

Modularisierung, Interfaces und ADT

Sonderübung

Dana Liebscher

PR10

28.11.2021

1. Wiederholung Unterprogramme
2. Wiederholung Modularisierung
 - Prinzip der Modularisierung
 - Vorteile der Modularisierung
3. Schnittstellen (INTERFACE-Blöcke)
 - Generische Schnittstellen
 - Übungsaufgabe
4. ADT - Abstrakte Datentypen
 - Allgemeine Syntax
 - Nutzen von ADT
 - Komponentenzugriff
 - Typkonstruktor
 - Übungsaufgabe

Zusammenfassung Unterprogramme

- **Funktionen**

- liefern ein Ergebnis, d. h. besitzen einen Rückgabewert
- Funktionsaufrufe stehen in Ausdrücken

- **Subroutinen**

- liefern **kein** Ergebnis
- werden mittels CALL-Anweisung aufgerufen

- **Rekursive UP** rufen sich selbst (direkt oder indirekt) auf

- In Fortran gibt es nur **Referenzparameter** (d.h. formales Argument wird mit aktuellem Argument assoziiert = Alias)

- Attribute für formale Argumente sind:

INTENT(IN), **INTENT(OUT)** und **INTENT(INOUT)**

Was ist Modularisierung?

- Idee: komplexes System nach dem Baukastenprinzip aus Einzelbausteinen zusammensetzen
- findet im alltäglichen Leben Anwendung, aber auch in vielen Bereichen der Informatik (insbesondere bei der Entwicklung von Programmen)
- UPe entsprechen einfacher Umsetzungen des Modularisierungsprinzips
- UPe als Bausteine bei der Entwicklung von Programmen verwendbar, falls die UPe als eigenständige Verarbeitungseinheiten konzipiert sind

Was ist Modularisierung?

Grundlegende Struktur eines Moduls

```
MODULE < Modulname >  
    IMPLICIT NONE  
    PRIVATE          ! anything that is not declared PUBLIC is PRIVATE  
    PUBLIC :: < Exportliste >  
    ...  
    CONTAINS  
        < UPe >  
END MODULE [< Modulname >]  
  
PROGRAM < Programmname >  
    USE < Modulname >  
    IMPLICIT NONE  
    ...  
END PROGRAM [< Programmname >]
```

Vorteile von Modularisierung

- **Strukturierung** von Programmen: strukturierte Programme können schneller verstanden und leichter überarbeitet werden
- **Vermeidung von Coderedundanz**: gleicher Programmcode muss nicht mehrfach geschrieben werden
- **Wiederverwendung** von Programmteilen: unabhängige Programmteile in verschiedenen Programmen verwendbar, somit schnellere und zuverlässigere Entwicklung von Programmen
- **Kollaboration**: mehrere Programmierer können unabhängig voneinander an verschiedenen Programmteilen arbeiten

Schnittstellen (INTERFACE-Blöcke)

Spezifische Schnittstellen

eindeutig definiert durch:

- Information, ob Subroutine oder Funktion
- Prozedurnamen/ Operatornamen bzw. -symbol/ Zuweisungssymbol (=)
- Liste der Argument- bzw. Operandentypen
- nur bei Funktionen: Ergebnistyp

Generische Schnittstellen und Überladung

- Überladung = mehrfache Verwendung von Prozedurnamen bzw. Operatornamen/ -symbolen (+, −, *, ..., auch: =) für verschiedene Prozeduren
- jede Überladung muss eindeutige **Schnittstelle** haben

Generische Schnittstellen und Überladung

Bei der Überladung eines Operators, müssen die Funktionsargumente immer das Attribut **INTENT(IN)** haben!

```
INTERFACE < generi_name >  
    MODULE PROCEDURE <function_1> [, ... , <function_i>]  
                        bzw. <subroutine_1> [, ... , <subroutine_i>]  
END INTERFACE
```

```
INTERFACE OPERATOR (+)                oder: -, *, /, <, .NOT., ... , .MYOPE.  
    MODULE PROCEDURE <function_1> [, ... , <function_i>]  
  
END INTERFACE
```

```
INTERFACE ASSIGNMENT (=)  
    MODULE PROCEDURE <subroutine_1> [, ... , <subroutine_i>]  
END INTERFACE
```


Generische Schnittstellen und Überladung - ein Beispiel

```
MODULE shifted
  IMPLICIT NONE
  ...

  INTERFACE OPERATOR (.S.)
    MODULE PROCEDURE SHIFT
  END INTERFACE
  ...

CONTAINS

  FUNCTION shift(word,Anzahl)
    CHARACTER(*), INTENT(IN) :: word
    INTEGER, INTENT(IN) :: anzahl
    CHARACTER(LEN(word)):: shift

    shift=' '
    shift(anzahl+1 : len(word)) = word(1 : len(word) - anzahl)
    shift(1 : anzahl) = word(len(word) - anzahl + 1 : len(word))
  END FUNCTION
  ...

END MODULE
```

Übungsaufgabe

Überlade in dem Taschenrechner die iterativen Funktionen, sodass diese wie folgt aufgerufen werden können:

- **.SUM. n** für die Dreieckszahl von n
- **n .UEBER. k** für den Binominalkoeffizienten $\binom{n}{k}$

Abstrakte Datentypen (auch: benutzerdefinierte Typen)

- definiert heterogene Datenstruktur
- heterogen bedeutet: Komponenten können unterschiedliche Typen haben

Allgemeine Syntax

```
TYPE < Typname >
```

```
  [PRIVATE]
```

```
    < Typvereinbarung für Komponenten >
```

```
END TYPE
```

Abstrakte Datentypen (auch: benutzerdefinierte Typen)

Beispiel

```
TYPE fraction
  INTEGER :: nominator
  INTEGER :: denominator
END TYPE
```

Die Komponententypen können verschieden sein:

```
TYPE bike
  CHARACTER(15) :: brand
  INTEGER :: size
  ...
END TYPE
```

Nutzen von ADT

- möglichst viel von der inneren Struktur verstecken
- Informationen, die zusammen gehören, aber unterschiedliche Datentypen haben, an einem Ort speichern (Zum Beispiel wird jeder Student über Name, s-Nummer und Matrikelnummer gekennzeichnet. Diese Infos sollten daher zusammen gespeichert werden.)
- erstellen von Datentypen, die in der Praxis gebraucht werden, aber nicht in F95 vorhanden sind (z.B. Rationale Zahlen, Intervallarithmetik, ...)

Komponentenzugriff

Problem: Es gibt fast keine intrinsischen (d.h. vordefinierte) Funktionen dazu
→ Alle gewünschten Funktionalitäten müsst ihr selbst programmieren!

Dazu benötigt ihr den **Komponentenzugriff**:

- die einzelnen Komponenten besitzen intrinsische Datentypen (d.h. Real, Integer, ...) und für diese gibt es intrinsische Funktionen
→ ihr könnt damit arbeiten
- Allgemeine Syntax: **<NameDesADT> % <NameDerKomponenten>**

Beispiel für Komponentenzugriff

Beispiel

```
TYPE Student
  INTEGER :: Alter
  CHARACTER :: Vorname, Name
END TYPE
```

...

```
FUNCTION aelter (a,b)
  TYPE(Student) :: a,b
  LOGICAL :: aelter

  aelter = a%Alter > b%Alter
END FUNCTION
```

Typkonstruktor

Um im Programmablauf Variablen eures ADTs zu definieren, ohne sie einlesen zu müssen, eine selbstgeschriebene Funktion zu verwenden oder sie komponentenweise belegen zu müssen, könnt ihr den Typkonstruktor verwenden.

Allgemeine Syntax: $\langle \text{NameDesADT} \rangle (\langle \text{Komponente1} \rangle, \langle \text{Komponente2} \rangle, \dots)$

Beispiel

```
TYPE Student
  INTEGER :: Alter
  CHARACTER(8) :: Vorname, Name
END TYPE
```

```
TYPE(Student) :: MaxMuster
```

```
MaxMuster = Student(20,'Max','Muster')
```


Übungsaufgabe

Wir möchten ein Teilnehmerverzeichnis für unsere Sonderübung anlegen. Dabei sollen zu jedem Teilnehmer folgende Angaben gespeichert werden:

- Vorname (max. 30 Zeichen),
- Name (max. 30 Zeichen),
- Alter,
- Semester

Schreibe dafür ein Modul mit einem passenden ADT und folgenden Funktionalitäten:

- eine **generische** (= mit generischem Interface) **Subroutine PUT**, die folgenden Satz ausgibt: "*Max Muster, 20 Jahre, ist Student im 1. Semester.*"
- eine **Funktion**, die die Studiendauer zweier Studenten vergleicht
- eine **Funktion**, die Nach- und Vornamen zweier Studenten lexikalisch vergleicht

Nutze dieses Modul in einem passenden Hauptprogramm.