnis - wurde die Berechnung von Fakultäten gewählt, tatsächlich ist es in diesem Fall allerdings effizienter, eine Schleife zu implementieren!

```
PROGRAM recursion
IMPLICIT NONE
INTEGER n, erg
   WRITE (*, FMT = "('n! ; Welches n? : ')", ADVANCE = 'NO')
   READ *, n
   CALL fak (n, erg)
   PRINT *, 'Fakultaet von: ',n ,' = ', erg
   erg = fakfunc (n)
   PRINT *, 'Fakultaet von: ',n ,' = ', erg
CONTAINS
```


Rekursive Unterprogramme

- Rekursive Subroutine

```
RECURSIVE SUBROUTINE fak (fn, ferg)

INTEGER, INTENT (IN) :: fn
INTEGER, INTENT (OUT) :: ferg

   IF (fn == 0) THEN
        ferg = 1
   ELSE
        CALL fak (fn - 1, ferg)
        IF (ferg < 0) THEN
             PRINT *, 'underflow'
             STOP
        END IF
        ferg = fn * ferg
   END IF

END SUBROUTINE fak
```

Rekursive Unterprogramme

- Rekursive Funktion

```
RECURSIVE INTEGER FUNCTION fakfunc (fn) RESULT (ferg)

INTEGER, INTENT (IN) :: fn

   IF (fn == 0) THEN
        ferg = 1
   ELSE
        ferg = fn * fakfunc (fn - 1)
        IF (ferg < 0) THEN
             PRINT *, 'underflow'
             STOP
        END IF
   END IF

END FUNCTION fakfunc

END PROGRAM recursion
```


Unterprogramme

- (Noch) nicht behandelt
  - Parameterübergabe bei nicht skalaren Datengrößen (Siehe Kapitel 1.7- Feldverarbeitung)
  - Module (Siehe Kapitel 1.11 - Module)
  - Externe Unterprogramme (Siehe Kapitel n.b.)
  - Nebeneinganspunkte - ENTRY-Anweisungen (Siehe Kapitel n.b.)
  - Optionale Formalparameter (Siehe Kapitel n.b.)

Programmierung Paralleler Systeme

3. Programmiermodell “Single Programm Multiple Data”

3.1 Fortran95

3.1 .4 Selbstdef. Datentypen (Strukturen)

3.1.5 Zeiger(Pointer)

3.1.6 Unterprogramme

Claus-Uwe Linster

IMMD IV, Lehrstuhl für Betriebssysteme
Universität Erlangen-Nürnberg

linster@informatik.uni-erlangen.de

3.1.4 Selbstdefinierte Typen

▲ Typdefinition

```
TYPE [,attribut]... [::] typname
    komponentendefinition
    [komponentendefinition]
    .
END TYPE [typname]
```

Die Komponentendefinition entspricht weitgehend den Typvereinbarungsanweisungen; jedoch

- keine Initialisierung (es ist eine Definition; keine Deklaration!)
- als Attribute sind nur
  - DIMENSION (mit expliziter Gestalt) und
  - POINTER (Siehe Kapitel 1.5)

  zugelassen.

Selbstdefinierte Typen

Als Attribute der Typdefinition (dann gültig für alle Deklarationen dieses Typs) sind zulässig:

- PRIVATE (Siehe Kapitel 1.11)
- PUBLIC (Siehe Kapitel 1.11)
- SEQUENCE
  - erzwingt eine Speicherordnung gemäß der Komponentenreihenfolge; dies kann hinsichtlich der Speichernutzung Verschnitt bedeuten.
  - Verschiedene Typdefinitionen mit gleichen Komponenten (z. B. in verschiedenen Gültigkeitsbereichen)sind nur dann wirklich gleich, wenn sie das SEQUENCE-Attribut besitzen.
  - Wenn ein selbstdefinierter Typ das SEQUENCE-Attribut besitzt, dann müssen auch alle ggf. vorhandenen Typ-Komponenten, die ihrerseits selbstdefinierte Typen sind, das SEQUENCE-Attribut besitzen.
- Type - Vereinbarungsanweisung

```
Syntax:TYPE (TYPENAME) [,attribut]... [::] dg [,dg]...
```


Selbstdefinierte Typen

▲ Beispiele für Typdefinitionen

```
TYPE:: DATUM
  SEQUENCE
  INTEGER :: tag
  CHARCATER (LEN=3) :: monat
  INTEGER :: jahr
END TYPE DATUM
TYPE :: MITARBEITER
  CHARACTER *20 :: name, vorname
  TYPE (DATUM) geburtsdatum
  REAL gehalt
  REAL lohnsteuer
  LOGICAL verheiratet
  TYPE (MITARBEITER), POINTER :: alphanext ! (Siehe Kapitel 7)
END TYPE MITARBEITER
```

▲ Beispiel für Typvereinbarungen

```
TYPE (MITARBEITER), DIMENSION (100) :: gehaltsliste
TYPE (DATUM), DIMENSION (365) :: kalender
```

Selbstdefinierte Typen

- Strukturkonstante

```
Syntax: TYPNAME (komponentenliste)
  z. B. DATUM (1, 'Jan', 1996)
```

- Zuweisungen
  - Komponentenweise Zuweisung

```
variablenname%komponentenname = wert
```

  - Objektweise Zuweisung

```
variablenname = strukturkonstante
variablenname = variablenname     !gleichen Typs!
```


Selbstdefinierte Typen

- Beispiele

```
kalender(1)  = DATUM(1,'Jan',1996)
kalender(41)%tag = 10
kalender(41)%monat = 'Feb'
kalender(41)%Jahr = 1941

gehaltsliste(35) = MITARBEITER ('Mueller', 'Horst',&
     DATUM(10,'Feb',1941),4523.12,1212.00,.TRUE.,&
     gehaltsliste(25))

gehaltsliste(42)  = gehaltsliste(35)
gehaltsliste(42)%vorname = 'Anton'
gehaltsliste(42)%alphanext = gehaltsliste(35)
! Kapitel 1.5
```

Selbstdefinierte Typen

weitere Beispiele

- Typdefinition

```
TYPE :: ERSATZTEIL
  SEQUENCE
  CHARACTER *10 pkw_typ
  CHARACTER *10 part_nb
END ERSATZTEIL

TYPE :: PKW
  SEQUENCE
  CHARACTER *10 farbe
  INTEGER fahrgestellnummer
  INTEGER zulassungsjahr
  TYPE (ERSATZTEIL), DIMENSION (100) :: vorderachse
END TYPE PKW
```


Selbstdefinierte Typen

weitere Beispiele

- Typvereinbarungen

```
TYPE (PKW), DIMENSION (200) :: lager
```

- Zuweisungen

```
lager(110)%vorderachse(12)%part_nb = '1000G1019'
lager(1)%farbe = 'RAL312'
```

Selbstdefinierte Typen

```
PROGRAM personal
IMPLICIT NONE
INTEGER, PARAMETER :: max = 10
INTEGER, PARAMETER :: fd = 10
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
!  CHARACTER, DIMENSION (:), POINTER :: beurteilung! Kap. 1.5
END TYPE MITARBEITER

TYPE (MITARBEITER), DIMENSION (max) :: pa! Personal_akte
CHARACTER *4 funktion
CHARACTER zeichen
INTEGER i
LOGICAL :: ende = .FALSE.
```


Selbstdefinierte Typen

```
OPEN(fd,FORM='UNFORMATTED',file = 'database.personal')
   ! Oeffnen der Datei
DO WHILE (.NOT. ende)      !Funktionen

   PRINT *, ' '
   PRINT *, 'PRINT *, 'Eingabe der gewünschten Fkt:'; PRINT *,
' '
   PRINT *, 'init = Initialisiere Datenbasis'
   PRINT *, 'read = Lesen der Datenbasis'
   PRINT *, 'einz = Einzel-Eingabe'
   PRINT *, 'eing = Gruppeneingabe'
   PRINT *, 'korr = Korrektur;'
   PRINT *, 'ausg = Ausgabe;'
   PRINT *, 'sumg = Summe Gehaelter;'
   PRINT *, 'suml = Summe Lohnsteuer;'
   PRINT *, 'pers = Personalliste;'
   PRINT *, 'ende = Beenden des Programms'
   PRINT *, ' '
   PRINT *, 'Funktion?'
   READ *, funktion
   PRINT *, ' '
```

Selbstdefinierte Typen

```
SELECT CASE (funktion)
   CASE ('init')
      PRINT *,'Sind Sie sicher? (y)'
      READ *,zeichen
      if (zeichen /= 'y') EXIT
      DO i = 1,max
            pa(i)%name = 'frei'
            pa(i)%vorname = 'frei'
      END DO
   CASE ('read')
      READ(fd) (pa(i), i=1,max)
   CASE ('einz')
      DO i = 1,max
            IF (pa(i)%name == 'frei') THEN
                  Print *,'Name:'
                  READ *, pa(i)%name
                  PRINT *, 'Vorname'
                  READ * , pa(i)%vorname
                  EXIT
            ELSE
                  CYCLE
            END IF
      END DO
```


Selbstdefinierte Typen

```
   CASE ('eing')
      PRINT *, 'Not yet implemented'
   CASE ('korr')
      PRINT *, 'Not yet implemented'
   CASE ('ausg')
      PRINT *, 'Not yet implemented'
   CASE ('sumg')
      PRINT *, 'Not yet implemented'
   CASE ('suml')
      PRINT *, 'Not yet implemented'
   CASE ('pers')
      DO i= 1,max
            PRINT *, pa(i)%name, ';', pa(i)%vorname
      END DO
   CASE ('ende')
      PRINT *, 'Programmende'
      WRITE(fd) (pa(i), i = 1,max)
      ende = .TRUE.
   CASE DEFAULT
      PRINT *, 'Falsche Eingabe!'
   END SELECT
END DO
END PROGRAM personal
```

3.1.5 Zeiger (Pointer)

■ Pointer Definition

- Der Inhalt einer Pointervariablen ist die Adresse eines Objekts auf das der Pointer zeigt.
- Konsequenterweise müßten Pointer - wie in anderen Sprachen (z. B. in C) - ein spezieller Datentyp sein. Da man in Fortran90 nicht direkt auf einen Pointer zugreifen kann (keine Adreßarithmetik!), handelt es sich nur um ein Attribut eines Datentyps.
- In Fortran90 finden Pointer hauptsächlich Verwendung bei der dynamischen Speicherverwaltung (während der Laufzeit des Programms; detailiert in Kapitel 1.8 dargestellt) und beim Aufbau von Datenstrukturen wie verketteten Listen, Baumstrukturen, etc.

Zeiger (Pointer)

■ Pointer-Vereinbarung

```
REAL, POINTER :: zeiger, next, current !Typ und Attribut
! oder
REAL zeiger, next, current                 !Typ
Pointer zeiger, next, current              !Attribut
```

- Aus der Pointerdefinition folgt:
  - das Pointer-Attribut darf nur für Variablen oder Funktionen spezifiziert werden
  - folgende weitere Attribute sind unzulässig: **ALLOCATABLE, EXTERNAL, INTENT, INTRINSIC, PARAMETER, TARGET**
  - ein Pointer darf weder bei einer Typvereinbarung noch mit Hilfe einer DATA-Anweisung oder E/A initialisiert werden

Zeiger (Pointer)

Pointer-Zuweisung/Freigabe

- Will man einem Pointer eine Variable (oder Funktion) zuweisen, so muß für diese Datengröße das TARGET-Attribut spezifiziert sein.
  - Spezifikation des TARGET-Attributs

```
INTEGER, TARGET :: feld, person       ! Typ und Attribut
! oder
INTEGER feld, person    ! Typ
TARGET feld, person     ! Attribut
```

  - Zuweisung

```
pointer => zielvariable
! oder
ALLOCATE (pointer [,pointer]...) [, STAT=statusvariable]
```

  - Freigabe von Pointerzielen

```
pointer => neuezielvariable          !Überschreiben
! oder
DEALLOCATE (pointer [,pointer]...) [, STAT=statusvariable]
```


Zeiger (Pointer)

Pointer-Zuordnungsstati

- zugeordnet:
  - dem Pointer wurde eine Variable mit dem TARGET-Attribut zugewiesen.
  - dem Pointer wurde ein Pointer zugewiesen, der selbst den Status "zugeordnet" besitzt.
  - dem Pointer wurde mit der ALLOCATE-Anweisung ein Speicherbereich zugeordnet
- nicht zugeordnet:
  - dem Pointer wird mit der NULLIFY-Anweisung eine Zuordnung entzogen (Keine damit verbundene Speicherfreigabe!).
  - NULLIFY (pointer [,pointer]...)
  - dem Pointer wird mit der DEALLOCATE-Anweisung ein Speicherbereich entzogen, der ihm mit einer ALLOCATE-Anweisung zugeordnet wurde.
  - dem Pointer wird ein Pointer mit dem Status "nicht zugeordnet" zugewiesen.

Zeiger (Pointer)

- Pointer-Zuordnungsstati(cont)
  - undefiniert:
    - dem Pointer wurde noch kein Ziel zugewiesen (weder '=>' noch **`ALLOCATE`**)
    - das zugeordnete Ziel wurde auf “anderer Weise” entfernt; z. B. ein **`DEALLOCATE`** auf einen Zeiger mit dem gleichen Ziel
    - dem Pointer wird ein Pointer zugewiesen mit dem Staus “undefiniert”
    - wenn das Pointerziel bei Ausführung einer **`END`**- oder **`RETURN`**-Anweisung undefiniert wird. (Kapitel 1.6)

Zeiger (Pointer)

- ASSOCIATED-Function
  - Intrinsic (vordefinierte) Funktion

    `Syntax: ASSOCIATED (pointervar [,targetvar]`

    - liefert den Wert **`.TRUE.`** beim Aufruf

      `ASSOCIATED (pointervar)`

    wenn **`pointervar`** irgendeinem Ziel zugeordnet ist und beim Aufruf

      `ASSOCIATED (pointervar, targetvar)`

    wenn **`pointervar targetvar`** zugeordnet ist.

    - sonst den Wert **`.FALSE.`**

Beispiele

- zu Datenstrukturen, verkettete Listen, Baumstrukturen, etc
  siehe auch Vorl. mit Übungen : "Programm- und Datenstrukturen"
- zu **`ALLOCATE, DEALLOCATE, NULLIFY`**
  siehe auch Feldverarbeitung (Dynamische Speicherverwaltung) Kap. 1.8

Zeiger (Pointer)

```
PROGRAM queue

! Warteschlangen-Demo-Programm adaptiert aus:
! Cooper Redwine; "Upgrading to FORTRAN 90", Springer-Verlag,
! 1995
!
! Queue - Policy: first in - first out (Hier zunaechst nur
! Eingabe)

IMPLICIT NONE
TYPE :: QUEUE_ELEMENT
   CHARACTER (20) :: data
   TYPE (QUEUE_ELEMENT), POINTER :: next
END TYPE

! Zeiger auf des erste, letzte, aktuelle Element: first, last,
! this

TYPE (QUEUE_ELEMENT), POINTER :: first, last, this

CHARACTER(20) :: buffer
CHARACTER *4 :: funktion
LOGICAL :: ende = .FALSE.
INTEGER :: io_stat = 1
```


Zeiger (Pointer)

```
   DO WHILE (.NOT. ende)         !Funktionen
   PRINT *, ' '
   PRINT *, 'Eingabe der gewünschten Fkt:'
   PRINT *, ' '
   PRINT *, 'init = Initialisiere Queue'
   PRINT *, 'read = Lese Queue'
   PRINT *, 'erze = Erzeuge Elemente'
   PRINT *, 'dele = Loesche 1. Element'
   PRINT *, 'ende = Beenden des Programms'
   PRINT *, ' '
   PRINT *, 'Funktion:'
   READ *, funktion
   PRINT *, ' '
```

Zeiger (Pointer)

```
SELECT CASE (funktion)
   CASE ('init')
      NULLIFY (first, last)! Initialisiert: leere Queue
   CASE ('read')

      ! Durchlaufe die Warteschlange und drucke Inhalte als
      ! Liste aus.

      this => first           ! aktuelles Element gleich
                              ! 1. Element
      PRINT *, 'Ausgabe'; PRINT *, ' '
      DO WHILE (ASSOCIATED (this))
            PRINT *, this%data
            this => this%next ! aktuelles Element gleich
                              ! naechstes Element
      END DO
```


Zeiger (Pointer)

```
CASE ('erze')
   PRINT *, 'Eingabe:'
   DO WHILE (io_stat >= 0)
      READ (*, "(A)", IOSTAT = io_stat) buffer

      IF (.NOT. ASSOCIATED (first)) THEN

      ! Queue ist leer; fordere den Speicher fuer das
      ! 1. Element an

            ALLOCATE (first)
            last => first     ! Beide Zeiger zeigen aufs
                              ! gleiche(erste)Element
      ELSE
      ! Queue ist nicht leer, neues Element gleich
      ! letztes Element

            ALLOCATE (last%next)
            last => last%next
      END IF
      last%data = buffer
      NULLIFY (last%next)     ! Letztes Element hat
                              ! keinen Nachfolger
   END DO
```

Zeiger (Pointer)

```
      CASE ('dele')
         PRINT *, 'Not yet implemented'
      CASE ('ende')
         PRINT *, 'Programmende'
         ende = .TRUE.
      CASE DEFAULT
         PRINT *, 'Falsche Eingabe!'
      END SELECT
END DO

END PROGRAM queue
```


3.1.6 Unterprogramme

- Definitionen/Terminologie
  - Programmeinheiten (programunits) sind die Bausteine eines ausführbaren Programms
    - Hauptprogramm (genau einmal)
    - ext. Unterprogramme
    - Module
    - Blockdatenprogrammeinheiten
  - Sie enden mit der END-Anweisung, liegen in separaten Dateien, sind getrennt übersetzbar, jedoch bis auf das Hauptprogramm nicht direkt ausführbar.
  - Ein Hauptprogramm ist ausführbar, wenn es keine offenen 'Externreferenzen' gibt - ausgenommen Bibliotheksreferenzen, wie z.b. SIN(x), COS(x), etc.
    Diese Eigenschaft haben unsere bisher geschriebenen Programme.

Unterprogramme

- Die weiteren Programmeinheiten sind nur dann ausführbar (getrennt testbar), wenn ein entsprechendes Hauptprogramm als “Testrahmen” zugebunden wird.
- Nach der Übersetzung sind diese Einheiten zunächst separate Bindemodule; sogenannte ".o-files"

■ Unterprogramme werden unterschieden

- nach ihrem “Definitionsort”
  - extern
  - eingebettet
  - vordefiniert in Bibliothek
- und ihrer Art
  - Subroutine
  - Funktion
  - Formelfunktion

Unterprogramme

■ Subroutine

- Subroutinendefinition

```
[...] SUBROUTINE sname [(...)]        !(SUBROUTINE-Anweisung)
    [Spezifikationsteil]
    ...
        Subroutinenrumpf
    ...
[CONTAINS]
    [interne Unterprogramme]
    ...
END [SUBROUTINE [sname]]
```

- Beispiele:

```
SUBROUTINE sname()                     !(leere Parameterliste)
SUBROUTINE sname (formalparameter[,formalparameter] ...)
```

- Subroutinenaufruf

```
CALL sname
CALL sname()
CALL sname(Aktualparameter[,Aktualparameter]...)
```

Unterprogramme

- Beispiel - eingebettetes Unterprogramm (CONTAINS)

```
PROGRAM summentest
IMPLICIT NONE
REAL a, b, ergebnis

   Print *, 'Eingabe: summand1 summand2 <r>'
   READ *, a,b

   call summe (a,b,ergebnis)

   PRINT *, 'Ergebnis = ',ergebnis

CONTAINS

      SUBROUTINE summe (summanda, summandb, erg)
      REAL summanda, summandb, erg
         erg = summanda + summandb
      END SUBROUTINE summe

END PROGRAM summentest
```


Unterprogramme

- Geltungsbereich/Gültigkeitsbereich
  - sagt aus, in welchen Bereichen eine Variable zugreifbar ist
    - lokal Variable sind nur gültig innerhalb eines Geltungsbereichs z. B. in einem Unterprogramm
    - globale Variable sind in der gesamten Programmeinheit, in der sie definiert wurden, zugreifbar, falls sie nicht durch eine lokale Definition “verdeckt” werden.
      (“Blickt” man aus der Aufrufschachtelung zurück in Richtung Aufrufer, so sind sämtliche globale Daten der aufrufenden Ebenen zugreifbar, es sei denn, sie sind durch lokale Vereinbarungen verdeckt.)

Unterprogramme

- Geltungsbereich/Gültigkeitsbereich(cont)
  - Transfer von Informationen zwischen Geltungsbereichen durch:
    - Parameter
      - Eingangsparameter
      - Ausgangsparameter; Ergebnisparameter
      - beide Richtungen
    - globale Daten
      - im äußersten Geltungsbereich definiert; also programmeinheitsglobal
      - COMMON-Bereiche
        Überall dort gültig, wo definiert; auch über Programmeinheiten hinweg

Unterprogramme

- Parameter
  - Formalparameter (bei der Definition)
  - Aktualparameter (beim Aufruf)

    Zum Zeitpunkt des Aufrufs werden die Formalparameter und die Aktualparameter einander zugeordnet, d.h. eine “Formal-Variable” im Unterprogramm referenziert den gleichen Speicher wie die “Aktual-Variable” im aufrufenden Programm.

  Man unterscheidet

    - positionsgerechte Schreibweise
    - Schlüsselwortschreibweise z. B.

      formalparameter = aktualparameter

    Die gemischte Schreibweise ist möglich. Beginnt man jedoch mit der Schlüsselwortschreibweise, so müssen auch alle folgenden Parameter in Schlüsselwortschreibweise folgen.

Unterprogramme

INTENT-Attribut/Anweisung

- Mit dem INTENT-Attribut kann die Art der Verwendung von Formalparametern spezifiziert werden:
  - Das INTENT(IN)-Attribut besagt, daß der Formalparameter ein Eingabeparameter ist. D.h. dieser Formalparameter darf innerhalb des Unterprogramms nicht redefiniert werden, er darf also z. B. nicht auf der linken Seite einer Zuweisung auftauchen oder Variable einer READ-Anweisung sein.
    Der Aktualparameter kann also sowohl eine Variable als auch eine Konstante sein.
  - Das INTENT(OUT)-Attribut besagt, daß der Formalparameter ein Ausgabeparameter ist. D. h. der Formalparameter muß erst definiert (ein Wert zugewiesen) werden, ehe im Unterprogramm auf ihn zugegriffen wird. Hier **muß** es sich also um eine Variable handeln.

Unterprogramme

INTENT-Attribut/Anweisung

- Mit dem INTENT-Attribut kann die Art der Verwendung von Formalparametern spezifiziert werden (cont):
  - Das INTENT(INOUT)-Attribut besagt, daß der Parameter sowohl Eingabe- als auch Ausgabeparameter sein soll. Auch hier **muß** es sich also um eine Variable handeln.

Wir werden im Rahmen dieses Kurses die Formalparameter mit INTENT-Attributen versehen denn: Keine Angabe = INTENT(INOUT), es sei denn, der Aktual-Parameter ist eine Konstante.

Unterprogramme

- Beispiel:

```
PROGRAM summentest
IMPLICIT NONE
REAL a, b, ergebnis

   Print *, 'Eingabe: summand1 summand2 <r>'
   READ *, a,b

   call summe (a,b,ergebnis)

   PRINT *, 'Ergebnis = ',ergebnis

CONTAINS

      SUBROUTINE summe (summanda, summandb, erg)
      REAL, INTENT(IN) :: summanda, summandb
      REAL, INTENT(out) :: erg
         erg = summanda + summandb
      END SUBROUTINE summe

END PROGRAM summentest
```


Unterprogramme

- FUNKTION

```
     [...] FUNCTION fname [(...)]!(FUNCTION-Anweisung)
        [Spezifikationsteil]
        ...
           Funktionsrumpf
           ...
     [CONTAINS]
        [interne Unterprogramme]
        ...
     END [FUNCTION [fname]]
```

  - der wesentliche Unterschied einer Funktion zu einer Subroutine ist, daß der Funktionsname gleichzeitig auch der Name der (lokalen) Ergebnisvariablen ist. Deshalb hat die Funktion einen Typ!
  - Der Aufruf einer Funktion kann überall dort stehen, wo auch eine Variable stehen darf; nur auf der linken Seite einer Zuweisung (natürlich) nicht.
  - Der Funktionsaufruf darf auch ein separates Statement sein - analog zur Subroutine - allerdings ohne “CALL”; jedoch mit “()”.

Unterprogramme

```
( ~uelinste/tmp/newton_fkt.f90 )

PROGRAM newton
REAL eingabe
! Author: Uwe Linster
! Anfordern und einlesen der Zahl, deren Quadratwurzel
! berechnet werden soll

   PRINT *, ' '

   ! Ausgabe auf das Sichtgerät ohne Zeilenvorschub
   ! Nichtvorrueckende Ausgabe: Kapitel 1.10

   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Zu berechnen ist die Quadratwurzel aus: ')

   READ *, eingabe
   PRINT *, ' '; PRINT *, ' '

   DO WHILE(eingabe <= 0)
      PRINT *, ' Die Eingabedaten entsprechen nicht den &
                 &Anforderungen des Programms (Zahl > 0!):'
      READ *, eingabe
   END DO
```


Unterprogramme

```
!Berechnung

   ergebnis = sqroot(eingabe)

!Ergebnisausgabe

   PRINT *, ' Die Quadratwurzel aus ', eingabe ;PRINT *, ' '
   PRINT *, ' ist ', ergebnis
! Oder auch gleich:
!  PRINT *, ' ist ', sqroot(eingabe)

PRINT *, ' '; PRINT *, ' Ende des Programms'
```

Unterprogramme

```
CONTAINS

REAL FUNCTION sqroot (a)
!Berechnung der Quadratwurzel nach dem Newton'schen Verfahren
!Eingabeparameter:a; Startwert der Iteration: 1
!Iterationsformel:x(i+1) = (x(i)*x(i) + a) / (2*x(i))

   REAL , INTENT(IN) :: a
   REAL x
      !a Größe, deren Quadratwurzel zu berechnen ist
      !x Berechnete Approximation der Quadratwurzel

   !Anfangsapproximation
   x = 1
   !Iteration
   DO WHILE (.NOT. (ABS(x*x - a) <= 1E-4))
      x = (x*x + a) / (2.*x)
   END DO
   sqroot = x
END FUNCTION sqroot

END PROGRAM newton
```


Unterprogramme

- COMMON-BEREICHE
  - Durch COMMON-Bereiche werden globale Speicherbereiche spezifiziert. Variable sind in den Geltungsbereichen direkt zugreifbar, in denen die COMMON-Bereiche spezifiziert wurden; auch über verschiedene Programmeinheiten hinweg.
    Man unterscheidet:
    - einen unbenannten COMMON-Block

```
Syntax:COMMON variablenliste
```

    - und beliebig viele benannte COMMON-Blöcke

```
Syntax:COMMON /blockname/ variablenliste [, /blockname/]
              variablenliste]...
```

    - auch die Mischschreibweise ist möglich:

```
Syntax:COMMON  variablenliste [[,]/[blockname]/
              variablenliste]...
```

Unterprogramme

- COMMON-BEREICHE(cont)
  - Eigenschaften
    - Durch Angabe der Variablen wird eine Speicherfolge festgelegt; um Fehler zu vermeiden sollte man mit “copy” und “paste” arbeiten.
    - Variablennamen in verschiedenen Geltungseinheiten dürfen verschieden sein; das sollte man jedoch nur tun, wenn es dafür auch einen plausiblen Grund gibt. Man greift auf dieselbe globale Speicheradresse zu, das sollte man durch den gleiche Variablennamen auch dokumentieren!
    - Initialisiert werden COMMON-Bereiche mit einer BLOCK DATA-Programmeinheit; jedoch nur die benannten Blöcke.
    - Selbstdefinierte Datentypen müssen das Attribut SEQUENCE besitzen. (Speicherfolge!)

Unterprogramme

- COMMON-Bereich - Beispiel (~uelinste/tmp/common_demo_bubble.f90

```
PROGRAM bubble
IMPLICIT NONE

! Zweck des Programms, Loesungsverfahren:
!
! Die Eintraege in der Datei titel sollen in lexikographisch
! sortierter Reihenfolge gedruckt werden. Das Programm geht
! davon aus, dass die Datei maximal 200 Eintraege besitzt und
! jeder Eintrag aus hoechstens 20 Zeichen besteht.
! Als Sortierverfahren wird 'bubble sort' verwendet.
!-------------------------------------------------------------
! Globale Variable; in COMMON-Bereiche spezifiziert
!    name Feld zur Aufnahme der Eintraege aus der Dateri titel
!    max Anzahl der Eintraege
!  fd Kanalnummer (Filedescriptor; Unit)
!-------------------------------------------------------------
! Initialisierung
!
! Ausgabe der Gesamtueberschrift
!
PRINT *, 'Eintraege der Datei titel lexikograpgisch sortiert:'
```

Unterprogramme

```
! Oeffnen der Datei
!
   CALL oeffnen_datei()

! Einlesen der Daten
!
   CALL einlesen()
!
! Sortieren
!
   CALL bubblesort()
!
! Ausgabe des Ergebnisses
!
   CALL ausgeben_screen()
   CALL ausgeben_datei()
```


Unterprogramme

```
! Die Unterprogramme:

CONTAINS

SUBROUTINE oeffnen_datei()

! Oeffnen der Datei "titel" mit Kanalnummer 10
! Global Data:

INTEGER fd
COMMON /datei/ fd
   fd = 10      ! Entfaelt bei BLOCK-DATA-INIT!!
! Local DATA: none

   open(fd, FORM = 'FORMATTED', file = 'titel')

END SUBROUTINE oeffnen_datei
```

Unterprogramme

```
SUBROUTINE einlesen()

! Einlesen der Eintraege aus der Datei "titel" in das Feld !
"name".
! Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

!Local Data:
INTEGER i

   DO i = 1, 200
      READ(fd, 1000, END = 1010) name(i)
1000  FORMAT (a)
   END DO

1010max = i - 1
   PRINT *, 'Gelesene Datensaetze: ',max

END SUBROUTINE einlesen
```


Unterprogramme

```
SUBROUTINE bubblesort()
! Sortiert nach dem "bubble-sort"-Verfahren
! Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

!Local Data

CHARACTER *20 :: hilf
LOGICAL flag
INTEGER i, j, dl

!  flag   zeigt an, ob bei einem Durchlauf Aenderungen in der
!              Anordnung vorgenommen wurden
!              (.FALSE.: Keine Umordnungen
!               .TRUE. : Es wurden Umordnungen vorgenommen)
!  hilf   Hilfsfeld waehrend des Vertauschens zweier Eintraege
!  dl Anzahl der Sortierschleifen
!
```

Unterprogramme

```
! Der Sortiervorgang

   dl = 0
   DO  i = max, 2, -1
      flag = .FALSE.

      DO j = 1, i - 1
          IF (name(j) > name(j + 1)) THEN
               hilf = name(j)
               name(j) = name(j + 1)
               name(j + 1) = hilf
               flag = .TRUE.
               dl = dl + 1
          END IF
        END DO

        if (.NOT. flag) THEN
         PRINT *, 'Anzahl der Durchlaeufe: ', dl
         EXIT
      END IF

   END DO

END SUBROUTINE bubblesort
```


Unterprogramme

```
SUBROUTINE ausgeben_screen()

!Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max
COMMON name, max

! Local Data:

INTEGER i

   DO i=1,max
      Print *, name(i)
   END DO

END SUBROUTINE ausgeben_screen
```

Unterprogramme

```
SUBROUTINE ausgeben_datei()

!Global Data

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

! Local Data

INTEGER i

   REWIND (fd)
   WRITE ( fd, 1030 )(name(i), i=1, max)
1030FORMAT (a)

END SUBROUTINE ausgeben_datei

END PROGRAM bubble
```


Unterprogramme

- BLOCK DATA
  - Eine BLOCK-DATA-Programmeinheit
    - ist eine nichtausführbare Programmeinheit, sie dient der Initialisierung von Variablen in **benannten** COMMON-Bereichen.
    - COMMON-Blöcke sind globale Speicherbereiche - gültig auch über Programm-Unit-Grenzen hinweg. Eine Initialisierung darf also Programmsystemweit nur einmal erfolgen. Dies könnte innerhalb einer “ausgezeichneten” Programmeinheit erfolgen oder wie hier in einer speziellen Programmeinheit

```
BLOCKDATA [name]
     [Spezifikationsteil]
     .
     .
     .
  END BLOCKDATA [name]
```

Unterprogramme

- BLOCK DATA(cont)

```
Beispiel
BLOCKDATA car

CHARACTER vorderachse *14, hinterachse *10, motor *11
INTEGER anzahl, seriennummer, partnb

COMMON /ersatzteile/ vorderachse, hinterachse, motor
COMMON /attribute/ anzahl, seriennummer, partnb

DATA vorderachse, hinterachse /'Schwingachsen', `Inkaachsen'/
DATA motor /'Dieselmotor'/
DATA anzahl, partnb /250, 4711/ ! seriennummer bleibt
uninitialisiert

END BLOCKDATA car
```

- Da diese Programm-Unit in einer eigenen Datei liegt muß beim Aufruf des Compilers diese Datei als weiterer Parameter angegeben werden:; z. B.

```
f90 hauptprogramm.f90 blockdata.f90 -o programm.out
```


Unterprogramme

- BLOCK DATA
  - Beispiel für Programm “bubble”:

Datei: bubble_blockdata.f90

```
BLOCKDATA bubble

   INTEGER fd

   COMMON /datei/ fd

   DATA fd /10/

END BLOCKDATA bubble
```

- Aufruf des Compilers:

```
f90 common_demo_bubble.f90 bubble_blockdata.f90 -o bubble.out
```

Unterprogramme

FORMELFUNKTION

- eine Formelfunktion ist eine Funktion, die nur aus einer einzigen Anweisung besteht und
- sie ist nur innerhalb einer Geltungseinheit aufrufbar.
- Formelfunktionsdefinition:

```
Syntax:ffname() = ausdruck oder
ffname(formalparameter[,formalparameter]...) =
                         ausdruck
```

- Sowohl 'ausdruck' als auch 'formalparameter' sind skalare Datenobjekte.
- Formelfunktionsaufruf:

```
Syntax:ffname() oder
ffname(formalparameter[,formalparameter]...)
```

- Der Typ der Formelfunktion wird separat (wie bei der Typvereinbarung einer Variablen) vereinbart; dieses gilt auch für die Formalparameter.

Unterprogramme

- Formelfunktion - Beispiel (~uelinste/tmp/capslock_fkt.f90)

```
PROGRAM capslock
IMPLICIT NONE
!
! Das Programm liest Kleinbuchstaben ein, wandelt sie
! in Grossbuchstaben um und druckt diese aus.
!  Variable
!
      CHARACTER, DIMENSION(20) :: zeichenin, zeichenout
      INTEGER i

   CHARACTER gross, klein

!  Formelfunktion:

   gross(klein) = CHAR ( ICHAR(klein) + ICHAR('A') - ICHAR('a'))
```

Unterprogramme

```
!  Einlesen der Zeichen

      PRINT *, ' '
   PRINT *, 'Kleinbuchstaben; max. 20; entweder durch "blank"
getrennt -'
   PRINT *, 'Abschluss <r> oder einzeln durch <r> getrennt!'
   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Eingabe: ')
      READ *, (zeichenin(i), i=1,20)

   DO i = 1,20
      zeichenout(i) = gross(zeichenin(i))
   END DO

   PRINT *, 'Ausgabe:',zeichenout

END PROGRAM capslock
```


Unterprogramme

- Formelfunktion - Beispiel (~uelinste/tmp/kredit_fkt.f90)

```
PROGRAM kredit
IMPLICIT NONE
!
!  Das Programm berechnet die Hoehe der monatlichen Raten fuer
! einen vorgebbaren Kaufpreis bei einzugebender Laufzeit. Raten
!  und Zinsen werden von zwei Jahren bis zu einer angebbaren
!  maximalen Laufzeit in Jahresschritten berechnet. Untere und
!  oebere Grenze des Zinssatzes werden ebenfalls eingelesen.
! ------------------------------------------------------------
!  Variable
!
      REAL preis, mrate, p
      INTEGER n, zmin, zmax, zsatz, zs, year, yearmax
!
! Berechnung der Monatsrate fuer die verschiedenen Zinssaetze
!          formel: r*p/(1-(1+r)**(-n)) mit r = zinssatz/12/100
!  hier als Formelfunktion realisiert:

   mrate(zs,p,n) = ((zs/100./12.) * p) / &
               (1.-(1. + (zs/100./12.))**(-n))
```

Unterprogramme

```
!        preis      Kaufpreis (Eingabe)
!       yearmax   maximale Laufzeit der Ratenzahlung (Eingabe)
!        zmin       minimaler Zinssatz (Eingabe)
!        zmax       maximaler Zinssatz (Eingabe)
!        year       Laufvariable fuer die Laufzeit
!        n          Zahl der Raten
!        zsatz      Laufvariable fuer den Zinssatz
! ------------------------------------------------------------
! Initialisierung
!
!  Einlesen von Kaufpreis, maximaler Laufzeit,
!  minimalem und maximalem Zinssatz
!
      PRINT *, ' '
   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Kaufpreis (REAL): ')
      READ *, preis
   WRITE (*, 2000, ADVANCE='NO')
2000FORMAT ('Laufzeit in Jahren (groesser als 2): ')
      READ *, yearmax
     WRITE (*, 3000, ADVANCE='NO')
3000FORMAT ('minimaler und maximaler Zins (in Prozent
(INTEGER)): ')
      READ *, zmin, zmax; PRINT *, ' '
```


Unterprogramme

```
! Aeussere Schleife

    DO year = 2, yearmax

!  Tabellenueberschrift
        PRINT *, ' '
        PRINT *, ' '
      PRINT *, ' Zinsen fuer eine Laufzeit von ', year, ' Jahren'
        PRINT *, ' '
       PRINT *, ' Zins-  Zahl der     Rate          Gesamtzins'
        PRINT *, ' satz    Raten        (DM)               (DM)'

! Innere Schleife
      DO zsatz = zmin, zmax
         n = 12.*year
          PRINT *, ' ', zsatz, '      ', n, ' ', &
               mrate (zsatz, preis, n) &
               ,' ', n * (mrate (zsatz, preis, n)) -preis
      END DO
    END DO
!
!        Abschlussausgabe
      PRINT *, ' ' ;PRINT *, ' Berechnung abgeschlossen.'
END PROGRAM kredit
```

Unterprogramme

- Rücksprung

- Es existieren drei alternative Methoden
  - der “normale” Rücksprung bei Erreichung des END-Statements
  - der “Ausnahme” - Rücksprung, wenn aufgrund einer Bedingung die Aufgabe des Unterprogramms erfüllt ist - etwa analog zum EXIT-Aussprung aus einer DO-Schleife

```
Syntax:RETURN
```

Bei beiden Methoden wird mit dem 1. ausführbaren Statement fortgefahren, das dem Unterprogrammaufruf folgt.

  - der “Ausnahme” - Rücksprung mit wählbarer Fortsetzungsadresse (label).

```
Syntax:RETURN ganzzahliger_ausdruck
```

  - Diese Möglichkeit wird nur der Vollständigkeit erwähnt (Im Falle, daß man mit vorhandenen Programmsystemen umgehen muß). Die Verwendung solcher Konstrukte führt zu undurchsichtigen, nicht mehr wartbaren Programmsystemen.

Unterprogramme

- SAVE-Anweisung

- Mit dem SAVE-Attribut wird der Wert einer **lokalen** Variablen in einem Geltungsbereich über die “Lebensdauer” eines Aufrufs hinaus “gesichert”. D. h., daß der Wert der Variablen auch beim nächsten Aufruf wieder verfügbar ist. Man spricht auch von “*Static”* - Variablen.

- Das SAVE-Attribut ist nur erlaubt bei
  - lokalen Variablen
  - benannten COMMON-Blöcken
    (hier ist Vorsicht geboten, ist doch ein COMMON-Block ein globaler Speicherbereich!?!)

- Das SAVE-Attribut ist insbesondere nicht erlaubt bei
  - einzelnen Variablen, die in einem COMMON-Bereich enthalten sind
  - Formalparametern
  - Funktionsvariablen
  - Namensgruppen (Siehe Kapitel 1.10)

Unterprogramme

SAVE-Anweisung

Attributspezifikation

- gleichzeitig mit der Typvereinbarung

```
REAL, SAVE :: skonto, summe
INTEGER, SAVE :: max
```

- oder getrennt mit einer SAVE-Anweisung

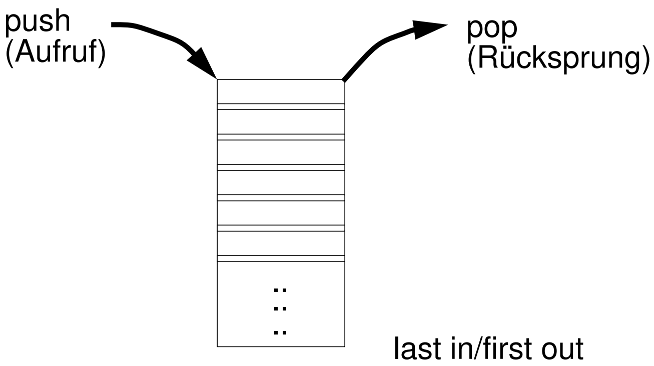
```
REAL skonto, summe
SAVE skonto, summe

INTEGER max
SAVE max
```

Eine SAVE-Anweisung ohne Angabe von Variablennamen bedeutet, das sämtliche lokale Variable in diesem Geltungsbereich diese Spezifikation erhalten.

Rekursive Unterprogramme

Unterprogramme (SUBROUTINE oder FUNCTION) sind dann rekursiv, wenn sie sich selbst direkt oder indirekt aufrufen.

Wenn Unterprogramme das Attribut RECURSION besitzen, muß der Compiler dafür Sorge tragen, daß jeweils bei einem rekursiven Aufruf die lokalen Variablen gerettet und nach dem Rücksprung aus einer tieferen Aufrufebene wieder bereitgestellt werden. Dies ist eine typische Anwendung für einen Kellerspeicher (Stack)!



Rekursive Unterprogramme

- Wann verwendet man rekursive Unterprogramme?
  - Es gibt eine Anzahl von Algorithmen, die "in sich" rekursiv definiert sind. Bei der iterativen Implementierung solcher Algorithmen ist der Programmieraufwand in der Regel beträchtlich und mit einer großen Anzahl von Abfragen behaftet, wie einige Beispiele des Kapitels noch zeigen werden.
  - Als einfaches Beispiel - zum Verständnis - wurde die Berechnung von Fakultäten gewählt, tatsächlich ist es in diesem Fall allerdings effizienter, eine Schleife zu implementieren!

```
PROGRAM recursion
IMPLICIT NONE
INTEGER n, erg
   WRITE (*, FMT = "('n! ; Welches n? : ')", ADVANCE = 'NO')
   READ *, n
   CALL fak (n, erg)
   PRINT *, 'Fakultaet von: ',n ,' = ', erg
   erg = fakfunc (n)
   PRINT *, 'Fakultaet von: ',n ,' = ', erg
CONTAINS
```


Rekursive Unterprogramme

- Rekursive Subroutine

```
RECURSIVE SUBROUTINE fak (fn, ferg)

INTEGER, INTENT (IN) :: fn
INTEGER, INTENT (OUT) :: ferg

   IF (fn == 0) THEN
         ferg = 1
   ELSE
         CALL fak (fn - 1, ferg)
         IF (ferg < 0) THEN
               PRINT *, 'underflow'
               STOP
         END IF
         ferg = fn * ferg
   END IF

END SUBROUTINE fak
```

Rekursive Unterprogramme

▲ Rekursive Funktion

```
RECURSIVE INTEGER FUNCTION fakfunc (fn) RESULT (ferg)

INTEGER, INTENT (IN) :: fn

   IF (fn == 0) THEN
        ferg = 1
   ELSE
        ferg = fn * fakfunc (fn - 1)
        IF (ferg < 0) THEN
             PRINT *, 'underflow'
             STOP
        END IF
   END IF

END FUNCTION fakfunc

END PROGRAM recursion
```


Unterprogramme

■ (Noch) nicht behandelt

- Parameterübergabe bei nicht skalaren Datengrößen (Siehe Kapitel 1.7- Feldverarbeitung)
- Module (Siehe Kapitel 1.11 - Module)
- Externe Unterprogramme (Siehe Kapitel n.b.)
- Nebeneinganspunkte - ENTRY-Anweisungen (Siehe Kapitel n.b.)
- Optionale Formalparameter (Siehe Kapitel n.b.)

Programmierung Paralleler Systeme

3. Programmiermodell “Single Programm Multiple Data”

3.1 Fortran95

3.1 .4 Selbstdef. Datentypen (Strukturen)

3.1.5 Zeiger(Pointer)

3.1.6 Unterprogramme

Claus-Uwe Linster

IMMD IV, Lehrstuhl für Betriebssysteme
Universität Erlangen-Nürnberg

linster@informatik.uni-erlangen.de

3.1.4 Selbstdefinierte Typen

▲ Typdefinition

```
TYPE [,attribut]... [::] typname
   komponentendefinition
   [komponentendefinition]
   .
END TYPE [typname]
```

Die Komponentendefinition entspricht weitgehend den Typvereinbarungsanweisungen; jedoch

- keine Initialisierung (es ist eine Definition; keine Deklaration!)
- als Attribute sind nur
  - DIMENSION (mit expliziter Gestalt) und
  - POINTER (Siehe Kapitel 1.5)

  zugelassen.

Selbstdefinierte Typen

Als Attribute der Typdefinition (dann gültig für alle Deklarationen dieses Typs) sind zulässig:

- PRIVATE (Siehe Kapitel 1.11)
- PUBLIC (Siehe Kapitel 1.11)
- SEQUENCE
  - erzwingt eine Speicherordnung gemäß der Komponentenreihenfolge; dies kann hinsichtlich der Speichernutzung Verschnitt bedeuten.
  - Verschiedene Typdefinitionen mit gleichen Komponenten (z. B. in verschiedenen Gültigkeitsbereichen)sind nur dann wirklich gleich, wenn sie das SEQUENCE-Attribut besitzen.
  - Wenn ein selbstdefinierter Typ das SEQUENCE-Attribut besitzt, dann müssen auch alle ggf. vorhandenen Typ-Komponenten, die ihrerseits selbstdefinierte Typen sind, das SEQUENCE-Attribut besitzen.
- Type - Vereinbarungsanweisung

```
Syntax:TYPE (TYPENAME) [,attribut]... [::] dg [,dg]...
```


Selbstdefinierte Typen

▲ Beispiele für Typdefinitionen

```
TYPE:: DATUM
  SEQUENCE
  INTEGER :: tag
  CHARCATER (LEN=3) :: monat
  INTEGER :: jahr
END TYPE DATUM
TYPE :: MITARBEITER
  CHARACTER *20 :: name, vorname
  TYPE (DATUM) geburtsdatum
  REAL gehalt
  REAL lohnsteuer
  LOGICAL verheiratet
  TYPE (MITARBEITER), POINTER :: alphanext ! (Siehe Kapitel 7)
END TYPE MITARBEITER
```

▲ Beispiel für Typvereinbarungen

```
TYPE (MITARBEITER), DIMENSION (100) :: gehaltsliste
TYPE (DATUM), DIMENSION (365) :: kalender
```

Selbstdefinierte Typen

- Strukturkonstante

```
Syntax: TYPNAME (komponentenliste)
  z. B. DATUM (1, 'Jan', 1996)
```

- Zuweisungen
  - Komponentenweise Zuweisung

```
variablenname%komponentenname = wert
```

  - Objektweise Zuweisung

```
variablenname = strukturkonstante
variablenname = variablenname     !gleichen Typs!
```


Selbstdefinierte Typen

- Beispiele

```
kalender(1)  = DATUM(1,'Jan',1996)
kalender(41)%tag = 10
kalender(41)%monat = 'Feb'
kalender(41)%Jahr = 1941

gehaltsliste(35) = MITARBEITER ('Mueller', 'Horst',&
     DATUM(10,'Feb',1941),4523.12,1212.00,.TRUE.,&
     gehaltsliste(25))

gehaltsliste(42)  = gehaltsliste(35)
gehaltsliste(42)%vorname = 'Anton'
gehaltsliste(42)%alphanext = gehaltsliste(35)
! Kapitel 1.5
```

Selbstdefinierte Typen

weitere Beispiele

- Typdefinition

```
TYPE :: ERSATZTEIL
  SEQUENCE
  CHARACTER *10 pkw_typ
  CHARACTER *10 part_nb
END ERSATZTEIL

TYPE :: PKW
  SEQUENCE
  CHARACTER *10 farbe
  INTEGER fahrgestellnummer
  INTEGER zulassungsjahr
  TYPE (ERSATZTEIL), DIMENSION (100) :: vorderachse
END TYPE PKW
```


Selbstdefinierte Typen

weitere Beispiele

- Typvereinbarungen

```
TYPE (PKW), DIMENSION (200) :: lager
```

- Zuweisungen

```
lager(110)%vorderachse(12)%part_nb = '1000G1019'
lager(1)%farbe = 'RAL312'
```

Selbstdefinierte Typen

```
PROGRAM personal
IMPLICIT NONE
INTEGER, PARAMETER :: max = 10
INTEGER, PARAMETER :: fd = 10
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
!  CHARACTER, DIMENSION (:), POINTER :: beurteilung! Kap. 1.5
END TYPE MITARBEITER

TYPE (MITARBEITER), DIMENSION (max) :: pa! Personal_akte
CHARACTER *4 funktion
CHARACTER zeichen
INTEGER i
LOGICAL :: ende = .FALSE.
```


Selbstdefinierte Typen

```
OPEN(fd,FORM='UNFORMATTED',file = 'database.personal')
   ! Oeffnen der Datei
DO WHILE (.NOT. ende)      !Funktionen

   PRINT *, ' '
   PRINT *, 'PRINT *, 'Eingabe der gewünschten Fkt:'; PRINT *,
' '
   PRINT *, 'init = Initialisiere Datenbasis'
   PRINT *, 'read = Lesen der Datenbasis'
   PRINT *, 'einz = Einzel-Eingabe'
   PRINT *, 'eing = Gruppeneingabe'
   PRINT *, 'korr = Korrektur;'
   PRINT *, 'ausg = Ausgabe;'
   PRINT *, 'sumg = Summe Gehaelter;'
   PRINT *, 'suml = Summe Lohnsteuer;'
   PRINT *, 'pers = Personalliste;'
   PRINT *, 'ende = Beenden des Programms'
   PRINT *, ' '
   PRINT *, 'Funktion?'
   READ *, funktion
   PRINT *, ' '
```

Selbstdefinierte Typen

```
SELECT CASE (funktion)
   CASE ('init')
      PRINT *,'Sind Sie sicher? (y)'
      READ *,zeichen
      if (zeichen /= 'y') EXIT
      DO i = 1,max
            pa(i)%name = 'frei'
            pa(i)%vorname = 'frei'
      END DO
   CASE ('read')
      READ(fd) (pa(i), i=1,max)
   CASE ('einz')
      DO i = 1,max
            IF (pa(i)%name == 'frei') THEN
                  Print *,'Name:'
                  READ *, pa(i)%name
                  PRINT *, 'Vorname'
                  READ * , pa(i)%vorname
                  EXIT
            ELSE
                  CYCLE
            END IF
      END DO
```


Selbstdefinierte Typen

```
   CASE ('eing')
      PRINT *, 'Not yet implemented'
   CASE ('korr')
      PRINT *, 'Not yet implemented'
   CASE ('ausg')
      PRINT *, 'Not yet implemented'
   CASE ('sumg')
      PRINT *, 'Not yet implemented'
   CASE ('suml')
      PRINT *, 'Not yet implemented'
   CASE ('pers')
      DO i= 1,max
            PRINT *, pa(i)%name, ';', pa(i)%vorname
      END DO
   CASE ('ende')
      PRINT *, 'Programmende'
      WRITE(fd) (pa(i), i = 1,max)
      ende = .TRUE.
   CASE DEFAULT
      PRINT *, 'Falsche Eingabe!'
   END SELECT
END DO
END PROGRAM personal
```

3.1.5 Zeiger (Pointer)

- Pointer Definition
  - Der Inhalt einer Pointervariablen ist die Adresse eines Objekts auf das der Pointer zeigt.
  - Konsequenterweise müßten Pointer - wie in anderen Sprachen (z. B. in C) - ein spezieller Datentyp sein. Da man in Fortran90 nicht direkt auf einen Pointer zugreifen kann (keine Adreßarithmetik!), handelt es sich nur um ein Attribut eines Datentyps.
  - In Fortran90 finden Pointer hauptsächlich Verwendung bei der dynamischen Speicherverwaltung (während der Laufzeit des Programms; detailiert in Kapitel 1.8 dargestellt) und beim Aufbau von Datenstrukturen wie verketteten Listen, Baumstrukturen, etc.

Zeiger (Pointer)

- Pointer-Vereinbarung

```
REAL, POINTER :: zeiger, next, current !Typ und Attribut
! oder
REAL zeiger, next, current                   !Typ
Pointer zeiger, next, current                !Attribut
```

- Aus der Pointerdefinition folgt:
  - das Pointer-Attribut darf nur für Variablen oder Funktionen spezifiziert werden
  - folgende weitere Attribute sind unzulässig: **ALLOCATABLE, EXTERNAL, INTENT, INTRINSIC, PARAMETER, TARGET**
  - ein Pointer darf weder bei einer Typvereinbarung noch mit Hilfe einer DATA-Anweisung oder E/A initialisiert werden

Zeiger (Pointer)

Pointer-Zuweisung/Freigabe

- Will man einem Pointer eine Variable (oder Funktion) zuweisen, so muß für diese Datengröße das TARGET-Attribut spezifiziert sein.
  - Spezifikation des TARGET-Attributs

```
INTEGER, TARGET :: feld, person       ! Typ und Attribut
! oder
INTEGER feld, person    ! Typ
TARGET feld, person     ! Attribut
```

  - Zuweisung

```
pointer => zielvariable
! oder
ALLOCATE (pointer [,pointer]...) [, STAT=statusvariable]
```

  - Freigabe von Pointerzielen

```
pointer => neuezielvariable          !Überschreiben
! oder
DEALLOCATE (pointer [,pointer]...) [, STAT=statusvariable]
```


Zeiger (Pointer)

Pointer-Zuordnungsstati

- zugeordnet:
  - dem Pointer wurde eine Variable mit dem TARGET-Attribut zugewiesen.
  - dem Pointer wurde ein Pointer zugewiesen, der selbst den Status “zugeordnet” besitzt.
  - dem Pointer wurde mit der ALLOCATE-Anweisung ein Speicherbereich zugeordnet
- nicht zugeordnet:
  - dem Pointer wird mit der NULLIFY-Anweisung eine Zuordnung entzogen (Keine damit verbundene Speicherfreigabe!).
  - NULLIFY (pointer [,pointer]...)
  - dem Pointer wird mit der DEALLOCATE-Anweisung ein Speicherbereich entzogen, der ihm mit einer ALLOCATE-Anweisung zugeordnet wurde.
  - dem Pointer wird ein Pointer mit dem Status “nicht zugeordnet” zugewiesen.

Zeiger (Pointer)

Pointer-Zuordnungsstati(cont)

- undefiniert:
  - dem Pointer wurde noch kein Ziel zugewiesen (weder '=>' noch **`ALLOCATE`**)
  - das zugeordnete Ziel wurde auf "anderer Weise" entfernt;
    z. B. ein **`DEALLOCATE`** auf einen Zeiger mit dem gleichen Ziel
  - dem Pointer wird ein Pointer zugewiesen mit dem Staus "undefiniert"
  - wenn das Pointerziel bei Ausführung einer **`END`**- oder **`RETURN`**-Anweisung undefiniert wird. (Kapitel 1.6)

Zeiger (Pointer)

ASSOCIATED-Function

- Intrinsic (vordefinierte) Funktion

  `Syntax: ASSOCIATED (pointervar [,targetvar]`

  - liefert den Wert **`.TRUE.`** beim Aufruf

    `ASSOCIATED (pointervar)`

    wenn **`pointervar`** irgendeinem Ziel zugeordnet ist und beim Aufruf

    `ASSOCIATED (pointervar, targetvar)`

    wenn **`pointervar targetvar`** zugeordnet ist.
  - sonst den Wert **`.FALSE.`**

Beispiele

- zu Datenstrukturen, verkettete Listen, Baumstrukturen, etc
  siehe auch Vorl. mit Übungen : "Programm- und Datenstrukturen"
- zu **`ALLOCATE, DEALLOCATE, NULLIFY`**
  siehe auch Feldverarbeitung (Dynamische Speicherverwaltung) Kap. 1.8

© Claus-Uwe Linster – pps3.1-2_spmd_f90: 19/3/98

Zeiger (Pointer)

```
PROGRAM queue

! Warteschlangen-Demo-Programm adaptiert aus:
! Cooper Redwine; "Upgrading to FORTRAN 90", Springer-Verlag,
! 1995
!
! Queue - Policy: first in - first out (Hier zunaechst nur
! Eingabe)

IMPLICIT NONE
TYPE :: QUEUE_ELEMENT
   CHARACTER (20) :: data
   TYPE (QUEUE_ELEMENT), POINTER :: next
END TYPE

! Zeiger auf des erste, letzte, aktuelle Element: first, last,
! this

TYPE (QUEUE_ELEMENT), POINTER :: first, last, this

CHARACTER(20) :: buffer
CHARACTER *4 :: funktion
LOGICAL :: ende = .FALSE.
INTEGER :: io_stat = 1
```


Zeiger (Pointer)

```
   DO WHILE (.NOT. ende)         !Funktionen
   PRINT *, ' '
   PRINT *, 'Eingabe der gewünschten Fkt:'
   PRINT *, ' '
   PRINT *, 'init = Initialisiere Queue'
   PRINT *, 'read = Lese Queue'
   PRINT *, 'erze = Erzeuge Elemente'
   PRINT *, 'dele = Loesche 1. Element'
   PRINT *, 'ende = Beenden des Programms'
   PRINT *, ' '
   PRINT *, 'Funktion:'
   READ *, funktion
   PRINT *, ' '
```

Zeiger (Pointer)

```
SELECT CASE (funktion)
   CASE ('init')
      NULLIFY (first, last)! Initialisiert: leere Queue
   CASE ('read')

      ! Durchlaufe die Warteschlange und drucke Inhalte als
      ! Liste aus.

      this => first           ! aktuelles Element gleich
                              ! 1. Element
      PRINT *, 'Ausgabe'; PRINT *, ' '
      DO WHILE (ASSOCIATED (this))
            PRINT *, this%data
            this => this%next ! aktuelles Element gleich
                              ! naechstes Element
      END DO
```


Zeiger (Pointer)

```
CASE ('erze')
   PRINT *, 'Eingabe:'
   DO WHILE (io_stat >= 0)
      READ (*, "(A)", IOSTAT = io_stat) buffer

      IF (.NOT. ASSOCIATED (first)) THEN

      ! Queue ist leer; fordere den Speicher fuer das
      ! 1. Element an

            ALLOCATE (first)
            last => first     ! Beide Zeiger zeigen aufs
                              ! gleiche(erste)Element
      ELSE
      ! Queue ist nicht leer, neues Element gleich
      ! letztes Element

            ALLOCATE (last%next)
            last => last%next
      END IF
      last%data = buffer
      NULLIFY (last%next)     ! Letztes Element hat
                              ! keinen Nachfolger
   END DO
```

Zeiger (Pointer)

```
      CASE ('dele')
         PRINT *, 'Not yet implemented'
      CASE ('ende')
         PRINT *, 'Programmende'
         ende = .TRUE.
      CASE DEFAULT
         PRINT *, 'Falsche Eingabe!'
      END SELECT
END DO

END PROGRAM queue
```


3.1.6 Unterprogramme

- Definitionen/Terminologie
  - Programmeinheiten (programunits) sind die Bausteine eines ausführbaren Programms
    - Hauptprogramm (genau einmal)
    - ext. Unterprogramme
    - Module
    - Blockdatenprogrammeinheiten
  - Sie enden mit der END-Anweisung, liegen in separaten Dateien, sind getrennt übersetzbar, jedoch bis auf das Hauptprogramm nicht direkt ausführbar.
  - Ein Hauptprogramm ist ausführbar, wenn es keine offenen 'Externreferenzen' gibt - ausgenommen Bibliotheksreferenzen, wie z.B. SIN(x), COS(x), etc.
    Diese Eigenschaft haben unsere bisher geschriebenen Programme.

Unterprogramme

- Die weiteren Programmeinheiten sind nur dann ausführbar (getrennt testbar), wenn ein entsprechendes Hauptprogramm als “Testrahmen” zugebunden wird.
- Nach der Übersetzung sind diese Einheiten zunächst separate Bindemodule; sogenannte ".o-files"

■ Unterprogramme werden unterschieden

- nach ihrem “Definitionsort”
  - extern
  - eingebettet
  - vordefiniert in Bibliothek
- und ihrer Art
  - Subroutine
  - Funktion
  - Formelfunktion

Unterprogramme

■ Subroutine

- Subroutinendefinition

```
[...] SUBROUTINE sname [(...)]        !(SUBROUTINE-Anweisung)
    [Spezifikationsteil]
    ...
        Subroutinenrumpf
    ...
[CONTAINS]
    [interne Unterprogramme]
    ...
END [SUBROUTINE [sname]]
```

- Beispiele:

```
SUBROUTINE sname()                     !(leere Parameterliste)
SUBROUTINE sname (formalparameter[,formalparameter] ...)
```

- Subroutinenaufruf

```
CALL sname
CALL sname()
CALL sname(Aktualparameter[,Aktualparameter]...)
```

Unterprogramme

- Beispiel - eingebettetes Unterprogramm (CONTAINS)

```
PROGRAM summentest
IMPLICIT NONE
REAL a, b, ergebnis

   Print *, 'Eingabe: summand1 summand2 <r>'
   READ *, a,b

   call summe (a,b,ergebnis)

   PRINT *, 'Ergebnis = ',ergebnis

CONTAINS

      SUBROUTINE summe (summanda, summandb, erg)
      REAL summanda, summandb, erg
         erg = summanda + summandb
      END SUBROUTINE summe

END PROGRAM summentest
```


Unterprogramme

- Geltungsbereich/Gültigkeitsbereich
  - sagt aus, in welchen Bereichen eine Variable zugreifbar ist
    - lokal Variable sind nur gültig innerhalb eines Geltungsbereichs z. B. in einem Unterprogramm
    - globale Variable sind in der gesamten Programmeinheit, in der sie definiert wurden, zugreifbar, falls sie nicht durch eine lokale Definition “verdeckt” werden.
      (“Blickt” man aus der Aufrufschachtelung zurück in Richtung Aufrufer, so sind sämtliche globale Daten der aufrufenden Ebenen zugreifbar, es sei denn, sie sind durch lokale Vereinbarungen verdeckt.)

Unterprogramme

- Geltungsbereich/Gültigkeitsbereich(cont)
  - Transfer von Informationen zwischen Geltungsbereichen durch:
    - Parameter
      - Eingangsparameter
      - Ausgangsparameter; Ergebnisparameter
      - beide Richtungen
    - globale Daten
      - im äußersten Geltungsbereich definiert; also programmeinheitsglobal
      - COMMON-Bereiche Überall dort gültig, wo definiert; auch über Programmeinheiten hinweg

Unterprogramme

- Parameter
  - Formalparameter (bei der Definition)
  - Aktualparameter (beim Aufruf)

    Zum Zeitpunkt des Aufrufs werden die Formalparameter und die Aktualparameter einander zugeordnet, d.h. eine “Formal-Variable” im Unterprogramm referenziert den gleichen Speicher wie die “Aktual-Variable” im aufrufenden Programm.

  Man unterscheidet
    - positionsgerechte Schreibweise
    - Schlüsselwortschreibweise z. B.

      formalparameter = aktualparameter

    Die gemischte Schreibweise ist möglich. Beginnt man jedoch mit der Schlüsselwortschreibweise, so müssen auch alle folgenden Parameter in Schlüsselwortschreibweise folgen.

Unterprogramme

INTENT-Attribut/Anweisung

- Mit dem INTENT-Attribut kann die Art der Verwendung von Formalparametern spezifiziert werden:
  - Das INTENT(IN)-Attribut besagt, daß der Formalparameter ein Eingabeparameter ist. D.h. dieser Formalparameter darf innerhalb des Unterprogramms nicht redefiniert werden, er darf also z. B. nicht auf der linken Seite einer Zuweisung auftauchen oder Variable einer READ-Anweisung sein.
    Der Aktualparameter kann also sowohl eine Variable als auch eine Konstante sein.
  - Das INTENT(OUT)-Attribut besagt, daß der Formalparameter ein Ausgabeparameter ist. D. h. der Formalparameter muß erst definiert (ein Wert zugewiesen) werden, ehe im Unterprogramm auf ihn zugegriffen wird. Hier **muß** es sich also um eine Variable handeln.

Unterprogramme

INTENT-Attribut/Anweisung

- Mit dem INTENT-Attribut kann die Art der Verwendung von Formalparametern spezifiziert werden (cont):
  - Das INTENT(INOUT)-Attribut besagt, daß der Parameter sowohl Eingabe- als auch Ausgabeparameter sein soll. Auch hier **muß** es sich also um eine Variable handeln.

Wir werden im Rahmen dieses Kurses die Formalparameter mit INTENT-Attributen versehen denn: Keine Angabe = INTENT(INOUT), es sei denn, der Aktual-Parameter ist eine Konstante.

Unterprogramme

- Beispiel:

```
PROGRAM summentest
IMPLICIT NONE
REAL a, b, ergebnis

   Print *, 'Eingabe: summand1 summand2 <r>'
   READ *, a,b

   call summe (a,b,ergebnis)

   PRINT *, 'Ergebnis = ',ergebnis

CONTAINS

      SUBROUTINE summe (summanda, summandb, erg)
      REAL, INTENT(IN) :: summanda, summandb
      REAL, INTENT(out) :: erg
         erg = summanda + summandb
      END SUBROUTINE summe

END PROGRAM summentest
```


Unterprogramme

- FUNKTION

```
[...] FUNCTION fname [(...)]!(FUNCTION-Anweisung)
   [Spezifikationsteil]
   ...
      Funktionsrumpf
      ...
[CONTAINS]
   [interne Unterprogramme]
   ...
END [FUNCTION [fname]]
```

- der wesentliche Unterschied einer Funktion zu einer Subroutine ist, daß der Funktionsname gleichzeitig auch der Name der (lokalen) Ergebnisvariablen ist. Deshalb hat die Funktion einen Typ!
- Der Aufruf einer Funktion kann überall dort stehen, wo auch eine Variable stehen darf; nur auf der linken Seite einer Zuweisung (natürlich) nicht.
- Der Funktionsaufruf darf auch ein separates Statement sein - analog zur Subroutine - allerdings ohne “CALL”; jedoch mit “()”.

Unterprogramme

```
( ~uelinste/tmp/newton_fkt.f90 )

PROGRAM newton
REAL eingabe
! Author: Uwe Linster
! Anfordern und einlesen der Zahl, deren Quadratwurzel
! berechnet werden soll

   PRINT *, ' '

   ! Ausgabe auf das Sichtgerät ohne Zeilenvorschub
   ! Nichtvorrueckende Ausgabe: Kapitel 1.10

   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Zu berechnen ist die Quadratwurzel aus: ')

   READ *, eingabe
   PRINT *, ' '; PRINT *, ' '

   DO WHILE(eingabe <= 0)
      PRINT *, ' Die Eingabedaten entsprechen nicht den &
                 &Anforderungen des Programms (Zahl > 0!):'
      READ *, eingabe
   END DO
```


Unterprogramme

```
!Berechnung

   ergebnis = sqroot(eingabe)

!Ergebnisausgabe

   PRINT *, ' Die Quadratwurzel aus ', eingabe ;PRINT *, ' '
   PRINT *, ' ist ', ergebnis
! Oder auch gleich:
!  PRINT *, ' ist ', sqroot(eingabe)

PRINT *, ' '; PRINT *, ' Ende des Programms'
```

Unterprogramme

```
CONTAINS

REAL FUNCTION sqroot (a)
!Berechnung der Quadratwurzel nach dem Newton'schen Verfahren
!Eingabeparameter:a; Startwert der Iteration: 1
!Iterationsformel:x(i+1) = (x(i)*x(i) + a) / (2*x(i))

   REAL , INTENT(IN) :: a
   REAL x
      !a Größe, deren Quadratwurzel zu berechnen ist
      !x Berechnete Approximation der Quadratwurzel

   !Anfangsapproximation
   x = 1
   !Iteration
   DO WHILE (.NOT. (ABS(x*x - a) <= 1E-4))
      x = (x*x + a) / (2.*x)
   END DO
   sqroot = x
END FUNCTION sqroot

END PROGRAM newton
```


Unterprogramme

- COMMON-BEREICHE
  - Durch COMMON-Bereiche werden globale Speicherbereiche spezifiziert. Variable sind in den Geltungsbereichen direkt zugreifbar, in denen die COMMON-Bereiche spezifiziert wurden; auch über verschiedene Programmeinheiten hinweg.
    Man unterscheidet:
    - einen unbenannten COMMON-Block

```
Syntax:COMMON variablenliste
```

- und beliebig viele benannte COMMON-Blöcke

```
Syntax:COMMON /blockname/ variablenliste [, /blockname/]
              variablenliste]...
```

- auch die Mischschreibweise ist möglich:

```
Syntax:COMMON  variablenliste [[,]/[blockname]/
              variablenliste]...
```

Unterprogramme

- COMMON-BEREICHE(cont)
  - Eigenschaften
    - Durch Angabe der Variablen wird eine Speicherfolge festgelegt; um Fehler zu vermeiden sollte man mit “copy” und “paste” arbeiten.
    - Variablennamen in verschiedenen Geltungseinheiten dürfen verschieden sein; das sollte man jedoch nur tun, wenn es dafür auch einen plausiblen Grund gibt. Man greift auf dieselbe globale Speicheradresse zu, das sollte man durch den gleiche Variablennamen auch dokumentieren!
    - Initialisiert werden COMMON-Bereiche mit einer BLOCK DATA-Programmeinheit; jedoch nur die benannten Blöcke.
    - Selbstdefinierte Datentypen müssen das Attribut SEQUENCE besitzen. (Speicherfolge!)

Unterprogramme

- COMMON-Bereich - Beispiel (~uelinste/tmp/common_demo_bubble.f90

```
PROGRAM bubble
IMPLICIT NONE

! Zweck des Programms, Loesungsverfahren:
!
! Die Eintraege in der Datei titel sollen in lexikographisch
! sortierter Reihenfolge gedruckt werden. Das Programm geht
! davon aus, dass die Datei maximal 200 Eintraege besitzt und
! jeder Eintrag aus hoechstens 20 Zeichen besteht.
! Als Sortierverfahren wird 'bubble sort' verwendet.
!-------------------------------------------------------------
! Globale Variable; in COMMON-Bereiche spezifiziert
!    name Feld zur Aufnahme der Eintraege aus der Dateri titel
!    max Anzahl der Eintraege
!  fd Kanalnummer (Filedescriptor; Unit)
!-------------------------------------------------------------
! Initialisierung
!
! Ausgabe der Gesamtueberschrift
!
PRINT *, 'Eintraege der Datei titel lexikograpgisch sortiert:'
```

Unterprogramme

```
! Oeffnen der Datei
!
   CALL oeffnen_datei()

! Einlesen der Daten
!
   CALL einlesen()
!
! Sortieren
!
   CALL bubblesort()
!
! Ausgabe des Ergebnisses
!
   CALL ausgeben_screen()
   CALL ausgeben_datei()
```


Unterprogramme

```
! Die Unterprogramme:

CONTAINS

SUBROUTINE oeffnen_datei()

! Oeffnen der Datei "titel" mit Kanalnummer 10
! Global Data:

INTEGER fd
COMMON /datei/ fd
   fd = 10      ! Entfaelt bei BLOCK-DATA-INIT!!
! Local DATA: none

   open(fd, FORM = 'FORMATTED', file = 'titel')

END SUBROUTINE oeffnen_datei
```

Unterprogramme

```
SUBROUTINE einlesen()

! Einlesen der Eintraege aus der Datei "titel" in das Feld !
"name".
! Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

!Local Data:
INTEGER i

   DO i = 1, 200
      READ(fd, 1000, END = 1010) name(i)
1000  FORMAT (a)
   END DO

1010max = i - 1
   PRINT *, 'Gelesene Datensaetze: ',max

END SUBROUTINE einlesen
```


Unterprogramme

```
SUBROUTINE bubblesort()
! Sortiert nach dem "bubble-sort"-Verfahren
! Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

!Local Data

CHARACTER *20 :: hilf
LOGICAL flag
INTEGER i, j, dl

!  flag   zeigt an, ob bei einem Durchlauf Aenderungen in der
!              Anordnung vorgenommen wurden
!              (.FALSE.: Keine Umordnungen
!               .TRUE. : Es wurden Umordnungen vorgenommen)
!  hilf   Hilfsfeld waehrend des Vertauschens zweier Eintraege
!  dl Anzahl der Sortierschleifen
!
```

Unterprogramme

```
! Der Sortiervorgang

   dl = 0
   DO  i = max, 2, -1
      flag = .FALSE.

      DO j = 1, i - 1
          IF (name(j) > name(j + 1)) THEN
               hilf = name(j)
               name(j) = name(j + 1)
               name(j + 1) = hilf
               flag = .TRUE.
               dl = dl + 1
          END IF
        END DO

        if (.NOT. flag) THEN
         PRINT *, 'Anzahl der Durchlaeufe: ', dl
         EXIT
      END IF

   END DO

END SUBROUTINE bubblesort
```


Unterprogramme

```
SUBROUTINE ausgeben_screen()

!Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max
COMMON name, max

! Local Data:

INTEGER i

   DO i=1,max
      Print *, name(i)
   END DO

END SUBROUTINE ausgeben_screen
```

Unterprogramme

```
SUBROUTINE ausgeben_datei()

!Global Data

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

! Local Data

INTEGER i

   REWIND (fd)
   WRITE ( fd, 1030 )(name(i), i=1, max)
1030FORMAT (a)

END SUBROUTINE ausgeben_datei

END PROGRAM bubble
```


Unterprogramme

- BLOCK DATA
  - Eine BLOCK-DATA-Programmeinheit
    - ist eine nichtausführbare Programmeinheit, sie dient der Initialisierung von Variablen in **benannten** COMMON-Bereichen.
    - COMMON-Blöcke sind globale Speicherbereiche - gültig auch über Programm-Unit-Grenzen hinweg. Eine Initialisierung darf also Programmsystemweit nur einmal erfolgen. Dies könnte innerhalb einer “ausgezeichneten” Programmeinheit erfolgen oder wie hier in einer speziellen Programmeinheit

```
BLOCKDATA [name]
     [Spezifikationsteil]
     .
     .
     .
  END BLOCKDATA [name]
```

Unterprogramme

- BLOCK DATA(cont)

```
Beispiel
BLOCKDATA car

CHARACTER vorderachse *14, hinterachse *10, motor *11
INTEGER anzahl, seriennummer, partnb

COMMON /ersatzteile/ vorderachse, hinterachse, motor
COMMON /attribute/ anzahl, seriennummer, partnb

DATA vorderachse, hinterachse /'Schwingachsen', `Inkaachsen'/
DATA motor /'Dieselmotor'/
DATA anzahl, partnb /250, 4711/ ! seriennummer bleibt
uninitialisiert

END BLOCKDATA car
```

- Da diese Programm-Unit in einer eigenen Datei liegt muß beim Aufruf des Compilers diese Datei als weiterer Parameter angegeben werden:; z. B.

```
f90 hauptprogramm.f90 blockdata.f90 -o programm.out
```


Unterprogramme

- BLOCK DATA
  - Beispiel für Programm “bubble”:

Datei: bubble_blockdata.f90

```
BLOCKDATA bubble

   INTEGER fd

   COMMON /datei/ fd

   DATA fd /10/

END BLOCKDATA bubble
```

- Aufruf des Compilers:

```
f90 common_demo_bubble.f90 bubble_blockdata.f90 -o bubble.out
```

Unterprogramme

■ FORMELFUNKTION

- eine Formelfunktion ist eine Funktion, die nur aus einer einzigen Anweisung besteht und
- sie ist nur innerhalb einer Geltungseinheit aufrufbar.
- Formelfunktionsdefinition:

```
Syntax:ffname() = ausdruck oder
ffname(formalparameter[,formalparameter]...) =
                                  ausdruck
```

- Sowohl 'ausdruck' als auch 'formalparameter' sind skalare Datenobjekte.
- Formelfunktionsaufruf:

```
Syntax:ffname() oder
ffname(formalparameter[,formalparameter]...)
```

- Der Typ der Formelfunktion wird separat (wie bei der Typvereinbarung einer Variablen) vereinbart; dieses gilt auch für die Formalparameter.

Unterprogramme

- Formelfunktion - Beispiel (~uelinste/tmp/capslock_fkt.f90)

```
PROGRAM capslock
IMPLICIT NONE
!
! Das Programm liest Kleinbuchstaben ein, wandelt sie
! in Grossbuchstaben um und druckt diese aus.
!  Variable
!
      CHARACTER, DIMENSION(20) :: zeichenin, zeichenout
      INTEGER i

   CHARACTER gross, klein

!  Formelfunktion:

   gross(klein) = CHAR ( ICHAR(klein) + ICHAR('A') - ICHAR('a'))
```

Unterprogramme

```
!  Einlesen der Zeichen

      PRINT *, ' '
   PRINT *, 'Kleinbuchstaben; max. 20; entweder durch "blank"
getrennt -'
   PRINT *, 'Abschluss <r> oder einzeln durch <r> getrennt!'
   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Eingabe: ')
      READ *, (zeichenin(i), i=1,20)

   DO i = 1,20
      zeichenout(i) = gross(zeichenin(i))
   END DO

   PRINT *, 'Ausgabe:',zeichenout

END PROGRAM capslock
```


Unterprogramme

- Formelfunktion - Beispiel (~uelinste/tmp/kredit_fkt.f90)

```
PROGRAM kredit
IMPLICIT NONE
!
!  Das Programm berechnet die Hoehe der monatlichen Raten fuer
! einen vorgebbaren Kaufpreis bei einzugebender Laufzeit. Raten
!  und Zinsen werden von zwei Jahren bis zu einer angebbaren
!  maximalen Laufzeit in Jahresschritten berechnet. Untere und
!  oebere Grenze des Zinssatzes werden ebenfalls eingelesen.
! ------------------------------------------------------------
!  Variable
!
      REAL preis, mrate, p
      INTEGER n, zmin, zmax, zsatz, zs, year, yearmax
!
! Berechnung der Monatsrate fuer die verschiedenen Zinssaetze
!          formel: r*p/(1-(1+r)**(-n)) mit r = zinssatz/12/100
!  hier als Formelfunktion realisiert:

   mrate(zs,p,n) = ((zs/100./12.) * p) / &
               (1.-(1. + (zs/100./12.))**(-n))
```

Unterprogramme

```
!        preis      Kaufpreis (Eingabe)
!       yearmax    maximale Laufzeit der Ratenzahlung (Eingabe)
!        zmin       minimaler Zinssatz (Eingabe)
!        zmax       maximaler Zinssatz (Eingabe)
!        year       Laufvariable fuer die Laufzeit
!        n          Zahl der Raten
!        zsatz      Laufvariable fuer den Zinssatz
! ------------------------------------------------------------
! Initialisierung
!
!  Einlesen von Kaufpreis, maximaler Laufzeit,
!  minimalem und maximalem Zinssatz
!
      PRINT *, ' '
   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Kaufpreis (REAL): ')
      READ *, preis
   WRITE (*, 2000, ADVANCE='NO')
2000FORMAT ('Laufzeit in Jahren (groesser als 2): ')
      READ *, yearmax
     WRITE (*, 3000, ADVANCE='NO')
3000FORMAT ('minimaler und maximaler Zins (in Prozent
(INTEGER)): ')
      READ *, zmin, zmax; PRINT *, ' '
```


Unterprogramme

```
! Aeussere Schleife

    DO year = 2, yearmax

!  Tabellenueberschrift
        PRINT *, ' '
        PRINT *, ' '
      PRINT *, ' Zinsen fuer eine Laufzeit von ', year, ' Jahren'
        PRINT *, ' '
       PRINT *, ' Zins-  Zahl der     Rate          Gesamtzins'
        PRINT *, ' satz    Raten       (DM)              (DM)'

! Innere Schleife
      DO zsatz = zmin, zmax
         n = 12.*year
          PRINT *, ' ', zsatz, '      ', n, ' ', &
               mrate (zsatz, preis, n) &
               ,' ', n * (mrate (zsatz, preis, n)) -preis
      END DO
    END DO
!
!        Abschlussausgabe
      PRINT *, ' ' ;PRINT *, ' Berechnung abgeschlossen.'
END PROGRAM kredit
```

Unterprogramme

- Rücksprung

- Es existieren drei alternative Methoden
  - der “normale” Rücksprung bei Erreichung des END-Statements
  - der “Ausnahme” - Rücksprung, wenn aufgrund einer Bedingung die Aufgabe des Unterprogramms erfüllt ist - etwa analog zum EXIT-Aussprung aus einer DO-Schleife

```
Syntax:RETURN
```

Bei beiden Methoden wird mit dem 1. ausführbaren Statement fortgefahren, das dem Unterprogrammaufruf folgt.

  - der “Ausnahme” - Rücksprung mit wählbarer Fortsetzungsadresse (label).

```
Syntax:RETURN ganzzahliger_ausdruck
```

  - Diese Möglichkeit wird nur der Vollständigkeit erwähnt (Im Falle, daß man mit vorhandenen Programmsystemen umgehen muß). Die Verwendung solcher Konstrukte führt zu undurchsichtigen, nicht mehr wartbaren Programmsystemen.

Unterprogramme

- SAVE-Anweisung

- Mit dem SAVE-Attribut wird der Wert einer **lokalen** Variablen in einem Geltungsbereich über die “Lebensdauer” eines Aufrufs hinaus “gesichert”. D. h., daß der Wert der Variablen auch beim nächsten Aufruf wieder verfügbar ist. Man spricht auch von “*Static”* - Variablen.

- Das SAVE-Attribut ist nur erlaubt bei
  - lokalen Variablen
  - benannten COMMON-Blöcken
    (hier ist Vorsicht geboten, ist doch ein COMMON-Block ein globaler Speicherbereich!?!)

- Das SAVE-Attribut ist insbesondere nicht erlaubt bei
  - einzelnen Variablen, die in einem COMMON-Bereich enthalten sind
  - Formalparametern
  - Funktionsvariablen
  - Namensgruppen (Siehe Kapitel 1.10)

Unterprogramme

■ SAVE-Anweisung

▲ Attributspezifikation

- gleichzeitig mit der Typvereinbarung

```
REAL, SAVE :: skonto, summe
INTEGER, SAVE :: max
```

- oder getrennt mit einer SAVE-Anweisung

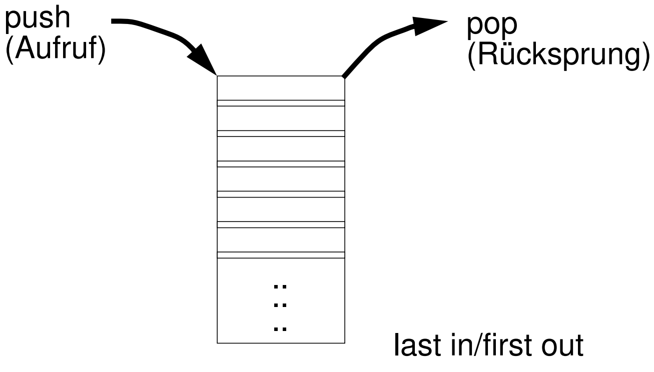
```
REAL skonto, summe
SAVE skonto, summe

INTEGER max
SAVE max
```

Eine SAVE-Anweisung ohne Angabe von Variablennamen bedeutet, das sämtliche lokale Variable in diesem Geltungsbereich diese Spezifikation erhalten.

Rekursive Unterprogramme

Unterprogramme (SUBROUTINE oder FUNCTION) sind dann rekursiv, wenn sie sich selbst direkt oder indirekt aufrufen.

Wenn Unterprogramme das Attribut RECURSION besitzen, muß der Compiler dafür Sorge tragen, daß jeweils bei einem rekursiven Aufruf die lokalen Variablen gerettet und nach dem Rücksprung aus einer tieferen Aufrufebene wieder bereitgestellt werden. Dies ist eine typische Anwendung für einen Kellerspeicher (Stack)!



Rekursive Unterprogramme

- Wann verwendet man rekursive Unterprogramme?
  - Es gibt eine Anzahl von Algorithmen, die "in sich" rekursiv definiert sind. Bei der iterativen Implementierung solcher Algorithmen ist der Programmieraufwand in der Regel beträchtlich und mit einer großen Anzahl von Abfragen behaftet, wie einige Beispiele des Kapitels noch zeigen werden.
  - Als einfaches Beispiel - zum Verständnis - wurde die Berechnung von Fakultäten gewählt, tatsächlich ist es in diesem Fall allerdings effizienter, eine Schleife zu implementieren!

```
PROGRAM recursion
IMPLICIT NONE
INTEGER n, erg
   WRITE (*, FMT = "('n! ; Welches n? : ')", ADVANCE = 'NO')
   READ *, n
   CALL fak (n, erg)
   PRINT *, 'Fakultaet von: ',n ,' = ', erg
   erg = fakfunc (n)
   PRINT *, 'Fakultaet von: ',n ,' = ', erg
CONTAINS
```


Rekursive Unterprogramme

- Rekursive Subroutine

```
RECURSIVE SUBROUTINE fak (fn, ferg)

INTEGER, INTENT (IN) :: fn
INTEGER, INTENT (OUT) :: ferg

   IF (fn == 0) THEN
        ferg = 1
   ELSE
        CALL fak (fn - 1, ferg)
        IF (ferg < 0) THEN
             PRINT *, 'underflow'
             STOP
        END IF
        ferg = fn * ferg
   END IF

END SUBROUTINE fak
```

Rekursive Unterprogramme

▲ Rekursive Funktion

```
RECURSIVE INTEGER FUNCTION fakfunc (fn) RESULT (ferg)

INTEGER, INTENT (IN) :: fn

   IF (fn == 0) THEN
         ferg = 1
   ELSE
         ferg = fn * fakfunc (fn - 1)
         IF (ferg < 0) THEN
               PRINT *, 'underflow'
               STOP
         END IF
   END IF

END FUNCTION fakfunc

END PROGRAM recursion
```


Unterprogramme

■ (Noch) nicht behandelt

- Parameterübergabe bei nicht skalaren Datengrößen (Siehe Kapitel 1.7- Feldverarbeitung)
- Module (Siehe Kapitel 1.11 - Module)
- Externe Unterprogramme (Siehe Kapitel n.b.)
- Nebeneinganspunkte - ENTRY-Anweisungen (Siehe Kapitel n.b.)
- Optionale Formalparameter (Siehe Kapitel n.b.)

Programmierung Paralleler Systeme

3. Programmiermodell “Single Programm Multiple Data”

3.1 Fortran95

3.1 .4 Selbstdef. Datentypen (Strukturen)

3.1.5 Zeiger(Pointer)

3.1.6 Unterprogramme

Claus-Uwe Linster

IMMD IV, Lehrstuhl für Betriebssysteme
Universität Erlangen-Nürnberg

linster@informatik.uni-erlangen.de

3.1.4 Selbstdefinierte Typen

▲ Typdefinition

```
TYPE [,attribut]... [::] typname
   komponentendefinition
   [komponentendefinition]
   .
END TYPE [typname]
```

Die Komponentendefinition entspricht weitgehend den Typvereinbarungsanweisungen; jedoch

- keine Initialisierung (es ist eine Definition; keine Deklaration!)
- als Attribute sind nur
  - DIMENSION (mit expliziter Gestalt) und
  - POINTER (Siehe Kapitel 1.5)

  zugelassen.

Selbstdefinierte Typen

Als Attribute der Typdefinition (dann gültig für alle Deklarationen dieses Typs) sind zulässig:

- PRIVATE (Siehe Kapitel 1.11)
- PUBLIC (Siehe Kapitel 1.11)
- SEQUENCE
  - erzwingt eine Speicherordnung gemäß der Komponentenreihenfolge; dies kann hinsichtlich der Speichernutzung Verschnitt bedeuten.
  - Verschiedene Typdefinitionen mit gleichen Komponenten (z. B. in verschiedenen Gültigkeitsbereichen)sind nur dann wirklich gleich, wenn sie das SEQUENCE-Attribut besitzen.
  - Wenn ein selbstdefinierter Typ das SEQUENCE-Attribut besitzt, dann müssen auch alle ggf. vorhandenen Typ-Komponenten, die ihrerseits selbstdefinierte Typen sind, das SEQUENCE-Attribut besitzen.
- Type - Vereinbarungsanweisung

```
Syntax:TYPE (TYPENAME) [,attribut]... [::] dg [,dg]...
```


Selbstdefinierte Typen

▲ Beispiele für Typdefinitionen

```
TYPE:: DATUM
  SEQUENCE
  INTEGER :: tag
  CHARCATER (LEN=3) :: monat
  INTEGER :: jahr
END TYPE DATUM
TYPE :: MITARBEITER
  CHARACTER *20 :: name, vorname
  TYPE (DATUM) geburtsdatum
  REAL gehalt
  REAL lohnsteuer
  LOGICAL verheiratet
  TYPE (MITARBEITER), POINTER :: alphanext ! (Siehe Kapitel 7)
END TYPE MITARBEITER
```

▲ Beispiel für Typvereinbarungen

```
TYPE (MITARBEITER), DIMENSION (100) :: gehaltsliste
TYPE (DATUM), DIMENSION (365) :: kalender
```

Selbstdefinierte Typen

- Strukturkonstante

```
Syntax: TYPNAME (komponentenliste)
  z. B. DATUM (1, 'Jan', 1996)
```

- Zuweisungen
  - Komponentenweise Zuweisung

```
variablenname%komponentenname = wert
```

  - Objektweise Zuweisung

```
variablenname = strukturkonstante
variablenname = variablenname      !gleichen Typs!
```


Selbstdefinierte Typen

- Beispiele

```
kalender(1)  = DATUM(1,'Jan',1996)
kalender(41)%tag = 10
kalender(41)%monat = 'Feb'
kalender(41)%Jahr = 1941

gehaltsliste(35) = MITARBEITER ('Mueller', 'Horst',&
     DATUM(10,'Feb',1941),4523.12,1212.00,.TRUE.,&
     gehaltsliste(25))

gehaltsliste(42)  = gehaltsliste(35)
gehaltsliste(42)%vorname = 'Anton'
gehaltsliste(42)%alphanext = gehaltsliste(35)
! Kapitel 1.5
```

Selbstdefinierte Typen

weitere Beispiele

- Typdefinition

```
TYPE :: ERSATZTEIL
  SEQUENCE
  CHARACTER *10 pkw_typ
  CHARACTER *10 part_nb
END ERSATZTEIL

TYPE :: PKW
  SEQUENCE
  CHARACTER *10 farbe
  INTEGER fahrgestellnummer
  INTEGER zulassungsjahr
  TYPE (ERSATZTEIL), DIMENSION (100) :: vorderachse
END TYPE PKW
```


Selbstdefinierte Typen

weitere Beispiele

- Typvereinbarungen

```
TYPE (PKW), DIMENSION (200) :: lager
```

- Zuweisungen

```
lager(110)%vorderachse(12)%part_nb = '1000G1019'
lager(1)%farbe = 'RAL312'
```

Selbstdefinierte Typen

```
PROGRAM personal
IMPLICIT NONE
INTEGER, PARAMETER :: max = 10
INTEGER, PARAMETER :: fd = 10
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
!  CHARACTER, DIMENSION (:), POINTER :: beurteilung! Kap. 1.5
END TYPE MITARBEITER

TYPE (MITARBEITER), DIMENSION (max) :: pa! Personal_akte
CHARACTER *4 funktion
CHARACTER zeichen
INTEGER i
LOGICAL :: ende = .FALSE.
```


Selbstdefinierte Typen

```
OPEN(fd,FORM='UNFORMATTED',file = 'database.personal')
   ! Oeffnen der Datei
DO WHILE (.NOT. ende)      !Funktionen

   PRINT *, ' '
   PRINT *, 'PRINT *, 'Eingabe der gewünschten Fkt:'; PRINT *,
' '
   PRINT *, 'init = Initialisiere Datenbasis'
   PRINT *, 'read = Lesen der Datenbasis'
   PRINT *, 'einz = Einzel-Eingabe'
   PRINT *, 'eing = Gruppeneingabe'
   PRINT *, 'korr = Korrektur;'
   PRINT *, 'ausg = Ausgabe;'
   PRINT *, 'sumg = Summe Gehaelter;'
   PRINT *, 'suml = Summe Lohnsteuer;'
   PRINT *, 'pers = Personalliste;'
   PRINT *, 'ende = Beenden des Programms'
   PRINT *, ' '
   PRINT *, 'Funktion?'
   READ *, funktion
   PRINT *, ' '
```

© Claus-Uwe Linster – pps3.1-2_spmd_f90: 19/3/98

Selbstdefinierte Typen

```
   SELECT CASE (funktion)
      CASE ('init')
         PRINT *,'Sind Sie sicher? (y)'
         READ *,zeichen
         if (zeichen /= 'y') EXIT
         DO i = 1,max
               pa(i)%name = 'frei'
               pa(i)%vorname = 'frei'
         END DO
      CASE ('read')
         READ(fd) (pa(i), i=1,max)
      CASE ('einz')
         DO i = 1,max
               IF (pa(i)%name == 'frei') THEN
                     Print *,'Name:'
                     READ *, pa(i)%name
                     PRINT *, 'Vorname'
                     READ * , pa(i)%vorname
                     EXIT
               ELSE
                     CYCLE
               END IF
         END DO
```


Selbstdefinierte Typen

```
      CASE ('eing')
         PRINT *, 'Not yet implemented'
      CASE ('korr')
         PRINT *, 'Not yet implemented'
      CASE ('ausg')
         PRINT *, 'Not yet implemented'
      CASE ('sumg')
         PRINT *, 'Not yet implemented'
      CASE ('suml')
         PRINT *, 'Not yet implemented'
      CASE ('pers')
         DO i= 1,max
               PRINT *, pa(i)%name, ';', pa(i)%vorname
         END DO
      CASE ('ende')
         PRINT *, 'Programmende'
         WRITE(fd) (pa(i), i = 1,max)
         ende = .TRUE.
      CASE DEFAULT
         PRINT *, 'Falsche Eingabe!'
      END SELECT
END DO
END PROGRAM personal
```

3.1.5 Zeiger (Pointer)

- Pointer Definition
  - Der Inhalt einer Pointervariablen ist die Adresse eines Objekts auf das der Pointer zeigt.
  - Konsequenterweise müßten Pointer - wie in anderen Sprachen (z. B. in C) - ein spezieller Datentyp sein. Da man in Fortran90 nicht direkt auf einen Pointer zugreifen kann (keine Adreßarithmetik!), handelt es sich nur um ein Attribut eines Datentyps.
  - In Fortran90 finden Pointer hauptsächlich Verwendung bei der dynamischen Speicherverwaltung (während der Laufzeit des Programms; detailiert in Kapitel 1.8 dargestellt) und beim Aufbau von Datenstrukturen wie verketteten Listen, Baumstrukturen, etc.

Zeiger (Pointer)

- Pointer-Vereinbarung

```
REAL, POINTER :: zeiger, next, current !Typ und Attribut
! oder
REAL zeiger, next, current                  !Typ
Pointer zeiger, next, current               !Attribut
```

- Aus der Pointerdefinition folgt:
  - das Pointer-Attribut darf nur für Variablen oder Funktionen spezifiziert werden
  - folgende weitere Attribute sind unzulässig: **ALLOCATABLE, EXTERNAL, INTENT, INTRINSIC, PARAMETER, TARGET**
  - ein Pointer darf weder bei einer Typvereinbarung noch mit Hilfe einer DATA-Anweisung oder E/A initialisiert werden

Zeiger (Pointer)

Pointer-Zuweisung/Freigabe

- Will man einem Pointer eine Variable (oder Funktion) zuweisen, so muß für diese Datengröße das TARGET-Attribut spezifiziert sein.
  - Spezifikation des TARGET-Attributs

```
INTEGER, TARGET :: feld, person      ! Typ und Attribut
! oder
INTEGER feld, person    ! Typ
TARGET feld, person     ! Attribut
```

  - Zuweisung

```
pointer => zielvariable
! oder
ALLOCATE (pointer [,pointer]...) [, STAT=statusvariable]
```

  - Freigabe von Pointerzielen

```
pointer => neuezielvariable         !Überschreiben
! oder
DEALLOCATE (pointer [,pointer]...) [, STAT=statusvariable]
```


Zeiger (Pointer)

Pointer-Zuordnungsstati

- zugeordnet:
  - dem Pointer wurde eine Variable mit dem TARGET-Attribut zugewiesen.
  - dem Pointer wurde ein Pointer zugewiesen, der selbst den Status “zugeordnet” besitzt.
  - dem Pointer wurde mit der ALLOCATE-Anweisung ein Speicherbereich zugeordnet
- nicht zugeordnet:
  - dem Pointer wird mit der NULLIFY-Anweisung eine Zuordnung entzogen (Keine damit verbundene Speicherfreigabe!).
  - NULLIFY (pointer [,pointer]...)
  - dem Pointer wird mit der DEALLOCATE-Anweisung ein Speicherbereich entzogen, der ihm mit einer ALLOCATE-Anweisung zugeordnet wurde.
  - dem Pointer wird ein Pointer mit dem Status “nicht zugeordnet” zugewiesen.

Zeiger (Pointer)

Pointer-Zuordnungsstati(cont)

- undefiniert:
  - dem Pointer wurde noch kein Ziel zugewiesen (weder '=>' noch **`ALLOCATE`**)
  - das zugeordnete Ziel wurde auf "anderer Weise" entfernt;
    z. B. ein **`DEALLOCATE`** auf einen Zeiger mit dem gleichen Ziel
  - dem Pointer wird ein Pointer zugewiesen mit dem Staus "undefiniert"
  - wenn das Pointerziel bei Ausführung einer **`END`**- oder **`RETURN`**-Anweisung undefiniert wird. (Kapitel 1.6)

Zeiger (Pointer)

ASSOCIATED-Function

- Intrinsic (vordefinierte) Funktion

  `Syntax: ASSOCIATED (pointervar [,targetvar]`

  - liefert den Wert **`.TRUE.`** beim Aufruf

    `ASSOCIATED (pointervar)`

  wenn **`pointervar`** irgendeinem Ziel zugeordnet ist und beim Aufruf

    `ASSOCIATED (pointervar, targetvar)`

  wenn **`pointervar targetvar`** zugeordnet ist.

  - sonst den Wert **`.FALSE.`**

Beispiele

- zu Datenstrukturen, verkettete Listen, Baumstrukturen, etc
  siehe auch Vorl. mit Übungen : "Programm- und Datenstrukturen"
- zu **`ALLOCATE, DEALLOCATE, NULLIFY`**
  siehe auch Feldverarbeitung (Dynamische Speicherverwaltung) Kap. 1.8

Zeiger (Pointer)

```
PROGRAM queue

! Warteschlangen-Demo-Programm adaptiert aus:
! Cooper Redwine; "Upgrading to FORTRAN 90", Springer-Verlag,
! 1995
!
! Queue - Policy: first in - first out (Hier zunaechst nur
! Eingabe)

IMPLICIT NONE
TYPE :: QUEUE_ELEMENT
   CHARACTER (20) :: data
   TYPE (QUEUE_ELEMENT), POINTER :: next
END TYPE

! Zeiger auf des erste, letzte, aktuelle Element: first, last,
! this

TYPE (QUEUE_ELEMENT), POINTER :: first, last, this

CHARACTER(20) :: buffer
CHARACTER *4 :: funktion
LOGICAL :: ende = .FALSE.
INTEGER :: io_stat = 1
```


Zeiger (Pointer)

```
   DO WHILE (.NOT. ende)         !Funktionen
   PRINT *, ' '
   PRINT *, 'Eingabe der gewünschten Fkt:'
   PRINT *, ' '
   PRINT *, 'init = Initialisiere Queue'
   PRINT *, 'read = Lese Queue'
   PRINT *, 'erze = Erzeuge Elemente'
   PRINT *, 'dele = Loesche 1. Element'
   PRINT *, 'ende = Beenden des Programms'
   PRINT *, ' '
   PRINT *, 'Funktion:'
   READ *, funktion
   PRINT *, ' '
```

Zeiger (Pointer)

```
SELECT CASE (funktion)
   CASE ('init')
      NULLIFY (first, last)! Initialisiert: leere Queue
   CASE ('read')

      ! Durchlaufe die Warteschlange und drucke Inhalte als
      ! Liste aus.

      this => first            ! aktuelles Element gleich
                               ! 1. Element
      PRINT *, 'Ausgabe'; PRINT *, ' '
      DO WHILE (ASSOCIATED (this))
            PRINT *, this%data
            this => this%next ! aktuelles Element gleich
                               ! naechstes Element
      END DO
```


Zeiger (Pointer)

```
CASE ('erze')
   PRINT *, 'Eingabe:'
   DO WHILE (io_stat >= 0)
      READ (*, "(A)", IOSTAT = io_stat) buffer

      IF (.NOT. ASSOCIATED (first)) THEN

      ! Queue ist leer; fordere den Speicher fuer das
      ! 1. Element an

            ALLOCATE (first)
            last => first      ! Beide Zeiger zeigen aufs
                               ! gleiche(erste)Element
      ELSE
      ! Queue ist nicht leer, neues Element gleich
      ! letztes Element

            ALLOCATE (last%next)
            last => last%next
      END IF
      last%data = buffer
      NULLIFY (last%next)      ! Letztes Element hat
                               ! keinen Nachfolger
   END DO
```

Zeiger (Pointer)

```
      CASE ('dele')
         PRINT *, 'Not yet implemented'
      CASE ('ende')
         PRINT *, 'Programmende'
         ende = .TRUE.
      CASE DEFAULT
         PRINT *, 'Falsche Eingabe!'
      END SELECT
END DO

END PROGRAM queue
```


3.1.6 Unterprogramme

- Definitionen/Terminologie
  - Programmeinheiten (programunits) sind die Bausteine eines ausführbaren Programms
    - Hauptprogramm (genau einmal)
    - ext. Unterprogramme
    - Module
    - Blockdatenprogrammeinheiten
  - Sie enden mit der END-Anweisung, liegen in separaten Dateien, sind getrennt übersetzbar, jedoch bis auf das Hauptprogramm nicht direkt ausführbar.
  - Ein Hauptprogramm ist ausführbar, wenn es keine offenen 'Externreferenzen' gibt - ausgenommen Bibliotheksreferenzen, wie z.b. SIN(x), COS(x), etc.
    Diese Eigenschaft haben unsere bisher geschriebenen Programme.

Unterprogramme

- Die weiteren Programmeinheiten sind nur dann ausführbar (getrennt testbar), wenn ein entsprechendes Hauptprogramm als “Testrahmen” zugebunden wird.
- Nach der Übersetzung sind diese Einheiten zunächst separate Bindemodule; sogenannte ".o-files"

■ Unterprogramme werden unterschieden

- nach ihrem “Definitionsort”
  - extern
  - eingebettet
  - vordefiniert in Bibliothek
- und ihrer Art
  - Subroutine
  - Funktion
  - Formelfunktion

Unterprogramme

■ Subroutine

- Subroutinendefinition

```
[...] SUBROUTINE sname [(...)]        !(SUBROUTINE-Anweisung)
    [Spezifikationsteil]
    ...
        Subroutinenrumpf
    ...
[CONTAINS]
    [interne Unterprogramme]
    ...
END [SUBROUTINE [sname]]
```

- Beispiele:

```
SUBROUTINE sname()                       !(leere Parameterliste)
SUBROUTINE sname (formalparameter[,formalparameter] ...)
```

- Subroutinenaufruf

```
CALL sname
CALL sname()
CALL sname(Aktualparameter[,Aktualparameter]...)
```

Unterprogramme

- Beispiel - eingebettetes Unterprogramm (CONTAINS)

```
PROGRAM summentest
IMPLICIT NONE
REAL a, b, ergebnis

   Print *, 'Eingabe: summand1 summand2 <r>'
   READ *, a,b

   call summe (a,b,ergebnis)

   PRINT *, 'Ergebnis = ',ergebnis

CONTAINS

      SUBROUTINE summe (summanda, summandb, erg)
      REAL summanda, summandb, erg
         erg = summanda + summandb
      END SUBROUTINE summe

END PROGRAM summentest
```


Unterprogramme

- Geltungsbereich/Gültigkeitsbereich
  - sagt aus, in welchen Bereichen eine Variable zugreifbar ist
    - lokal Variable sind nur gültig innerhalb eines Geltungsbereichs z. B. in einem Unterprogramm
    - globale Variable sind in der gesamten Programmeinheit, in der sie definiert wurden, zugreifbar, falls sie nicht durch eine lokale Definition “verdeckt” werden.
      (“Blickt” man aus der Aufrufschachtelung zurück in Richtung Aufrufer, so sind sämtliche globale Daten der aufrufenden Ebenen zugreifbar, es sei denn, sie sind durch lokale Vereinbarungen verdeckt.)

Unterprogramme

- Geltungsbereich/Gültigkeitsbereich(cont)
  - Transfer von Informationen zwischen Geltungsbereichen durch:
    - Parameter
      - Eingangsparameter
      - Ausgangsparameter; Ergebnisparameter
      - beide Richtungen
    - globale Daten
      - im äußersten Geltungsbereich definiert; also programmeinheitsglobal
      - COMMON-Bereiche
        Überall dort gültig, wo definiert; auch über Programmeinheiten hinweg

Unterprogramme

- Parameter
  - Formalparameter (bei der Definition)
  - Aktualparameter (beim Aufruf)

    Zum Zeitpunkt des Aufrufs werden die Formalparameter und die Aktualparameter einander zugeordnet, d.h. eine “Formal-Variable” im Unterprogramm referenziert den gleichen Speicher wie die “Aktual-Variable” im aufrufenden Programm.

  Man unterscheidet
    - positionsgerechte Schreibweise
    - Schlüsselwortschreibweise z. B.

      formalparameter = aktualparameter

    Die gemischte Schreibweise ist möglich. Beginnt man jedoch mit der Schlüsselwortschreibweise, so müssen auch alle folgenden Parameter in Schlüsselwortschreibweise folgen.

Unterprogramme

INTENT-Attribut/Anweisung

- Mit dem INTENT-Attribut kann die Art der Verwendung von Formalparametern spezifiziert werden:
  - Das INTENT(IN)-Attribut besagt, daß der Formalparameter ein Eingabeparameter ist. D.h. dieser Formalparameter darf innerhalb des Unterprogramms nicht redefiniert werden, er darf also z. B. nicht auf der linken Seite einer Zuweisung auftauchen oder Variable einer READ-Anweisung sein.
    Der Aktualparameter kann also sowohl eine Variable als auch eine Konstante sein.
  - Das INTENT(OUT)-Attribut besagt, daß der Formalparameter ein Ausgabeparameter ist. D. h. der Formalparameter muß erst definiert (ein Wert zugewiesen) werden, ehe im Unterprogramm auf ihn zugegriffen wird. Hier **muß** es sich also um eine Variable handeln.

Unterprogramme

INTENT-Attribut/Anweisung

- Mit dem INTENT-Attribut kann die Art der Verwendung von Formalparametern spezifiziert werden (cont):
  - Das INTENT(INOUT)-Attribut besagt, daß der Parameter sowohl Eingabe- als auch Ausgabeparameter sein soll. Auch hier **muß** es sich also um eine Variable handeln.

Wir werden im Rahmen dieses Kurses die Formalparameter mit INTENT-Attributen versehen denn: Keine Angabe = INTENT(INOUT), es sei denn, der Aktual-Parameter ist eine Konstante.

Unterprogramme

- Beispiel:

```
PROGRAM summentest
IMPLICIT NONE
REAL a, b, ergebnis

   Print *, 'Eingabe: summand1 summand2 <r>'
   READ *, a,b

   call summe (a,b,ergebnis)

   PRINT *, 'Ergebnis = ',ergebnis

CONTAINS

      SUBROUTINE summe (summanda, summandb, erg)
      REAL, INTENT(IN) :: summanda, summandb
      REAL, INTENT(out) :: erg
         erg = summanda + summandb
      END SUBROUTINE summe

END PROGRAM summentest
```


Unterprogramme

- FUNKTION

```
[...] FUNCTION fname [(...)]!(FUNCTION-Anweisung)
   [Spezifikationsteil]
   ...
      Funktionsrumpf
      ...
[CONTAINS]
   [interne Unterprogramme]
   ...
END [FUNCTION [fname]]
```

- der wesentliche Unterschied einer Funktion zu einer Subroutine ist, daß der Funktionsname gleichzeitig auch der Name der (lokalen) Ergebnisvariablen ist. Deshalb hat die Funktion einen Typ!
- Der Aufruf einer Funktion kann überall dort stehen, wo auch eine Variable stehen darf; nur auf der linken Seite einer Zuweisung (natürlich) nicht.
- Der Funktionsaufruf darf auch ein separates Statement sein - analog zur Subroutine - allerdings ohne “CALL”; jedoch mit “()”.

Unterprogramme

```
( ~uelinste/tmp/newton_fkt.f90 )

PROGRAM newton
REAL eingabe
! Author: Uwe Linster
! Anfordern und einlesen der Zahl, deren Quadratwurzel
! berechnet werden soll

   PRINT *, ' '

   ! Ausgabe auf das Sichtgerät ohne Zeilenvorschub
   ! Nichtvorrueckende Ausgabe: Kapitel 1.10

   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Zu berechnen ist die Quadratwurzel aus: ')

   READ *, eingabe
   PRINT *, ' '; PRINT *, ' '

   DO WHILE(eingabe <= 0)
      PRINT *, ' Die Eingabedaten entsprechen nicht den &
               &Anforderungen des Programms (Zahl > 0!):'
      READ *, eingabe
   END DO
```


Unterprogramme

```
!Berechnung

   ergebnis = sqroot(eingabe)

!Ergebnisausgabe

   PRINT *, ' Die Quadratwurzel aus ', eingabe ;PRINT *, ' '
   PRINT *, ' ist ', ergebnis
! Oder auch gleich:
!  PRINT *, ' ist ', sqroot(eingabe)

PRINT *, ' '; PRINT *, ' Ende des Programms'
```

Unterprogramme

```
CONTAINS

REAL FUNCTION sqroot (a)
!Berechnung der Quadratwurzel nach dem Newton'schen Verfahren
!Eingabeparameter:a; Startwert der Iteration: 1
!Iterationsformel:x(i+1) = (x(i)*x(i) + a) / (2*x(i))

   REAL , INTENT(IN) :: a
   REAL x
      !a Größe, deren Quadratwurzel zu berechnen ist
      !x Berechnete Approximation der Quadratwurzel

   !Anfangsapproximation
   x = 1
   !Iteration
   DO WHILE (.NOT. (ABS(x*x - a) <= 1E-4))
      x = (x*x + a) / (2.*x)
   END DO
   sqroot = x
END FUNCTION sqroot

END PROGRAM newton
```


Unterprogramme

- COMMON-BEREICHE
  - Durch COMMON-Bereiche werden globale Speicherbereiche spezifiziert. Variable sind in den Geltungsbereichen direkt zugreifbar, in denen die COMMON-Bereiche spezifiziert wurden; auch über verschiedene Programmeinheiten hinweg.
    Man unterscheidet:
    - einen unbenannten COMMON-Block

```
Syntax:COMMON variablenliste
```

    - und beliebig viele benannte COMMON-Blöcke

```
Syntax:COMMON /blockname/ variablenliste [, /blockname/]
              variablenliste]...
```

    - auch die Mischschreibweise ist möglich:

```
Syntax:COMMON  variablenliste [[,]/[blockname]/
              variablenliste]...
```

Unterprogramme

- COMMON-BEREICHE(cont)
  - Eigenschaften
    - Durch Angabe der Variablen wird eine Speicherfolge festgelegt; um Fehler zu vermeiden sollte man mit “copy” und “paste” arbeiten.
    - Variablennamen in verschiedenen Geltungseinheiten dürfen verschieden sein; das sollte man jedoch nur tun, wenn es dafür auch einen plausiblen Grund gibt. Man greift auf dieselbe globale Speicheradresse zu, das sollte man durch den gleiche Variablennamen auch dokumentieren!
    - Initialisiert werden COMMON-Bereiche mit einer BLOCK DATA-Programmeinheit; jedoch nur die benannten Blöcke.
    - Selbstdefinierte Datentypen müssen das Attribut SEQUENCE besitzen. (Speicherfolge!)

Unterprogramme

- COMMON-Bereich - Beispiel (~uelinste/tmp/common_demo_bubble.f90

```
PROGRAM bubble
IMPLICIT NONE

! Zweck des Programms, Loesungsverfahren:
!
! Die Eintraege in der Datei titel sollen in lexikographisch
! sortierter Reihenfolge gedruckt werden. Das Programm geht
! davon aus, dass die Datei maximal 200 Eintraege besitzt und
! jeder Eintrag aus hoechstens 20 Zeichen besteht.
! Als Sortierverfahren wird 'bubble sort' verwendet.
!-------------------------------------------------------------
! Globale Variable; in COMMON-Bereiche spezifiziert
!    name Feld zur Aufnahme der Eintraege aus der Dateri titel
!    max Anzahl der Eintraege
!  fd Kanalnummer (Filedescriptor; Unit)
!-------------------------------------------------------------
! Initialisierung
!
! Ausgabe der Gesamtueberschrift
!
PRINT *, 'Eintraege der Datei titel lexikograpgisch sortiert:'
```

Unterprogramme

```
! Oeffnen der Datei
!
   CALL oeffnen_datei()

! Einlesen der Daten
!
   CALL einlesen()
!
! Sortieren
!
   CALL bubblesort()
!
! Ausgabe des Ergebnisses
!
   CALL ausgeben_screen()
   CALL ausgeben_datei()
```


Unterprogramme

```
! Die Unterprogramme:

CONTAINS

SUBROUTINE oeffnen_datei()

! Oeffnen der Datei "titel" mit Kanalnummer 10
! Global Data:

INTEGER fd
COMMON /datei/ fd
   fd = 10      ! Entfaelt bei BLOCK-DATA-INIT!!
! Local DATA: none

   open(fd, FORM = 'FORMATTED', file = 'titel')

END SUBROUTINE oeffnen_datei
```

Unterprogramme

```
SUBROUTINE einlesen()

! Einlesen der Eintraege aus der Datei "titel" in das Feld !
"name".
! Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

!Local Data:
INTEGER i

   DO i = 1, 200
      READ(fd, 1000, END = 1010) name(i)
1000  FORMAT (a)
   END DO

1010max = i - 1
   PRINT *, 'Gelesene Datensaetze: ',max

END SUBROUTINE einlesen
```


Unterprogramme

```
SUBROUTINE bubblesort()
! Sortiert nach dem "bubble-sort"-Verfahren
! Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

!Local Data

CHARACTER *20 :: hilf
LOGICAL flag
INTEGER i, j, dl

!  flag   zeigt an, ob bei einem Durchlauf Aenderungen in der
!              Anordnung vorgenommen wurden
!              (.FALSE.: Keine Umordnungen
!               .TRUE. : Es wurden Umordnungen vorgenommen)
!  hilf   Hilfsfeld waehrend des Vertauschens zweier Eintraege
!  dl Anzahl der Sortierschleifen
!
```

Unterprogramme

```
! Der Sortiervorgang

   dl = 0
   DO  i = max, 2, -1
      flag = .FALSE.

      DO j = 1, i - 1
          IF (name(j) > name(j + 1)) THEN
               hilf = name(j)
               name(j) = name(j + 1)
               name(j + 1) = hilf
               flag = .TRUE.
               dl = dl + 1
          END IF
        END DO

        if (.NOT. flag) THEN
         PRINT *, 'Anzahl der Durchlauefe: ', dl
         EXIT
      END IF

   END DO

END SUBROUTINE bubblesort
```


Unterprogramme

```
SUBROUTINE ausgeben_screen()

!Global Data:

CHARACTER *20, DIMENSION (200) :: name
INTEGER max
COMMON name, max

! Local Data:

INTEGER i

   DO i=1,max
      Print *, name(i)
   END DO

END SUBROUTINE ausgeben_screen
```

Unterprogramme

```
SUBROUTINE ausgeben_datei()

!Global Data

CHARACTER *20, DIMENSION (200) :: name
INTEGER max, fd
COMMON name, max
COMMON /datei/ fd

! Local Data

INTEGER i

   REWIND (fd)
   WRITE ( fd, 1030 )(name(i), i=1, max)
1030FORMAT (a)

END SUBROUTINE ausgeben_datei

END PROGRAM bubble
```


Unterprogramme

- BLOCK DATA
  - Eine BLOCK-DATA-Programmeinheit
    - ist eine nichtausführbare Programmeinheit, sie dient der Initialisierung von Variablen in **benannten** COMMON-Bereichen.
    - COMMON-Blöcke sind globale Speicherbereiche - gültig auch über Programm-Unit-Grenzen hinweg. Eine Initialisierung darf also Programmsystemweit nur einmal erfolgen. Dies könnte innerhalb einer “ausgezeichneten” Programmeinheit erfolgen oder wie hier in einer speziellen Programmeinheit

```
BLOCKDATA [name]
     [Spezifikationsteil]
     .
     .
     .
  END BLOCKDATA [name]
```

Unterprogramme

- BLOCK DATA(cont)

```
Beispiel
BLOCKDATA car

CHARACTER vorderachse *14, hinterachse *10, motor *11
INTEGER anzahl, seriennummer, partnb

COMMON /ersatzteile/ vorderachse, hinterachse, motor
COMMON /attribute/ anzahl, seriennummer, partnb

DATA vorderachse, hinterachse /'Schwingachsen', `Inkaachsen'/
DATA motor /'Dieselmotor'/
DATA anzahl, partnb /250, 4711/ ! seriennummer bleibt
uninitialisiert

END BLOCKDATA car
```

- Da diese Programm-Unit in einer eigenen Datei liegt muß beim Aufruf des Compilers diese Datei als weiterer Parameter angegeben werden:; z. B.

`f90 hauptprogramm.f90 blockdata.f90 -o programm.out`

Unterprogramme

- BLOCK DATA
  - Beispiel für Programm “bubble”:

Datei: bubble_blockdata.f90

```
BLOCKDATA bubble

   INTEGER fd

   COMMON /datei/ fd

   DATA fd /10/

END BLOCKDATA bubble
```

- Aufruf des Compilers:

```
f90 common_demo_bubble.f90 bubble_blockdata.f90 -o bubble.out
```

Unterprogramme

FORMELFUNKTION

- eine Formelfunktion ist eine Funktion, die nur aus einer einzigen Anweisung besteht und
- sie ist nur innerhalb einer Geltungseinheit aufrufbar.
- Formelfunktionsdefinition:

```
Syntax:ffname() = ausdruck oder
ffname(formalparameter[,formalparameter]...) =
                              ausdruck
```

- Sowohl 'ausdruck' als auch 'formalparameter' sind skalare Datenobjekte.
- Formelfunktionsaufruf:

```
Syntax:ffname() oder
ffname(formalparameter[,formalparameter]...)
```

- Der Typ der Formelfunktion wird separat (wie bei der Typvereinbarung einer Variablen) vereinbart; dieses gilt auch für die Formalparameter.

Unterprogramme

- Formelfunktion - Beispiel (~uelinste/tmp/capslock_fkt.f90)

```
PROGRAM capslock
IMPLICIT NONE
!
! Das Programm liest Kleinbuchstaben ein, wandelt sie
! in Grossbuchstaben um und druckt diese aus.
!  Variable
!
      CHARACTER, DIMENSION(20) :: zeichenin, zeichenout
      INTEGER i

   CHARACTER gross, klein

!  Formelfunktion:

   gross(klein) = CHAR ( ICHAR(klein) + ICHAR('A') - ICHAR('a'))
```

Unterprogramme

```
!  Einlesen der Zeichen

      PRINT *, ' '
   PRINT *, 'Kleinbuchstaben; max. 20; entweder durch "blank"
getrennt -'
   PRINT *, 'Abschluss <r> oder einzeln durch <r> getrennt!'
   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Eingabe: ')
      READ *, (zeichenin(i), i=1,20)

   DO i = 1,20
      zeichenout(i) = gross(zeichenin(i))
   END DO

   PRINT *, 'Ausgabe:',zeichenout

END PROGRAM capslock
```


Unterprogramme

- Formelfunktion - Beispiel (~uelinste/tmp/kredit_fkt.f90)

```
PROGRAM kredit
IMPLICIT NONE
!
!  Das Programm berechnet die Hoehe der monatlichen Raten fuer
! einen vorgebbaren Kaufpreis bei einzugebender Laufzeit. Raten
!  und Zinsen werden von zwei Jahren bis zu einer angebbaren
!  maximalen Laufzeit in Jahresschritten berechnet. Untere und
!  oebere Grenze des Zinssatzes werden ebenfalls eingelesen.
! ------------------------------------------------------------
!  Variable
!
      REAL preis, mrate, p
      INTEGER n, zmin, zmax, zsatz, zs, year, yearmax
!
! Berechnung der Monatsrate fuer die verschiedenen Zinssaetze
!          formel: r*p/(1-(1+r)**(-n)) mit r = zinssatz/12/100
!  hier als Formelfunktion realisiert:

   mrate(zs,p,n) = ((zs/100./12.) * p) / &
                (1.-(1. + (zs/100./12.))**(-n))
```

Unterprogramme

```
!        preis      Kaufpreis (Eingabe)
!       yearmax   maximale Laufzeit der Ratenzahlung (Eingabe)
!        zmin       minimaler Zinssatz (Eingabe)
!        zmax       maximaler Zinssatz (Eingabe)
!        year       Laufvariable fuer die Laufzeit
!        n          Zahl der Raten
!        zsatz      Laufvariable fuer den Zinssatz
! ------------------------------------------------------------
! Initialisierung
!
!  Einlesen von Kaufpreis, maximaler Laufzeit,
!  minimalem und maximalem Zinssatz
!
      PRINT *, ' '
   WRITE (*, 1000, ADVANCE='NO')
1000FORMAT ('Kaufpreis (REAL): ')
      READ *, preis
   WRITE (*, 2000, ADVANCE='NO')
2000FORMAT ('Laufzeit in Jahren (groesser als 2): ')
      READ *, yearmax
     WRITE (*, 3000, ADVANCE='NO')
3000FORMAT ('minimaler und maximaler Zins (in Prozent
(INTEGER)): ')
      READ *, zmin, zmax; PRINT *, ' '
```


Unterprogramme

```
! Aeussere Schleife

    DO year = 2, yearmax

!  Tabellenueberschrift
        PRINT *, ' '
        PRINT *, ' '
      PRINT *, ' Zinsen fuer eine Laufzeit von ', year, ' Jahren'
        PRINT *, ' '
       PRINT *, ' Zins-  Zahl der     Rate          Gesamtzins'
        PRINT *, ' satz    Raten       (DM)              (DM)'

! Innere Schleife
      DO zsatz = zmin, zmax
         n = 12.*year
          PRINT *, ' ', zsatz, '      ', n, ' ', &
               mrate (zsatz, preis, n) &
               ,' ', n * (mrate (zsatz, preis, n)) -preis
      END DO
    END DO
!
!        Abschlussausgabe
      PRINT *, ' ' ;PRINT *, ' Berechnung abgeschlossen.'
END PROGRAM kredit
```

Unterprogramme

- Rücksprung

- Es existieren drei alternative Methoden
  - der “normale” Rücksprung bei Erreichung des END-Statements
  - der “Ausnahme” - Rücksprung, wenn aufgrund einer Bedingung die Aufgabe des Unterprogramms erfüllt ist - etwa analog zum EXIT-Aussprung aus einer DO-Schleife

```
Syntax:RETURN
```

Bei beiden Methoden wird mit dem 1. ausführbaren Statement fortgefahren, das dem Unterprogrammaufruf folgt.

  - der “Ausnahme” - Rücksprung mit wählbarer Fortsetzungsadresse (label).

```
Syntax:RETURN ganzzahliger_ausdruck
```

  - Diese Möglichkeit wird nur der Vollständigkeit erwähnt (Im Falle, daß man mit vorhandenen Programmsystemen umgehen muß). Die Verwendung solcher Konstrukte führt zu undurchsichtigen, nicht mehr wartbaren Programmsystemen.

Unterprogramme

- SAVE-Anweisung

- Mit dem SAVE-Attribut wird der Wert einer **lokalen** Variablen in einem Geltungsbereich über die “Lebensdauer” eines Aufrufs hinaus “gesichert”. D. h., daß der Wert der Variablen auch beim nächsten Aufruf wieder verfügbar ist. Man spricht auch von “*Static”* - Variablen.

- Das SAVE-Attribut ist nur erlaubt bei
  - lokalen Variablen
  - benannten COMMON-Blöcken
    (hier ist Vorsicht geboten, ist doch ein COMMON-Block ein globaler Speicherbereich!?!)

- Das SAVE-Attribut ist insbesondere nicht erlaubt bei
  - einzelnen Variablen, die in einem COMMON-Bereich enthalten sind
  - Formalparametern
  - Funktionsvariablen
  - Namensgruppen (Siehe Kapitel 1.10)

Unterprogramme

SAVE-Anweisung

Attributspezifikation

- gleichzeitig mit der Typvereinbarung

```
REAL, SAVE :: skonto, summe
INTEGER, SAVE :: max
```

- oder getrennt mit einer SAVE-Anweisung

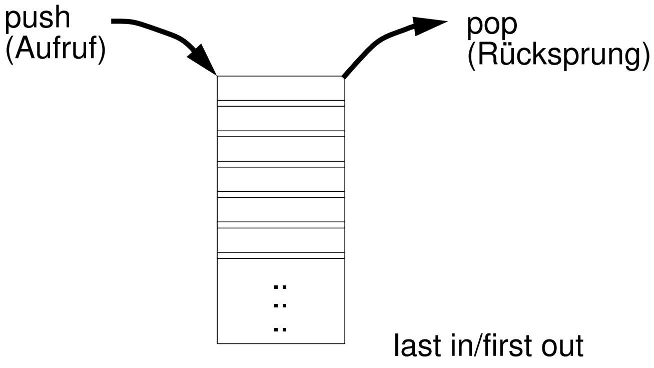
```
REAL skonto, summe
SAVE skonto, summe

INTEGER max
SAVE max
```

Eine SAVE-Anweisung ohne Angabe von Variablennamen bedeutet, das sämtliche lokale Variable in diesem Geltungsbereich diese Spezifikation erhalten.

Rekursive Unterprogramme

Unterprogramme (SUBROUTINE oder FUNCTION) sind dann rekursiv, wenn sie sich selbst direkt oder indirekt aufrufen.

Wenn Unterprogramme das Attribut RECURSION besitzen, muß der Compiler dafür Sorge tragen, daß jeweils bei einem rekursiven Aufruf die lokalen Variablen gerettet und nach dem Rücksprung aus einer tieferen Aufrufebene wieder bereitgestellt werden. Dies ist eine typische Anwendung für einen Kellerspeicher (Stack)!



Rekursive Unterprogramme

- Wann verwendet man rekursive Unterprogramme?
  - Es gibt eine Anzahl von Algorithmen, die "in sich" rekursiv definiert sind. Bei der iterativen Implementierung solcher Algorithmen ist der Programmieraufwand in der Regel beträchtlich und mit einer großen Anzahl von Abfragen behaftet, wie einige Beispiele des Kapitels noch zeigen werden.
  - Als einfaches Beispiel - zum Verständnis - wurde die Berechnung von Fakultäten gewählt, tatsächlich ist es in diesem Fall allerdings effizienter, eine Schleife zu implementieren!

```
PROGRAM recursion
IMPLICIT NONE
INTEGER n, erg
   WRITE (*, FMT = "('n! ; Welches n? : ')", ADVANCE = 'NO')
   READ *, n
   CALL fak (n, erg)
   PRINT *, 'Fakultaet von: ',n ,' = ', erg
   erg = fakfunc (n)
   PRINT *, 'Fakultaet von: ',n ,' = ', erg
CONTAINS
```


Rekursive Unterprogramme

- Rekursive Subroutine

```
RECURSIVE SUBROUTINE fak (fn, ferg)

INTEGER, INTENT (IN) :: fn
INTEGER, INTENT (OUT) :: ferg

   IF (fn == 0) THEN
        ferg = 1
   ELSE
        CALL fak (fn – 1, ferg)
        IF (ferg < 0) THEN
             PRINT *, 'underflow'
             STOP
        END IF
        ferg = fn * ferg
   END IF

END SUBROUTINE fak
```

Rekursive Unterprogramme

▲ Rekursive Funktion

```
RECURSIVE INTEGER FUNCTION fakfunc (fn) RESULT (ferg)

INTEGER, INTENT (IN) :: fn

   IF (fn == 0) THEN
         ferg = 1
   ELSE
         ferg = fn * fakfunc (fn - 1)
         IF (ferg < 0) THEN
               PRINT *, 'underflow'
               STOP
         END IF
   END IF

END FUNCTION fakfunc

END PROGRAM recursion
```


Unterprogramme

■ (Noch) nicht behandelt

- Parameterübergabe bei nicht skalaren Datengrößen (Siehe Kapitel 1.7- Feldverarbeitung)
- Module (Siehe Kapitel 1.11 - Module)
- Externe Unterprogramme (Siehe Kapitel n.b.)
- Nebeneinganspunkte - ENTRY-Anweisungen (Siehe Kapitel n.b.)
- Optionale Formalparameter (Siehe Kapitel n.b.)