Wissenschaftliches Programmieren

Bálint Aradi
http://www.bccms.uni-bremen.de/cms/people/b-aradi

12. Geltungsbereiche, Namensimporte

Quellen:

- Metcalf et al., Modern Fortran explained

Geltungsbereiche

Geltungsbereich (*scope*): Ein Bereich im Quellcode, in dem auf alle in diesem Bereich definierten oder in diesen Bereich importierten Namen zugegriffen werden kann.

```
program Test
  use TestModule
  implicit none

  integer :: ii, jj
  real(dp) :: r1
  :
  call TestSub(ii, jj)
  call TestSub2(...)
  ii = 2 * jj
  kk = 3 *ii
  :
end program Test
```

Scope leer, keine Objekte definiert

Geltungsbereich für das Program-Block

Fehler! kk ist in diesem Scope nicht definiert!

Importiert die Namen, die vom Modul TestModule exportiert werden (dp, TestSub, TestSub2), in den lokalen Geltungsbereich

Fügt neue (lokale) Variablen zum GB hinzu

```
module TestModule
  implicit none
  integer, parameter :: dp = 8
contains
  subroutine TestSub(...)
  end subroutine TestSub

  subroutine TestSub2(...)
  end subroutine TestSub2
end module TestModule
```

Geltungsbereiche in Modulen

```
module TestModule
  implicit none
  integer, parameter :: dp = 8
contains
  subroutine TestSub(arg1, arg2, ...)
    integer, intent(in) :: arg1, arg2

    integer :: ii, jj
    real(dp) :: rTmp
    :
  end subroutine TestSub

  subroutine TestSub2(arg1, arg2)
    :
    call TestSub(...)
    :
  end subroutine TestSub2
end module TestModule
```

G1

G2

G3

- Die Geltungsbereiche sind disjunkt, jede Programmzeile gehört eindeutig zu einem Geltungsbereich.
- Alle Namen (Variablen, Unterprogramme), die im Geltungsbereich des Modules definiert worden sind (G1), automatisch in den Geltungs-bereich der Modulunterprogramme übernommen (G2, G3) (Umgebungszuordnung – *host association*)
- **Umgekehrt gilt das nicht!**

G1: Geltungsbereich von TestModule — `dp, TestSub, TestSub2`

G2: Geltungsbereich von Subroutine TestSub — `arg1, arg2, ii, jj, rTmp + TestSub2, dp`

G3: Geltungssbereich von Subroutine TestSub2 — `arg2, arg2 + TestSub, dp`

Lokale Variablen

- Lokale Variablen eines GBs werden beim Eintreten in den GB neu (meistens auf dem Stapel/Stack) erzeugt und beim Verlassen des GBs wieder vernichtet.

Inhalt lokaler Variablen geht beim Verlassen des GBs verloren!

```
module TestModule
  implicit none
contains
  subroutine TestSub()
    integer :: ii
    write(*, *) ii
    ii = 5
    write(*,*) ii
  end subroutine TestSub
end module TestModule
```

```
program Test
  implicit none

  call TestSub()
  call TestSub()

end program Test
```

Variable ii hat einen nicht definierten Wert (**und zwar beim jedem Aufruf!**)

- *Es kann vorkommen, dass lokalen Variablen immer wieder der selbe Speicherbereich zugeordnet wird und deswegen der alte Wert **zufällig** erhalten bleibt*

Status eines Unterprogrammes speichern

- Wie kann der Status eines Unterprogrammes kann zwischen Aufrufen gespeichert/übergeben werden?

Beispiel:

```
!> Berechne höchsten Fibonacci-Term
!! kleiner als 100.
program TestFibonacci
  implicit none

  integer :: term, oldTerm

  term = 0
  do while (term < 100)
    oldTerm = term
    term = calcNextFibonacci()
  end do
  write(*,*) "Highest term < 100:"
  write(*,*) oldTerm

end program TestFibonacci
```

Damit bei nachfolgenden Aufrufen für nächste Reihenglieder nicht immer die ganze Reihe neu berechnet werden muss, müsste das Unterprogramm calcNextFibonacci() die letzten zwei Reihenmitglieder irgendwo speichern

```
function calcNextFibonacci()
  :
  integer :: nLast, nPrev
  :
end function calcNextFibonacci
```

Wert der lokalen Variablen geht beim Verlassen des Unteprogrammes verloren!

Status eines Unterprogrammes speichern – intent(inout) Variable

- Status wird als intent(inout) Variable beim Aufrufen empfangen und beim Verlassen des Unterprogrammes wieder zurückgegeben

```
program Test
  use TestModule
  implicit none

  integer :: status

  status = 1
  call TestSub(status)
  call TestSub(status)
  call TestSub(status)
  :
end program Test
```

→

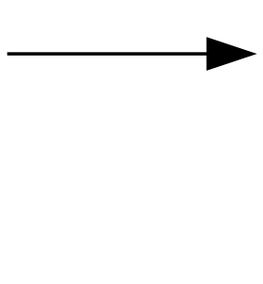
```
1
2
3
:
```

```
module TestModule
  implicit none
contains
  subroutine TestSub(status)
    integer, intent(inout) :: status

    write(*,*) status
    status = status + 1
  end subrouroutine TestSub
end module TestModule
```

Status eines Unterprogrammes speichern – save Variable

- Variable wird mit dem **save-Attribut** versehen:

```
program Test
  use TestModule
  implicit none

  call TestSub()
  call TestSub()
  call TestSub()
  :
end program Test
```



```
1
2
3
:
```

```
module TestModule
  implicit none
contains
  subroutine TestSub()

   integer, save :: status = 0

    status = status + 1
    write(*,*) status

  end subrouroutine TestSub
end module TestModule
```

- Save-Variablen behalten ihren Wert auch nach Verlassen des Geltungsbereiches!
- Initialisierungswert für Save-Variablen kann bei der Deklaration angegeben werden. Dieser Wert wird nur für **Initialisierung beim ersten Aufruf** verwendet.
- Bei **nachfolgenden Aufrufen** werden sie **mit dem vorangehenden Wert** initialisiert.

Status eines Unterprogrammes speichern – save Variable (#2)

Ordnet man einer Variable bei der Deklaration einen Wert zu, erhält die Variable automatisch das save Attribut

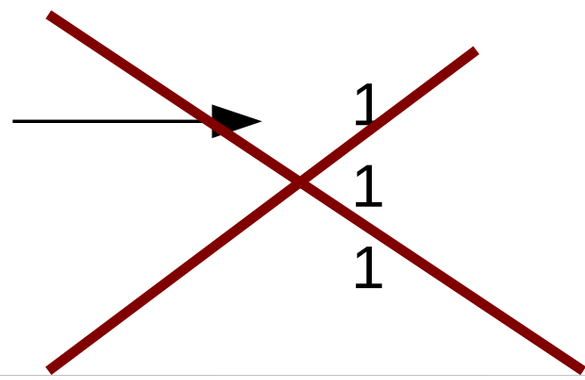
```
module TestModule
  implicit none
contains
  subroutine TestSub()

   integer, save :: status = 0

    status = status + 1
    write(*,*) status

  end subrouroutine TestSub
end module TestModule
```

Äquivalent

```
module TestModule
  implicit none
contains
  subroutine TestSub()

    integer :: status = 0

    status = status + 1
    write(*,*) status

  end subrouroutine TestSub
end module TestModule
```

```
1
2
3
:
```

- Das unterscheidet sich wesentlich von anderen Programmiersprachen wie C/C++ etc.
- **Zur Klarheit sollte das save Attribut immer explizit angegeben werden**



Status eines Unterprogrammes speichern – Modulvariable

- Status eines Modulunterprogrammes kann auch in einer Modulvariable gespeichert werden
- Die Modulvariablen können von allen Modulunterprogrammen gelesen und modifiziert werden!

```
module TestModule
  implicit none

  integer :: status = 0

contains

  subroutine TestSub()
    status = status + 1
    write(*,*) status
  end subroutine TestSub

end module TestModule
```

```
subroutine TestSub2()
  status = status + 1
  write(*,*) status
end subroutine TestSub2
```

TestSub2 würde die selbe Variable manipulieren, wie TestSub!

- Modulvariablen können auch von Programmen/Modulen modifiziert werden, die das Modul importieren!

```
program Test
  use TestModule
  implicit none

  status = 2
```

Wertzuweisung der importierten Modulvariable

Private Modulvariablen

- Private Modulvariablen können nur **von den Unterprogrammen des Moduls** gelesen/geschrieben werden:

```
module TestModule
  implicit none

  integer :: status = 0
  private :: status

contains

  subroutine TestSub()
    status = status + 1
    write(*,*) status
  end subroutine TestSub

end module TestModule
```

```
program Test
  use TestModule
  implicit none

  write(*,*) status
  status = 2
```

Fehler! Private Modulvariable kann weder gelesen noch geschrieben werden.

Es sollte jede Modulvariable als privat deklariert werden, bei der eine Veränderung durch andere Programmteile nicht erwünscht ist. **(Abkapselung)**

Private Modulunterprogramme

- Ähnlich zu Modulvariablen können auch Modulunterprogramme als privat deklariert werden:

```
module TestModule
  implicit none

  private :: helper

contains

  subroutine Sub()
    :
    call helper()
    :
  end subroutine Sub

  subroutine helper()
    :
  end subroutine helper

end module TestModule
```

```
program Test
  use TestModule
  implicit none

   call Sub()
   call helper()

end program Test
```

Fehler! Privates Modulunterprogramm kann außerhalb des Moduls nicht aufgerufen werden.

OK. Modulunterprogramme sehen alle Objekte, die im Geltungsbereich des Wirtsmoduls definiert wurden.

Verkapselung

Grundsätzlich sollten nur diejenigen Objekte eines Moduls von außen zugreifbar sein, die für die Kommunikation des Modules mit anderen Programmteilen verantwortlich sind. Alles andere sollte als privat Deklariert werden.

- In der Praxis: Man deklariert am besten das Modul generell als *private* und kennzeichnet die zu exportierenden Namen als öffentlich (*public*)

Ab hier hat jeder Eintrag das Attribut „private“

Die Einträge Sub, limit werden explizit exportiert

```
module TestModule
  implicit none
  private

  integer, parameter :: limit = 100
  integer :: state

  public :: sub, limit

contains
  subroutine sub()
    :
  end subroutine sub

  subroutine helper()
    :
  end subroutine helper
end module TestModule
```

```
program Test
  use TestModule
  implicit none

  write(*,*) limit
  call sub()
  write(*,*) state
  call helper()
```

OK, public Einträge!

Fehler, private Einträge!

Verkapselung (#2)

- Wenn es wichtig ist, dass Modulvariablen nur in kontrollierter Weise von draußen verändert werden können, sollten sie als private deklariert werden und nur durch public Modulunterprogramme manipuliert werden.

```
module TestModule
  implicit none
  private

  integer :: status  ! Must be >= 0

  public :: setStatus

contains

  subroutine setStatus(value)
    integer, intent(in) :: value

    if (value >= 0) then
       status = value
    else
      write (*,*) "Bad status value"
    end if
  end subroutine setStatus
```

```
program Test
  use TestModule
  implicit none

  call setStatus(1)
  call setStatus(-1)
  :
```

Fehlermeldung zur Laufzeit

Wäre status nicht „private“ gewesen, hätte man die Variable unkontrolliert verändern können.

```
program Test
  use TestModule
  implicit none

  status = -1
  :
```

Inkonsistenter Wert, aber keine Fehlermeldung!

Verkapselung (#3)

- Private Modulvariablen können über entsprechende Routinen abgefragt werden:

```
module TestModule
  implicit none
  private

  integer :: status  ! Must be >= 0

  public :: setStatus, getStatus

contains

   :

  function getStatus()
    integer :: getStatus

    getStatus = status

  end function

end module TestModule
```

```
program Test
  use TestModule
  implicit none

  call setStatus(1)
  write (*,*) getStatus()
  :
```

- Variable von unkontrolliertem Zugang geschützt
- Wert kann in kontrollierter Weise via setStatus() (**setter**) gesetzt werden.
- Wert kann in kontrollierter Weise via getStatus (**getter**) gelesen werden.

Verkapselung (#4)

- **Problem:** Beim Aufrufen der get-Routine wird evtl. eine Kopie der Modulvariable erstellt. Wenn die Modulvariable ein großes Array ist, ist das langsam und ineffizient.

- **Lösung:** Modulvariable mit **protected** Attribut:
  Außerhalb des Moduls lesbar (*wenn gleichzeitig auch public*), aber nur von den Routinen im Modul veränderbar

```
module TestModule
  implicit none
  private

  public :: setStatus, status

  integer, protected :: status

contains

  subroutine setStatus(value)
    integer, intent(in) :: value
    :
    status = value
    :
```

```
program Test
  use TestModule
  implicit none

  call setStatus(1)
  write (*,*) status
  :
```

- keine get-Routine nötig, Variable kann direkt gelesen werden.
- set-Routine nötig, da Variable nicht direkt geschrieben werden kann.
- protected-Attribut muss zusätzlich zum public-Attribut deklariert werden.

Namenskonflikte bei Import

- Ein Name darf in einem Geltungsbereich nur einmal definiert werden.
- Es kann keine lokale Variable deklariert werden, die schon (via use-Befehl) importiert wurde.

```
module TestModule
  implicit none
  private

  public :: limit

  integer :: limit
  :
```

public Deklaration kann vor tatsächlichen Variablendeklaration stehen

```
program Test
  use TestModule
  implicit none

  integer :: limit
  :
```

Fehler! Name bereits definiert!

Umbenennung beim Import

- Wenn ein Name aus einem importierten Modul mit einer lokalen Name kollidieren würde, kann der Moduleintrag beim Importieren umbenannt werden
- Der Name wird nur im lokalen Geltungsbereich geändert

```
program Test
  use TestModule, testLimit => limit
  implicit none

  integer :: limit

  limit = 12
  testLimit = 24
  write(*,*) "Local limit:"
  write(*,*) limit
  write(*,*) "Module limit:"
  call printLimit()

end program Test
```

Limit im lokalen GB als testLimit

```
 Local limit:
          12
 Module limit:
          24
```

```
module TestModule
  implicit none
  private

  public :: limit, printLimit

  integer :: limit

contains

  subroutine printLimit()
    write(*,*) limit
  end subroutine printLimit

end module TestModule
```

Name „limit“ aus dem Geltungsbereich des Moduls automatisch übernommen (*Umgebungszuordnung*)

Selektierter Import

- Werden nur gewisse Einträge aus einem Modul gebraucht, kann der Import via „**only**“ eingeschränkt werden:

```
program Test
  use TestModule, only : printLimit, &
    &testLimit => limit
  implicit none

  integer :: limit

  limit = 12
  testLimit = 24
  limit2 = 5
  write(*,*) "Local limit:"
  write(*,*) limit
  write(*,*) "Module limit:"
  call printLimit()

end program Test
```

```
module TestModule
  implicit none
  private

  public :: limit, printLimit, &
    &limit2

  integer :: limit, limit2

contains
  :
end module TestModule
```

Fehler! Name limit2 wurde nicht importiert.

Schränkt import auf printLimit und limit ein, limit wird in diesem Geltungsbereich in testLimit umbenannt

Verdecken von Namen

- Wenn Namen aus der Umgebung **automatisch** übernommen werden (*Umgebungszuordnung – host association*), verdecken lokale Variablen die Umgebungsvariablen mit identischen Namen:

```
module TestModule
  :
  integer :: limit, limit2

contains

  subroutine Test()

    integer :: limit

    limit = 12
    limit2 = 24

  end subroutine Test

end module TestModule
```

Variablen im Geltungsbereich des Modules

Modulvariablen limit und limit2 via Umgebungszuordnung automatisch übernommen

Lokale Variable „limit“ wird erzeugt und verdeckt Modulvariable „limit“

Wertzuweisung der **lokalen** Variable (Modulvariable limit bleibt unverändert!)

Wertzuweisung der Modulvariable

Ende des Unterprogrammes, lokale Variable „limit“ wird vernichtet

Zusammenfassung

- Es sollten nur diejenigen Objekte eines Moduls exportiert werden, die zur Kommunikation mit anderen Programmteilen notwendig sind. (private Variablen)
- Es sollten nur diejenigen Teile eines Moduls importiert werden, die tatsächlich gebraucht werden (use only).

Aufgabe (Fortgeschrittene Programmierung)

```
program Fibonacci
  use FibonacciIterator
  implicit none

  integer :: order, maxval, nextTerm

  lpMain: do
    write(*,"(A)", advance="no") "> Order (0 or negative number to stop): "
    read (*,*) order
    if (order <= 0) then
      exit
    end if
    write(*,"(A)", advance="no") "> Maximal value for the last element: "
    read(*,*) maxVal
    call Fibonacci_init(order)
    write(*,"(A)") "Fibonacci terms:"
    nextTerm = Fibonacci_getNext()
    do while (nextTerm < maxVal)
      write(*,*) nextTerm
      nextTerm = Fibonacci_getNext()
    end do
  end do lpMain

end program Fibonacci
```

Aufgabe (Fortgeschrittene Programmierung)

- Schreiben Sie ein Modul FibonacciIterator, der mit dem angegebenen Fibonacci-Program zusammenarbeitet.
- Der interner Status des Iterators (Anzahl der bisher berechneten Elemente, Felder zum festhalten der letzten Elemente, etc.) soll in privaten Modulvariablen gespeichert werden.
- Die nötigen Felder sollen in der Routine initFibonacci allokiert werden.
- Implementationsoptimierung:Versuchen Sie den Iterator so zu verwirklichen, dass die Größe des zum Speichern gewisser Reihenelemente allokierten Feldes nur (order+1) ist. (order = Ordnung der Fib.Reihe)
- Implementationsoptimierung: Versuchen Sie den Iterator so zu verwirklichen, dass niemals ein Element des Feldes auf eine andere Position in diesem Feld geschrieben wird. (Es sollten keine Elemente innerhalb des Feldes hin- und hergeschoben werden.)
- Algorithmusoptimierung: Versuchen Sie den Iterator so zu verwirklichen, dass die Berechnung des nächsten Terms ohne Summierung auskommt. Die Berechnung sollte (außer eventuellen arithmetischen Operationen zur Positionsbestimmung in einem Feld) maximal zwei arithmetische Operationen enthalten.