hochschule mannheim

Fakultät I

Workshop

Eine alte Programmiersprache: Fortran

Informatikworkshop (IWS)

Janina Hamm, Can Arsoy, Richard Steinbrecht, Max Heidinger, Christopher Rotter

03.03.2021

hochschule mannheim

Agenda

1. Was ist Fortran?
2. Geschichte
3. Basics
   - Lochkarten
   - Einfaches Programm
   - Variablen & Datentypen
   - I/O
   - Verzweigungen
   - Schleifen
10 min Pause
4. Modularität
5. Arrays
6. Objektorientierung
40 min Pause
7. Parallelisierung
8. High-Performance-Computing
9. Diskussionsrunde

Das Fortran-Team

Janina Hamm

Max Heidinger

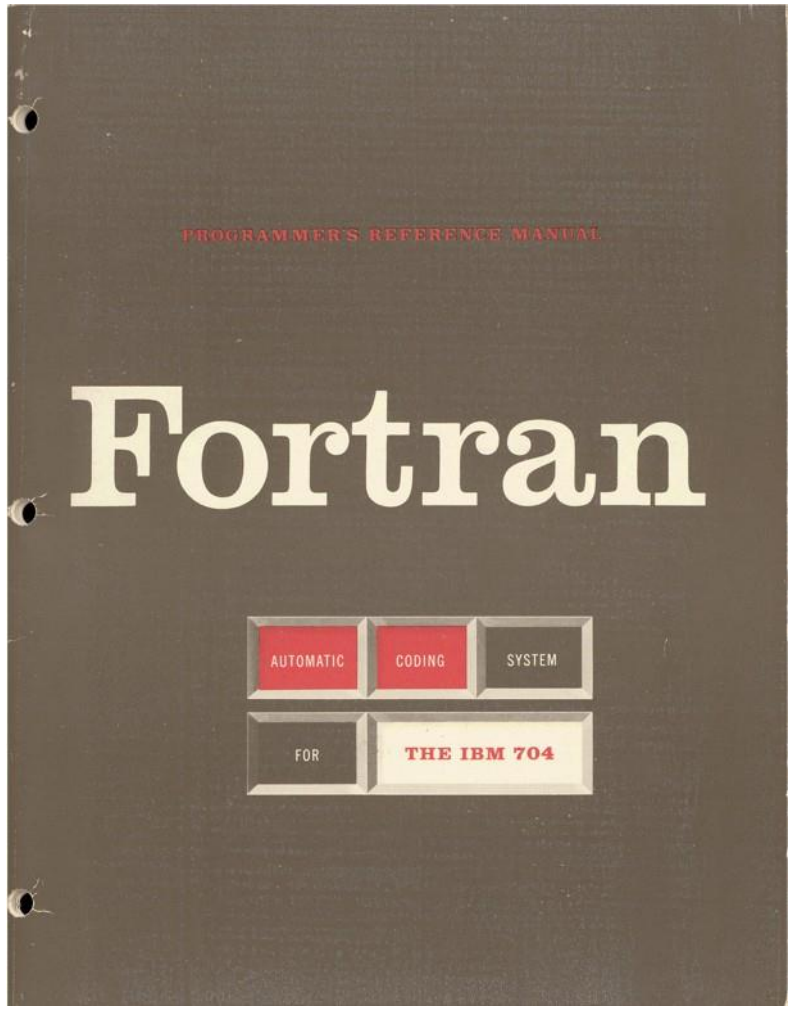
Christopher Rotter

Richard Steinbrecht

Can Arsoy

Was ist Fortran?

- Höhere Programmiersprache
- Prozedurale
- Imperative
- Objektorientierte (Ab 2003)
- Starke Typisierung (Explizite & Implizite)
- Sehr alte Sprache die immer wieder angepasst wurde



Verwendungszweck von Fortran

- Verwendung im mathematischen und numerischen Bereich konzipiert!
  - Spezielle Konstrukte der Sprache, die C/C++ nicht haben
    - Potenzoperator (x**2 = x^2)
    - Automatische Zuweisung des Integer-Datentypes an Variablen I, j, k, l, m,n (implizit)

- Kommt bei theoretischen Berechnungen zum Einsatz und findet sich vor allem auf Großrechnern, die komplexe Berechnungen durchführen wieder

- Seit der Version Fortran 90 bzw. 2003 gibt es in der Sprache objektorientierte Strukturen, die eine breiteren Einsatz erlauben, zum Beispiel im technischen Bereich und im Ingenieurswesen.

Warum Fortran nutzen?

- Für numerische Berechnungen, gibt es wirklich keine bessere Sprache. Der Compiler wurde für die Durchführung von Berechnungen immer wieder optimiert
- Bester Grund ist, dass es eine enorme Menge an existierendem Code für wissenschaftliche Programmierer und numerischen Berechnungen gibt
- Numerical Recipes: Die Kunst des wissenschaftlichen Rechnens ist seit Jahrzehnten ein unverzichtbares Buch für alle wissenschaftlichen Programmierer. Als es 1986 zum ersten Mal herauskam, wurden nur Fortran-Beispielen genutzt
- Es war immer Bestandteile der Neuausgaben! Spezial Ausgaben mit Pascal oder C haben Jahre gedauert

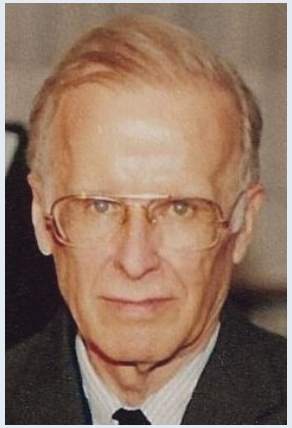
hochschule mannheim

Geschichte

Geschichte

- Weltweit erste Programmiersprache auf hohem Niveau
- Entwicklung durch ein kleines Team unter der Leitung von John Backus bei IBM
- Erste Version wurde 1957 als Coding Tool für IBM 704 genutzt
- Bis heute immer noch die Sprache der Wahl, bei numerischen Berechnungen in Wissenschaft und Technik

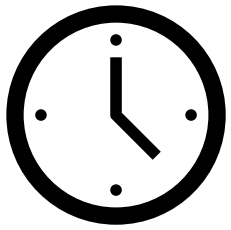
- Pionier der Informatik
- Leiter des Entwicklungsteams der ersten (tatsächlich realisierten) Programmiersprache
- Mit Peter Naur hat er dir Backus-Naur-Form entwickelt, eine Notation zur Beschreibung formaler Sprachen



John Warner Backus

Geschichte - Die Zeit vor FORTRAN

- Programme für die ersten Computer, bestanden aus Sequenzen von numerischen Codes
- Jeder Code stellte eine Operation dar
  - Hole eine Zahl von Speicherplatz X und lege sie in Register A oder addiere die Zahl in Register A zur Zahl in Register B



Zeitaufwendig

Fehleranfällig

Schwer ausfindig

Geschichte - Die Zeit vor FORTRAN

- 1950 war es möglich, Programme zu schreiben, die Mnemonics anstelle von numerischen Codes verwendeten
- Durch ein spezielles Programm (Assembler) wurden die Mnemonics in Zahlencodes umgewandelt
- Vereinfachte die Programmierung, aber selbst einfache Programme brauchten dutzende Operationen
- Es war immernoch schwierig Fehler aufzuspüren

Mnemonics

- "Eine Vorrichtung, wie z. B. ein Muster aus Buchstaben, Ideen oder Assoziationen, die dabei hilft, sich an etwas zu erinnern."
- Es wird normalerweise von Assembler-Programmierern verwendet, um sich die Operationen zu merken, die eine Maschine ausführen kann, wie "ADD" und "MUL" und "MOV" usw.
- Dies ist Assembler-spezifisch!

Geschichte - Mnenomics

- Bestandteil der Assembler Syntax
- Jede Zeile beginnt mit einem Label, einem Tab, einem Sternchen oder einem Semikolon
- Jede Zeile besteht aus 4 Felder

```
{label[:]} mnemonic {operand list} {;comment}
```

```
loop:    mov #COUNT,r5    ; get counter
```

Geschichte - Entstehung von FORTRAN

- Schlussfolgerung war, dass Backus eine Programmiersprache erschaffen wollte, die es erlaubt, Berechnungen irgendwie auszudrücken, das dem einer mathematischen Notation ähnelt
- Der Grundpfeiler seiner Idee war, ein Übersetzungsprogramm (optimierter Compiler) zu nutzen, das die Ausdrücke in numerische Codes übersetzen würde

- 1953 begann Backus mit einem kleinem Team die Machbarkeitsstudie
- Drei Jahre später wurde das Handbuch "Mathematical Formula Translating System" vorgestellt, kurz FORTRAN
- IBM stellte darauf den FORTRAN-Compiler für IBM 704 zur Verfügung

IBM 704 Mainframe

Seiteneffekte der FORTRAN Entwicklung

- Geschichtsbucheintrag für die erste Hochsprache der Welt
- Implementierung des ersten optimierten Compilers, der nicht nur FORTRAN Programme in die numerischen Codes der IBM 704 übersetze, sondern auch Codes produzierte, die fast so schnell liefen wie alles, was händisch implementiert wurde
- Anzahl der Anweisungen wurden um den Faktor 20 minimiert
- Erste Programmiersprache die Hardware-unabhängig ist (FORTRAN IV)

Weiterentwicklung der Sprache – FORTRAN I

- Enthielt 32 Anweisungen
- 14 davon beschränkten sich auschließelich auf Ein- und Ausgaben auf Magentbänder, Papier und Lochkarten
- Anweisungen zur Steuerung von Schaltern und Lämpchen an der Operatorkonsole waren auch vorhanden
- Beeinflusst durch den Compiler A-0

- Die Anzahl der Anweisungen für die Übersetzung der Formeln eines Programmes, war nicht wirklich groß
- Sehr unübersichtlich für heutige Programmierer
- Keine wirkliche Abgrenzung von Teilen eines Programme

C ← FOR COMMENT	CONTINUATION	FORTRAN STATEMENT	IDENTIFICATION
STATEMENT NUMBER 1–5	6	7–72	73–80
C		PROGRAM FOR FINDING THE LARGEST VALUE	
C	X	ATTAINED BY A SET OF NUMBERS	
		DIMENSION A(999)	
		FREQUENCY 30(2,1,10), 5(100)	
		READ 1, N,-(A(I), I = 1,N)	
	1	FORMAT (I3/(12F6.2))	
		BIGA = A(1)	
	5	DO 20 I = 2,N	
	30	IF (BIGA-A(I)) 10,20,20	
	10	BIGA = A(I)	
	20	CONTINUE	
		PRINT 2, N, BIGA	
	2	FORMAT (22H1THE LARGEST OF THESE I3, 12H NUMBERS IS F7.2)	
		STOP 77777	

Weiterentwicklung der Sprache – FORTRAN II

- Strukturierung über Unterprogramme und Funktionen nun möglich
- Übergabe von Daten zwischen Programmteilen über Parameterliste (Referenzen) und Rückgabewerten
- Daten konnten in Blöcke zusammengefasst werden durch den Ausdruck COMMON und in Unterprogrammen genutzt werden
- Durch die Bennenung der COMMON-Blöcke, war es nun möglich, strukturierteren Code zu schreiben (Ähnlichkeit von Struct in C)
- Die Ausdrücke FUNCTION, SUBROUTINE, END, RETURN und CALL wurden hinzugefügt
- DOUBLE PRECISION und COMPLEX wurden neue Datentypen

Weiterentwicklung der Sprache – FORTRAN IV

- Erste FORTRAN Version, die nicht Maschinenabhängig war
- Maschinenspezifische Ausdrücke wie READ, INPUT und TAPE wurden entfernt
- LOGICAL Datentypen, logische Boolean Ausdrücke und logische IF-Ausdrücke wurden nun bereitgestellt
- Wurde für den IBM 7030, 7090, 7094 uns später für den 1401 veröffentlicht
- Es wurden System/360 Compiler zur Verfügung gestellt (F,G,H)
- FORTRAN IV wurde zum wichtigsten Werkzeug der Universitäten, um Programmieren den Studenten näher zu bringen

Compiler Version	Speicher
FORTRAN IV F für DOS/360	64K bytes
FORTRAN IV G für OS/360	126K bytes
FORTRAN IV H für OS/360	256K bytes

Weiterentwicklung der Sprache – FORTRAN 66

- Bidlungs eines Komitees der ANSI, um einen Standard für FORTRAN zu entwickeln
- Es sind daruch die Versionen FORTRAN aka FORTRAN 66 und Basic FORTRAN entstanden
- FORTRAN 66 war im Grunde FORTRAN IV mit einigen Aktualisierungen, da es schon der Standard war
- Basic FORTRAN war eine Erweiterung von FORTRAN II (Maschinenspezifische Asudrücke wurden entfernt)
- Das durch den ersten Standard definierte FORTRAN, offiziell als X3.9-1966 bezeichnet, wurde zur ersten Industriestandard Version von FORTRAN
- Intrinsic und EXTERNAL wurden hinzugefügt
- Kommentare und 6-char-lange Identifiers wurden das erste mal genutzt

Häufig genutzte Versionen – FORTRAN 77

- 1969 veranlasste das ANSI Komitee, die Überarbeitung des FORTRAN-Standards 66
- Dadurch wurde im Jahr 1978 der nächste Standard, X3.9-1978 – auch als FORTRAN 77 bekannt - veröffentlicht
- Es wurden viele neue Funktionen hinzugefügt, die die Unzulänglichkeiten von FORTRAN 66 behebten
- IF und END IF Blöcke mit ELSE und ELSE IF wurden standartisiert
- OPEN, CLOSE und INQUIRE haben die I/O Eigenschaften erweitert
- Intrinsics wie LGE, LGT, LLE und LLT, für lexikale Vergleiche von Strings und Ausdrücke wie INCLUDE, IMPLICIT/NONE DO WHILE, CYCLE und END DO, wurden durch das US Verteidigungsministerium angefordert
- Vereinfachung von nicht-numerischen Datentypen durch den Datentyp CHARACTER
- Durch PARAMETER konnte man Variablen in jedem Programmabschnitt anpassen (dynamischer Speicher)

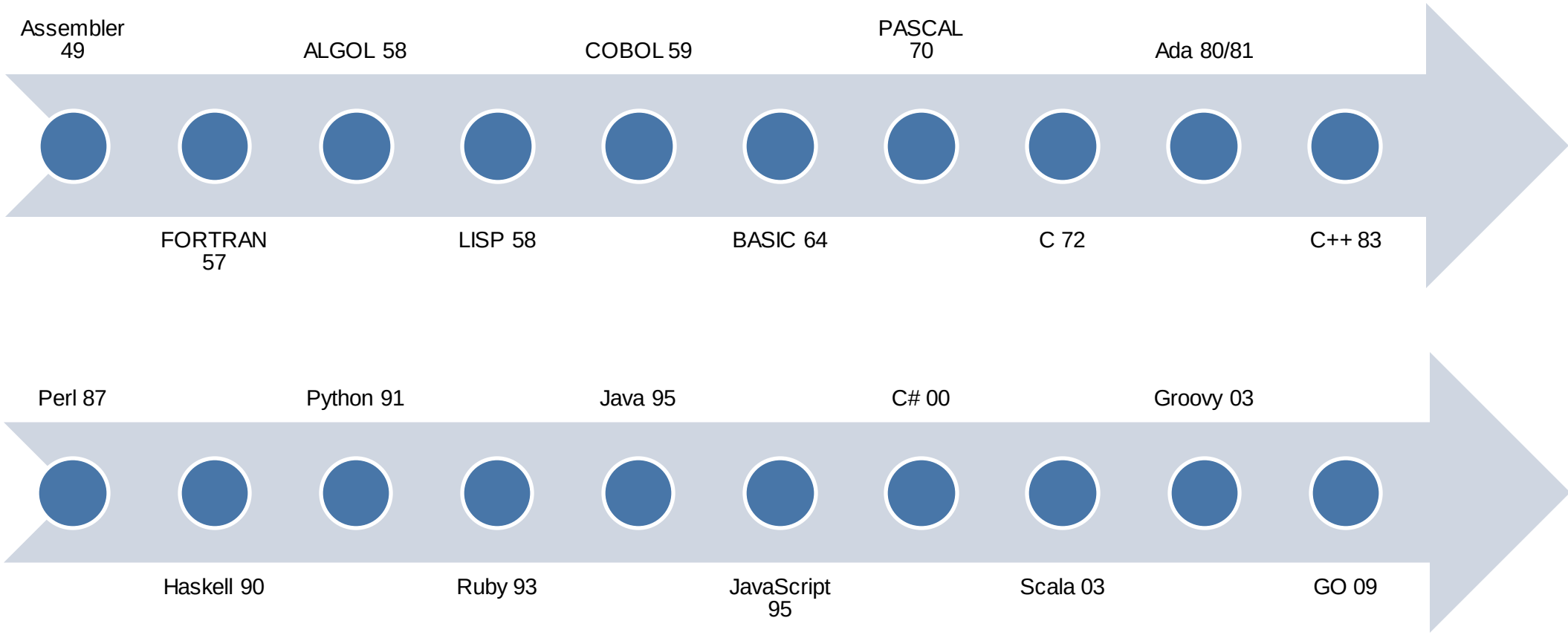
Häufig genutze Versionen – Fortran 90

- Wurde 1991 als ISO Standard und 1992 als ANSI Standard akzeptiert
- Free Format wurde implementiert
- Operationen auf Felder und nicht mehr nur einzelne Elemente
- Verwendung von Modulen mit Zugriffsrechten
- Operationen an ganzen Arrays nun möglich
- RECURSIVE Funktionen unterstütz
- Dynamische Speicherallokation
- SELECT und CASE Konstruktion zur Mehrweg Selektion
- Abgeleitete und abstrakte Datentypen

Fortran 95 - 2018

- Durch das Einbetten von HP-Fortran wurde das parallele Programmieren ermöglicht
- ALLOCATABLE-Arrays werden automatisch deallokiert, wenn sie aus dem Scope sind (Speicherleks werden vermieden)
- AB 2003 volle Unterstützung von OOP
- Asynchrone Ein- und Ausgaben nun möglich
- Erweiterung der parallelen Programmierung
- Einführung von Coarray Fortran (paralleles Ausführungsmodell)
- Rekursive zuweisbare Komponenten als alternative zu Zeigern in abgeleiteten Typen
- Weiterentwicklung im Bereich der Interoperabilität mit C
- Zusätzliche parallele Funktionen in 2018

Sprachen Timeline



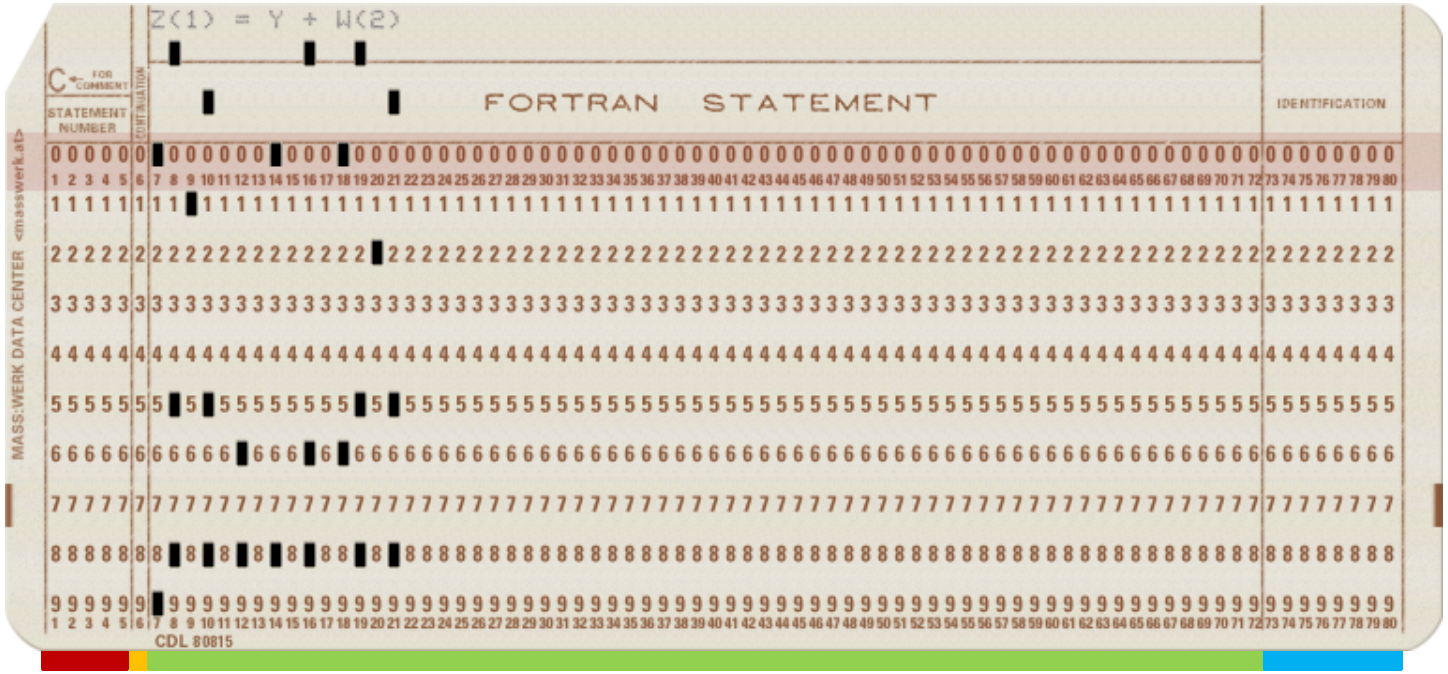
hochschule mannheim

Basics - Lochkarten

Lochkarten

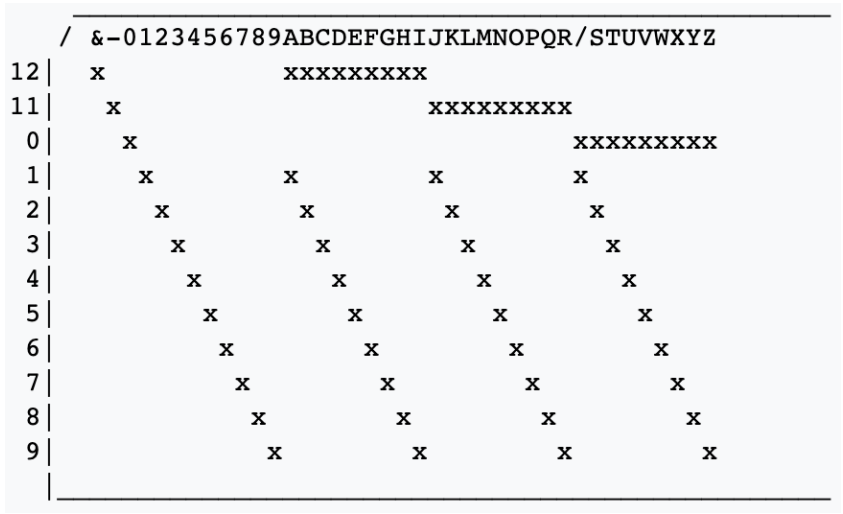
- Haben schon vor Computern existiert => Bekannte Technik von z.B. Spieldosen
- Genutzt für Ein- und Ausgabe als auch für dauerhaften Speicher

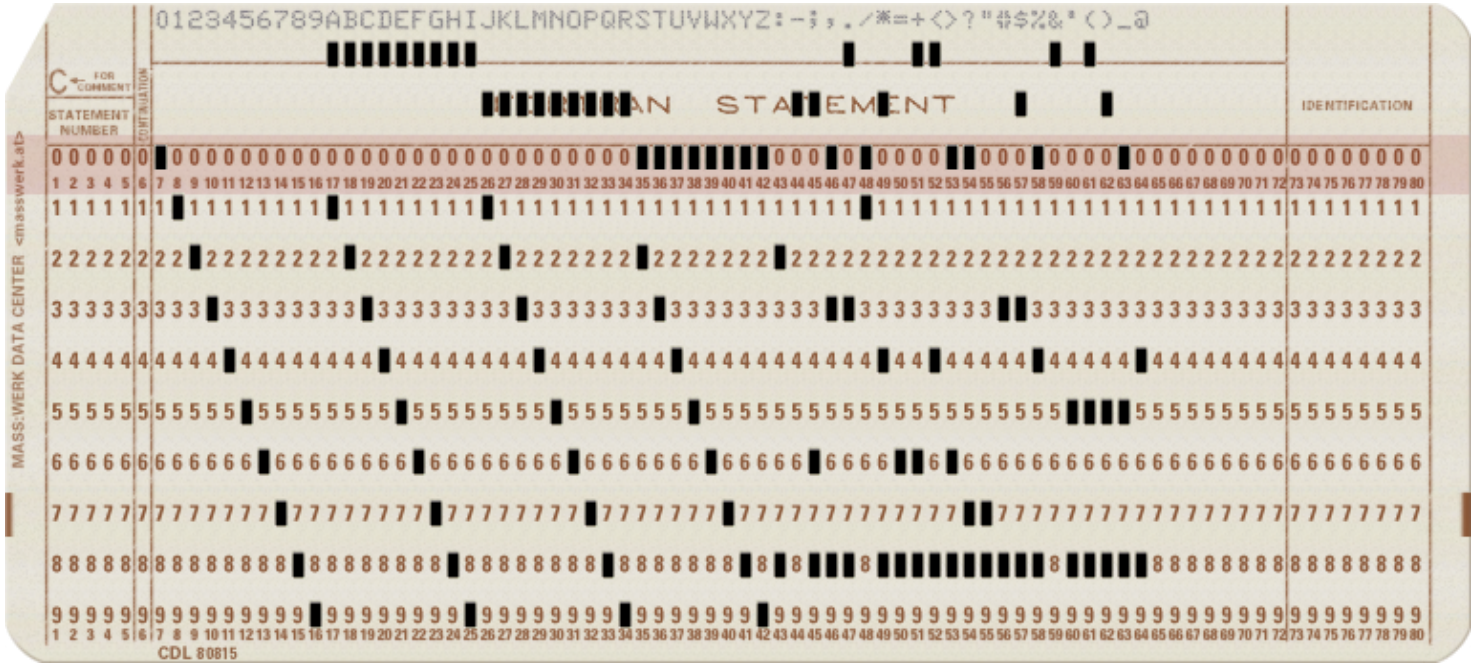
- Eine Lochkarte = eine Zeile Code
- Insgesamt 80 Zeichen:
  - 1-5: Statement Number
  - 6: Continuation Indikator
  - 7-72: Code
  - 73-80: Klassifizierung
- Angabe der Zeichen z.B. in
  - EBCDIC
  - Binärformat



Extended Binary Coded Decimal Interchange Code - EBCDIC

- Art Kodierung der Zeichen auf Lochkarten in zwölf Zeilen

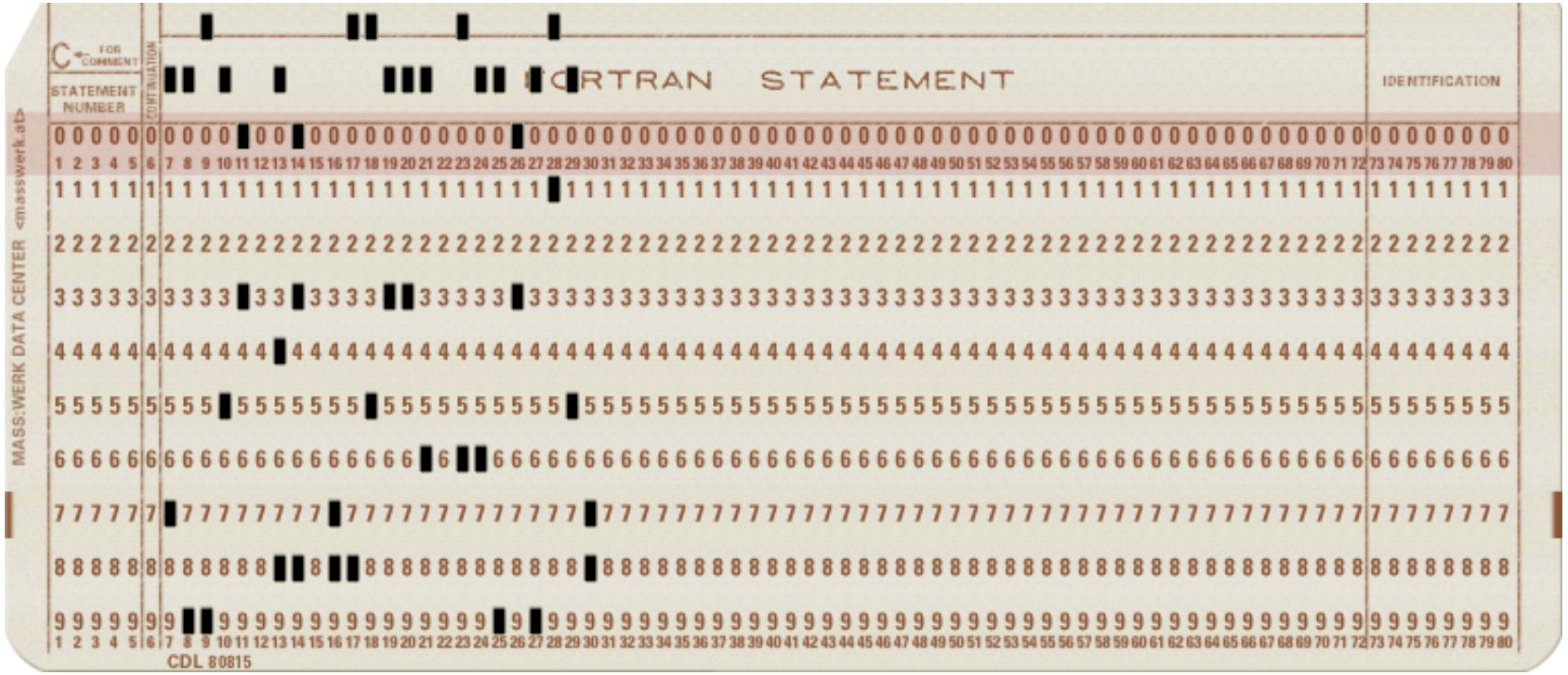




hochschule mannheim

Übung: Lochkarten – 5 min

Experimentieren mit Lochkarten – 5 min



- Erstellen von virtuellen Lochkarten
  - https://www.masswerk.at/keypunch/?c=fortran
- Erstellen, Einlesen und Ausführung von virtuellen Lochkarten
  - https://www.masswerk.at/card-readpunch/

hochschule mannheim

Basics – Einfaches Programm

Programmstruktur Fortran 95

Basis-Struktur

1. program Name
2. Vereinbarungsteil
3. Aktionsteil
4. end [program [Name]]

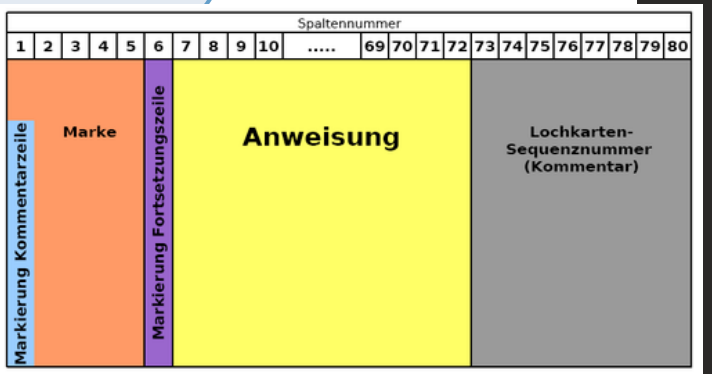
Fortgeschrittene-Struktur

1. program Name
2. Module einbinden (use...)
3. Vereinbarungsteil
4. Aktionsteil
5. Nicht exekutierbare Funktion (function)
6. Interne Unterprogramme (contains)
7. end [program [Name]]

```
! ------------------------------------------------------
! Compute the area of a triangle using Heron's formula
! ------------------------------------------------------

PROGRAM  HeronFormula
   IMPLICIT  NONE

   REAL     :: a, b, c              ! three sides
   REAL     :: s                    ! half of perimeter
   REAL     :: Area                 ! triangle area
   LOGICAL  :: Cond_1, Cond_2       ! two logical conditions

   READ(*,*)  a, b, c

   WRITE(*,*)  "a = ", a
   WRITE(*,*)  "b = ", b
   WRITE(*,*)  "c = ", c
   WRITE(*,*)

   Cond_1 = (a > 0.) .AND. (b > 0.) .AND. (c > 0.0)
   Cond_2 = (a + b > c) .AND. (a + c > b) .AND. (b + c > a)
   IF (Cond_1 .AND. Cond_2) THEN
      s    = (a + b + c) / 2.0
      Area = SQRT(s * (s - a) * (s - b) * (s - c))
      WRITE(*,*) "Triangle area = ", Area
   ELSE
      WRITE(*,*) "ERROR: this is not a triangle!"
   END IF

END PROGRAM  HeronFormula
```

Programmstruktur FORTRAN 77

1. program Name
2. Vereinbarungsteil
3. Aktionsteil
4. end [program [Name]]



Spalte	Inhalt	Bedeutung
1	C oder *	Kennzeichnet eine Kommentarzeile
1 bis 5	Eine Zahl 1 bis 99999	Anweisungsnummer (Marke)
6	Leerzeichen oder 0 (Null)	Beginn einer Anweisung (das ist der Normalfall)
6	sonstiges Zeichen	Fortsetzungszeile (standardmäßig sind bis zu 19 Fortsetzungszeilen erlaubt)
7 bis 72		FORTRAN-Befehl (Anweisung)
73 bis 80	beliebige Zeichen	Kommentar (ursprünglich für Lochkarten-Sequenznummern)

```
      program HeronsFormulaStrict
      real area, real s, real a, real b, real c
      logical Cond_1
      logical Cond_2

c -----------------------------------------------
c Compute the are of a triangle using Heron's formula
c -----------------------------------------------

      read (*,*)  a, b, c

      write (*,*) "a = ", a
      write (*,*) "b = ", b
      write (*,*) "c = ", c
      write (*,*)

      Cond_1 = (a > 0.0) .AND. (b > 0.0) .AND. (c > 0.0)
      Cond_2 = ((a + b) > c) .AND. ((a + c) > b) .AND. ((b + c) > a)
      if(Cond_1 .AND. Cond_2) then
      s = (a + b + c) / 2.0
      area = sqrt(s * (s - a) * (s - b) * (s - c))

      write (*,*) "Triangle are = ", area
      else
      write (*,*) "ERROR: this is not a triangle"
      end if

      end
```

Free Format vs. Strict Format

```
! ------------------------------------------------------
! Compute the area of a triangle using Heron's formula
! ------------------------------------------------------

PROGRAM  HeronFormula
  IMPLICIT  NONE

  REAL     :: a, b, c          ! three sides
  REAL     :: s                ! half of perimeter
  REAL     :: Area             ! triangle area
  LOGICAL  :: Cond_1, Cond_2   ! two logical conditions

  READ(*,*)  a, b, c

  WRITE(*,*)  "a = ", a
  WRITE(*,*)  "b = ", b
  WRITE(*,*)  "c = ", c
  WRITE(*,*)

  Cond_1 = (a > 0.) .AND. (b > 0.) .AND. (c > 0.0)
  Cond_2 = (a + b > c) .AND. (a + c > b) .AND. (b + c > a)
  IF (Cond_1 .AND. Cond_2) THEN
     s    = (a + b + c) / 2.0
     Area = SQRT(s * (s - a) * (s - b) * (s - c))
     WRITE(*,*) "Triangle area = ", Area
  ELSE
     WRITE(*,*) "ERROR: this is not a triangle!"
  END IF

END PROGRAM  HeronFormula
```

```
      program HeronsFormulaStrict
      real area, real s, real a, real b, real c
      logical Cond_1
      logical Cond_2

c -----------------------------------------------------
c Compute the are of a triangle using Heron's formula
c -----------------------------------------------------

      read (*,*)  a, b, c

      write (*,*) "a = ", a
      write (*,*) "b = ", b
      write (*,*) "c = ", c
      write (*,*)

      Cond_1 = (a > 0.0) .AND. (b > 0.0) .AND. (c > 0.0)
      Cond_2 = ((a + b) > c) .AND. ((a + c) > b) .AND. ((b + c) > a)
      if(Cond_1 .AND. Cond_2) then
      s = (a + b + c) / 2.0
      area = sqrt(s * (s - a) * (s - b) * (s - c))

      write (*,*) "Triangle are = ", area
      else
      write (*,*) "ERROR: this is not a triangle"
      end if

      end
```

FORTRAN 77 vs. Fortran 95

- Alle Fortran Schlüsselwörter sollten im Upper-Case geschrieben werden
- Keine Unterscheidung von Groß- und Kleinschreibung (Case-Insensitive)

	Fortran 95
Erlaubte Zeichen	[A-Z a-z], [0-9], [77 Sonderzeichen], [Underscrore _]
Symbolische Namen	Maximal 31 Zeichen Muss immer mit einem Buchstaben beginnen Leerzeichen werden NICHT ignoriert
Kommentare	Kommentar beginnt mit einem !, kann auch am Ende einer Zeile drangehängt werden

	FORTRAN 77
Erlaubte Zeichen	[A bis Z], [0-9], [+-*/=():,.'$ Leerzeichen
Symbolische Namen	Maximal 6 Zeichen Muss immer mit einem Buchstaben beginnen Leerzeichen werden ignoriert
Kommentare	Kommentare beginnen mit einem c in Spalte 1

Kompilieren und Ausführen

Kompilieren

```
cango@cango-UX510UWK:~/Desktop/fortran$ gfortran heronsFormulaStrict.f -o heronsFormulaStrict
```

Ausführen

```
cango@cango-UX510UWK:~/Desktop/fortran$ ./heronsFormulaStrict
2,3,4
 a =     2.00000000
 b =     3.00000000
 c =     4.00000000

 Triangle are =     2.90473747
```

hochschule mannheim

Übung: Basics – 15 min

Übung

1. Directory für Fortranübungen erstellen (Übung 1)
2. Mit Editor eine Fortran95- Datei (.f95) erstellen
3. Programm erstellen, das den Satz des Heron nutzt
4. Kompilieren mit gfortran und Output-Datei erstellen (benennen!)
5. Executable ausführen und Ausgabe testen

Für die ganz Schnellen!

- Programm kann um den Satz des Pythagoras ergänzt werden. Fehlende Seite ausrechnen!

hochschule mannheim

Basics – Variablen & Datentypen

Datentypen

Integer

```
Integer :: VariableName
```

Real

```
Real :: VariableName
```

Character

```
Character :: VariableName
```

Complex

```
Complex :: VariableName
```

Logical

```
Logical :: VariableName
```

Datentypen

```
program Test2
implicit none

  real           :: p, q, realRes
  integer        :: i, j, intRes

  p = 2.0; q = 3.0
  i = 2; j = 3.0

  realRes = p / q
  intRes = i / j

  write(*,*) "Floating number = ", realRes
  write(*,*) "Integer number = ", intRes

end program Test2
```

```
 Floating number =    0.666666687
 Integer number =              0
```

```
program Test
implicit none

  !two byte integer
  integer(kind = 2) :: shortval

  !sixteen byte integer
  integer(kind = 16) :: veryverylongval

  !default integer
  integer :: defval

  write (*,*) "shortval = ", huge(shortval)
  write (*,*) "veryverylongval = ", huge(veryverylongval)

end program Test
```

```
 shortval =   32767
 veryverylongval =  170141183460469231731687303715884105727
```

Datentypen

```
program Test3
implicit none

  character(len=12), parameter :: arg = "FortranKings"

  write (*,*) arg

end program Test3
```

datentyp, parameter :: *symname* = *wert*

Datentyp	Kommentar	Mögliche Werte
logical	Logischer Datentyp (wahr oder falsch)	.TRUE., .FALSE.

Datentyp	Kommentar	Beispiele (Konstanten)
`integer`	Ganzzahlen	15, -6500, 200000000
`real`	Gleitkommazahlen einfacher Genauigkeit	3.1415, -5.5, .7e3, 12.5E-5
(`double precision`)	Gleitkommazahlen doppelter Genauigkeit (aus FORTRAN 77)	3.1415D0, -5.5D0, .7d3, 12.5D-5
`complex`	Komplexe Zahlen (zwei `real` -Zahlen)	(3.1415, -5.5), (1.4, 7.1E4)

Datentyp	Kommentar	Beispiel (Konstante)
character(n)	Zeichenkette (String) mit einer Länge von n Zeichen	'Hallo, Welt!', "Hallo, Welt!"
character(len=n)	-"-	
character*n	-"- (FORTRAN 77-Stil, sollte nicht mehr verwendet werden)	
character	Zeichenkette (String) mit einer Länge von einem Zeichen	'H', "h"

Variablen

Integer

```
Integer :: total

total = 200
```

Real

```
Real :: average

average = 16.67
```

Character

```
Character :: msg

msg = "IWS Rocks"
```

Complex

```
complex:: cx

cx = (3.0, 5.0)
=> 3.0 + 5.0i
```

Logical

```
Logical:: done

done = .true.
```

Variablen

- Symbolischer Name
- Datentyp
- Wert
- Speicherplatz

- K + 2 = X geht nicht, da kein gültiger Ausdruck (Falscher L-Wert)
- Verkettete Mehrfachzuweisung ist nicht möglich (i = j = k = 1.5)
- Deklaration ohne Initialisierung wird die Speicheradresse der Variable hinterlegt

Explizit => Implicit none

```
real              :: p, q, realRes
integer           :: i, j, intRes

p = 2.0; q = 3.0
i = 2; j = 3.0
```

Sollte immer gewählt, da Fehler so minimiert werden!

Implizit => Implicit none fehlt

```
program Test2


  p = 2.0          !Datentyp real wird genommen
  q = 3.0
  i = 2            !Datentp integer wird genommen
  j = 3.0
```

Operatoren

```
integer :: a, b

a = 5
b = 6

write (*,*) A < B
! Ausgabe: T
```

Operator in Fortran 95	Operator in FORTRAN 77	Kommentar
<	.LT.	less than (kleiner als, <)
<=	.LE.	less equal (kleiner gleich, <=)
>	.GT.	greater than (größer als, >)
>=	.GE.	greater equal (größer gleich, >=)
==	.EQ.	equal (gleich, ==)
/=	.NE.	not equal (ungleich, !=)

Operator	Kommentar
.NOT.	logisches NICHT
.AND.	logisches UND
.OR.	logisches ODER
.EQV.	logische Äquivalenz
.NEQV.	logische Antivalenz

```
logical :: bool

bool = .TRUE.

write(*,*) .NOT. Bool
! Ausgabe: F
```

Operatorenpriorität

Die Priorität der arithmetischen Operatoren entspricht den mathematischen Konventionen.

- Klammerung vor allem anderen, z.B. `(a+b)*c` ⇔ `a*c+b*c`
- Exponentiation vor Punktrechnung, z.B. `a*b**c` ⇔ `a*(b**c)`
- Punktrechnung vor Strichrechnung, z.B. `a+b*c` ⇔ `a+(b*c)`

Operator	Kommentar
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
**	Exponentiation

Operator	Kommentar
//	Operator zum Verknüpfen von Strings

```
character(len=4) :: a
character(len=10) :: b

 a = 'How '
 b = 'do you do.'

 write(*,*) a // b
 ! Ausgabe: How do you do
```

hochschule mannheim

Basics – I/O

I/O

- Werkzeuge zum Lesen und Schreiben von formatierten (Spezifikation, wie die Daten für die Eingabe geparst oder für die Ausgabe dargestellt werden sollen) als auch von unformatierten (binären) Daten
- Dateizugriff kann sequentiell oder zufällig erfolgen
- Fortran 2003-Standard: Streaming I/O hinzugefügt
- Hier: formatierte, sequentielle E/A
- Lesen von Daten mit *read*-Befehl, Schreiben von Daten mit *write*-Befehl oder (dem kürzeren) *print*-Befehl (Schreiben auf Standardausgabe)

I/O: Print

- Print ohne Format-Angabe:
  - format=*
  - Automatische Formatierung der Ausgabe

```
! print always writes to standard output
print <format>, <list>
```

- Print mit Format-Angabe:
  - z.B. '(a11,a8)'
  - Formatspezifikation ist Zeichenkette
  - Tatsächliches Format eingeschlossen in Klammern, hier: a11,a8
  - Abhängig von Variablentypen
  - Optional: Zahl, die die Länge des Ausgabefeldes angibt
    - Achtung bei integer!

Variable Type	Edit Descr.	Example	Comment
string	a	a11	A string is printed into an 11 character wide output field
string	a	a	The length of the output field is the length of the string
integer	i	i4	An integer is printed into a 4 character wide output field
integer	i	i	INVALID! The length has to be specified
real	f	f12.5	A real is printed into a 12 character wide output field. The 5 after the decimal point means that 5 decimal digits of the floating point number are to be printed. No exponent is used with this edit descriptor
real	e	e12.5	A real is printed with an exponent
real	es	es12.5	Same as e, but number does not start with a zero
whitespace	x	x	Add a blank character
whitespace	x	4x	Four whitespace characters are added

I/O: Print – Beispiele 1/2

```
character(len=12)       :: c = 'abcdefghijkl'
integer                 :: i = 10000
real                    :: r = 3.14159e2
logical                 :: l = .true.

print '(a)', c          ! abcdefghijkl
print '(a4)', c         ! abcd
print '(i8)', i         ! 10000
print '(i4)', i         ! ****          ! If the number cannot be
                                        ! printed in the available
                                        ! field, asterisks appear

print '(f12.5)', r      ! 314.159
print '(f12.2)', r      ! 314.16
print '(f6.4)', r       ! ******        ! Not enough space again

print '(e12.5)', r      ! 0.314159e3    ! Format with exponent,
                                        ! number starts with zero

print '(es12.5)', r     ! 3.14159e2     ! Format with exponent

print '(l)', l          ! T             ! T or F is printed for
print '(l1)', l         ! T             ! .true. or .false.
```

I/O: Print – Beispiele 2/2

- Formate können wiederholt und gruppiert werden
  - Gruppierung mit Klammern ()
  - Wiederholung mit Zahl vor der Gruppierung

```
print '(3es12.5)', x, y, z          ! Three numbers are printed
                                    ! on one line with the
                                    ! same format (es12.5)

print '(3(es12.5,1x))', x, y, z     ! Same as above, with a
                                    ! whitespace added

print '(3(es12.5,2x))', x, y, z     ! Same as above, with two
                                    ! whitespaces added
```

I/O: Write

- Ähnlich wie die print-Anweisung, aber erlaubt Angabe, wo ausgegeben werden soll
- list: auszugebende Werte/Variablen
- format: Formatspezifikation wie bei print
- unit: Zahl, verweist auf Ausgabeziel
  - 6: Standardausgabe (alternativ: *)
  - 0: Standardfehler
  - 6 und 0 immer zugänglich, müssen nicht geöffnet werden
  - Falls Ausgabe in Datei, muss Datei zuerst geöffnet werden

```
write(<unit>, <format>) <list>
```

I/O: Read

- Ähnlich wie die write-Anweisung
- list: einzulesende Variablen
- format: Formatspezifikation
- unit: Eingabequelle
  - 5: Standardeingabe (alternativ: *)
  - 5 (wie 6 und 0 bei write) immer zugänglich, muss nicht geöffnet werden
  - Falls Eingabe von Datei, muss Datei zuerst geöffnet werden

```
read(<unit>, <format>) <list>
```

I/O: Read – Einlesen von Kommandozeile

- Vorgehen:
  - Variablen definieren
  - Write-Anweisung
    - Default: Zeilenumbruch nach write-Anweisung
    - Zeilenumbruch vermeiden: Hinzufügen von *advance='no'* (Eingabeaufforderung direkt nach Zeichenkette)
  - Read-Anweisung

```
character(len=8) :: name
integer          :: age

write(6,'(a)',advance='no') 'Please enter your age : '
read(5,'(i4)') age
```

I/O: Dateien

- Kanäle für Standardfehler/Eingabe/Ausgabe automatisch geöffnet
- Festplattendateien müssen vor Schreiben/ Lesen geöffnet werden
- unit: Nummer, bei Öffnen an Datei gebunden, benötigt für Lesen/Schreiben der Datei
- Unit-Nummer:
  - Ganzzahl zwischen 0 und 255
  - 0, 5 und 6 bereits Standardfehler/Eingabe/ Ausgabe zugewiesen
  - Steht nach Schließen einer Datei wieder zur Verfügung

```
open(unit=10, file='data.txt')

…

close(10)
```

I/O: Dateien – Parameter für open-Befehl

Element	Kommentar
[unit =] x	x ist eine Dateinummer (Ganzzahl, sollte über 10 liegen, da oft Nummern unter 10 fix zugeordnet sind, z.B. der Standardein-, ausgabe).
file = x	x ist der externe Dateiname
iostat = x	x ist 0 wenn open fehlerfrei ausgeführt wurde, ansonsten eine systemabhängige Fehlernummer
status = x	Dateistatus: 'old' ... Datei existiert bereits 'new' ... Datei wird neu erzeugt 'scratch' ... namenlose temporäre Datei 'unknown' ... System bestimmt Dateistatus selbst 'replace' ... der Inhalt einer bereits vorhandenen Datei wird gelöscht.
access = x	Zugriffsmethode: 'sequential' ... Sequentielle Datei 'direct' ... direkter Zugriff 'stream' ... binärer Zugriff
position = x	Den Dateisatzzeiger beim Öffnen der Datei an eine bestimmte Position setzen. ('asis', 'rewind', 'append')
form = x	Format: 'formatted' oder 'unformatted'
action = x	'read' ... nur Lesezugriff 'write' ... nur Schreibzugriff 'readwrite' ... Lesen und Schreiben
recl = x	Datensatzlänge (positive Zahl, access='direct', in Bytes)
err = x	Im Fehlerfall Sprung zur Marke x
blank = x	'null' oder 'zero' (nur für form='formatted')
delim = x	'apostrophe' 'quote' 'none'
pad = x	'yes' oder 'no' (nur für form='formatted')

Eingestellte Vorgabewerte sind:

- status = 'unknown'
- position = 'asis'
- access = 'sequential'
- form = 'formatted'

- Wichtig für Übung später:
  - unit
  - file
  - iostat

I/O: Dateien – Zeilenweise einlesen

```
open(unit=12, file='data.txt', iostat=ios, status='old')
if (ios /= 0) stop "Error opening file"

do
    read(12,'(a)', iostat=ios) line
    ! Bei EOF wird ios /= 0
    if (ios /= 0) exit
    ...
end do
close(12)
```

- *status:* Dateistatus, wenn 'old', dann vorausgesetzt, dass Datei bereits existiert
- *iostat:* I/O-Status, integer, gibt an ob Dateioperation erfolgreich war
  - ➢ =0: erfolgreich, =/0: nicht erfolgreich
- *stop*: beendet Programm

hochschule mannheim

Basics – Verzweigungen

Verzweigungen: if … then / else if … then / else

- "else if" und "else" ist optional
- "endif" markiert Ende der Verzweigung
- Ablauf:
  - Wird logischer Ausdruck zu *.true.* ausgewertet, dann führe zugehörigen Codeblock aus
  - Wenn nicht, dann werte die nächste "else if"-Anweisung aus
  - Ansonsten führe "else"-Zweig aus

```
if (<logical expression>) then
  <code1>
else if (<logical expression>) then
  <code2>
else
  <code3>
endif
```

==	equal
\=	not equal
<, <=	less than, less than or equal
>, >=	greater than, greater than or equal
.not.	negation
.and.	logical and
.or.	logical or
()	brackets for ordering

Verzweigungen: if-Kurzschreibweise

- Enthält "if"-Klausel nur eine Anweisung, kann kürzere Form verwendet werden

```
if (<logical expression>) <code>
```

Verzweigungen: Arithmetisches IF - 1/2

- 3 Verzweigungen abhängig vom Ergebnis eines arithmetischen Ausdrucks
- Anweisung übergibt Kontrollfluss an eines der sog. Labels im Code
- Wenn Ergebnis des arithm. Ausdrucks:
  - <0: Verwendung Label1
  - =0: Verwendung Label2
  - >0: Verwendung Label3
- Anweisung erfordert drei Labels, erlaubt aber Wiederverwendung von Labels, d.h. Anweisung kann zu 2 Zweigen vereinfacht werden

```
if (<arith_expr>) label1, label2, label3
```

Verzweigungen: Arithmetisches IF – 2/2

- Funktion überflüssig, da gleiche Funktionalität durch if-Anweisung und das if-else-Konstrukt geboten

```
! Mit arithmetischem IF
    if (X) 100, 100, 200
100 print *, "Negative or zero"
200 continue

! Mit IF-Anweisung
if (X<=0) print*, "Negative or zero"
```

```
! Mit arithmetischem IF
    if (X) 100, 110, 120
100 print*, "Negative"
    goto 200
110 print*, "Zero"
    goto 200
120 print*, "Positive"
200 continue

! Mit IF/ELSE-Konstrukt
if (X<0) then
  print*, "Negative"
else if (X==0) then
  print*, "Zero"
else
  print*, "Positive"
end if
```

Verzweigungen: continue

- *continue* ist "do-nothing"-Anweisung
- Lediglich Referenz, zu der man springen kann

```
[label] CONTINUE
```

hochschule mannheim

Basics – Schleifen

Schleifen: do-loop

```
do loop_variable = loop_start, loop_end, loop_increment
  ...
enddo
```

- *loop_increment*
  - optional, kann weggelassen werden wenn es =1 ist
  - kann positiv oder negativ sein (zum Abwärtszählen)

Schleifen: do-while

```
do while (<logical expression>)
   ...
enddo
```

- ermöglicht komplexere Ausgangsbedingungen
- nicht zu verwechseln mit do-while in Java!

Schleifen: Befehl "exit"

- kann in Schleife mit oder ohne Schleifenvariable & Grenzen erscheinen
- erlaubt verlassen der Schleife an beliebiger Stelle
  - I.d.R. in "if"-Anweisung platziert
  - Platzierung am Anfang: while-Schleife
  - Platzierung am Ende: until-Schleife ("do-while")
  - do-exit flexibler als übliches do-while

```
! do mit exit und Schleifenindex
do i = loop_start, loop_end
  if (<logical expression>) exit
  <code>
enddo
```

```
do
  if (<logical expression>) exit
  <code>
Enddo
```

```
do
  <code>
  if (<logical expression>) exit
  <code>
enddo
```

```
do
  <code>
  if (<logical expression>) exit
enddo
```

Schleifen: Befehl "cycle"

- Schleifeniterationen können mit "cycle"-Anweisung übersprungen werden
- Bei Ausführung des "cycle"-Befehls wird sofort die nächste Schleifeniteration gestartet

```
do i = loop_start, loop_end
  <code>
  if (<logical expression>) cycle
  <code>
enddo
```

Schleifen: Bennenung von do-Konstrukten

- do-Konstrukte können benannt werden
- besonders nützlich, wenn es verschachtelte do-Konstrukte gibt

```
do1: do i=1, 5
  do j=1,6

    ! This cycles the j construct
    if (j==3) cycle

    ! This cycles the j construct
    if (j==4) cycle

    ! This cycles the i construct
    if (i+j==7) cycle do1

    ! This exits the i construct
    if (i*j==15) exit do1

  end do
end do1
```

Schleifen: implied do

```
print *, (<item-1>, <item-2>, ...., <item-n>, do-variable = initial, final, step)
```

- schnelle Möglichkeit, bei Ein- oder Ausgabe viele Elemente aufzulisten
- Elemente können Variablen, einschließlich Array-Elemente, Ausdrücke oder sogar implizierte do-Schleifen sein

```
write(*,*) ( 'Hallo', i = 1, 5 )
! Ausgabe: HalloHalloHalloHalloHallo

print *, ( i, i*i, i = 1, 10, 3 )
! Ausgabe: 1,  1,  4,  16,  7,  49,  10,  100

print *, ( i, ( i*j, j = 1, 3), i = 1, 3 )
! Ausgabe: 1, 1, 2, 3, 2, 2, 4, 6, 3, 3, 6, 9
```

Schleifen: Spezialkonstrukte für Arrays

- *forall*: Arrayelemente über Indexbereiche manipulieren
- *where*: Arrayelementen in Abhängigkeit von vorgegebenen Bedingungen neue Werte zuweisen

```
integer, dimension(5)    :: a = (/ 1, 2, 3, 4, 5 /), b = (/ 5, 1, -1, 1, 7 /)
integer :: i,j

! forall-Einzeiler (für 1-dimensionale Felder)
! forall(index = start:end[:step]) anweisung
forall( i = 1:3 ) a(i) = 0
write( *, * ) a ! Ausgabe: 0  0  0  4  5

! where-Einzeiler
! where(<bedingung>) variable = ausdruck
where( b >= 3 ) b = 0
write(*,*) b ! Ausgabe: 0  1  -1  1  0
```

- *Mehr zu Arrays später*

hochschule mannheim

Übung: I/O, Verzweigungen & Schleifen - 30 min

Übung zu I/O, Verzweigungen, Schleifen

1) Schreibt ein Programm, das zunächst "Hello World!" auf der Konsole ausgibt und anschließend nur die ersten 5 Zeichen von "Hello World!".
2) Schreibt ein Programm, das die Variable x 25-mal und die Variable y 20-mal erhöht.
3) Schreibt ein Programm, das zunächst via Kommandozeile Name und Alter abfragt und anschließend den Input in einer Datei ablegt.
4) Schreibt ein Programm, dass zunächst eine Datei einliest und anschließend auf der Konsole die Anzahl der Zeilen in dieser Datei ausgibt.
5) Zusatz: Erweitert das Programm der vorherigen Übung (Heron & Pythagoras), sodass Ein- und Ausgabe benutzerfreundlicher sind (z.B. welche Formel soll auf Eingabe angewendet werden, genaue Fehlerangabe beispielsweise "c ist zu klein" o.ä.).

hochschule mannheim

Pause - 10 min

hochschule mannheim

Modularisierung

Modularisierung - Subroutinen

- Subroutinen sind Unterprogramme ohne expliziten Rückgabetyp (entspricht void-Methoden in C-ähnlichen Sprachen)

- Beginnen mit dem Schlüsselwort
  `subroutine` und enden mit
  `end subroutine`

- Können Parameter haben, die sowohl als Input als auch als Output fungieren können:
  `intent(in) :: a, b`
  `intent(out) :: c`

```
subroutine sayHi
  write(*, *) "Hi!"
end subroutine sayHi

call sayHi
>>> Hi!

subroutine add(a, b, c)
  implicit none
  real, intent(in) :: a, b
  real, intent(out) :: c
  c = a + b
end subroutine add

real :: sum
call add(3.0, 4.0, sum)
>>> sum = 7.0
```

Modularisierung - Subroutinen

- Subroutinen und Funktionen müssen im `contains`-Bereich des Programms oder Moduls stehen
- Verwendung im Hauptteil des Programms mit dem Schlüsselwort `call`
- Unterprogramme haben einen eigenen Scope, daher kann wieder `implicit none` verwendet werden

```
program Example

  call sayHi

contains

  subroutine sayHi
    implicit none
    write(*, *) "Hi!"
  end subroutine sayHi

end program

>>> Hi!
```

Modularisierung - Funktionen

- Sehr ähnlich zu Subroutinen, haben aber einen Rückgabewert
- Rückgabevariable kann implizit sein (Variable hat den Namen der Funktion)
- Rückgabevariable kann mit dem Schlüsselwort `result` explizit gemacht werden und wird dann im Funktionsrumpf separat deklariert
- Fortran arbeitet bei der Parameter-übergabe immer nach *Call by reference*

```
real function add(a, b)
  implicit none
  real, intent(in) :: a, b
  add = a + b
end function add

add(3.0, 4.0)
>>> 7.0

function add2(a, b) result(c)
  implicit none
  real, intent(in) :: a, b
  real :: c
  c = a + b
end function add2

add2(3.0, 4.0)
>>> 7.0
```

Modularisierung - Module

- Module werden normalerweise in einer eigenen Datei deklariert, in der gleichen Datei müssen sie vor dem Programm stehen
- Schlüsselwort `module` statt `program`
- Verwendung im Programm (Import) mit dem Schlüsselwort `use`
- Deklarierte Variablen und Funktionen sind public (können auch private gemacht werden)

```
! somelib.f90
module SomeLib
  implicit none
  real, parameter :: pi = 3.14159265
contains
  function add(a, b) result(c)
    ! ...
  end function add
end module
```

```
! program.f90
program ModuleExample
  use SomeLib
  implicit none
  real sum
  sum = add(pi, 1.0)
end program
```

Modularisierung – Kompilierung

- Option 1: Einzeln kompilieren: `gfortran -c somelib.f90 program.f90`
  - Erzeugt Dateien `somelib.mod`, `somelib.o`, `program.o`
  - Anschließend kann das Programm wie folgt gelinkt werden:

    `gfortran somelib.o program.o`

- Option 2: Zusammen kompilieren: `gfortran somelib.f90 program.f90`
  - **Wichtig**: Hier spielt die Reihenfolge eine Rolle – Dependencies müssen zuerst aufgezählt werden

hochschule mannheim

Arrays

Arrays – Deklaration

- Deklaration einer Variable als Array mit dem Schlüsselwort `dimension`

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

- Arrays fester Länge:

  `dimension(3)`

```
integer :: a = 5
real, dimension(a) :: vecB
```

- Arrays laufzeitabhängiger Länge:

  `dimension(a)`

- Arrays impliziter Länge, z. B. als Funktionsparameter:

  `dimension(:)`

```
subroutine foo(vecC)

  implicit none
  real, dimension(:), intent(in) :: vecC

end subroutine foo
```

Arrays – Indizierung

- Des erste Element hat standardmäßig den Index 1, nicht 0 wie in den meisten Sprachen üblich

- Die Indices können durch den Programmierer anders gesetzt werden, auch mit negativem Vorzeichen:

  `dimension(-1:1)`

- Zugriff auf einzelne Elemente oder Teilarrays:

  `vecB(-1)` oder `vecB(-1:0)`

```
real, dimension(2) :: vecA

>>> Index 1 bis 2

real, dimension(-1:1) :: vecB

>>> Index -1 bis 1

real :: a
a = vecB(-1)

real, dimension(2) :: vecC
vecC = vecB(-1:0)

>>> Index 1 bis 2 mit den ersten beiden
Werten aus vecB
```

Arrays – Zuweisung

- Konstante Zuweisung mit gleichem Wert:

```
vecA = 0
```

- Konstante Zuweisung mit beliebigen Werten:

```
vecB = (/ 0, 2 /)
```

- Indizierte Zuweisung:

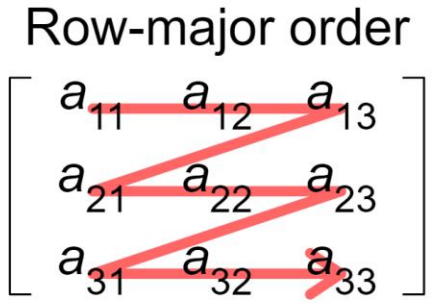
```
vecC(1:2) = vecB
```

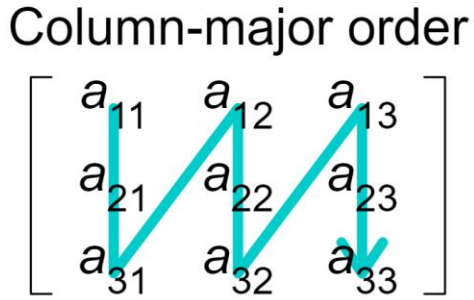
- Bei einer Zuweisung von Elementen aus einem Array zu einem anderen werden die Werte kopiert

```
real, dimension(2) :: vecA = 0

>>> (0.0, 0.0)

real, dimension(2) :: vecB = (/ 0, 2 /)

>>> (0.0, 2.0)

real, dimension(3) :: vecC = 1
vecC(1:2) = vecB

>>> (0.0, 2.0, 1.0)
```

Arrays – Mehrdimensional

- Fortran bietet native n-dimensionale Arrays (vgl. Arrays of Arrays in Java)
- Bis zu 15 Dimensionen möglich (F08)
- Arrays orientieren sich bei der Indizierung an mathematischen Konventionen für Vektoren und Matrizen
- Im Speicher hingegen liegen Arrays entsprechend der **Column-major order** vor





https://upload.wikimedia.org/wikipedia/commons/4/4d/Row_and_column_major_order.svg

```
real, dimension(2, 3) :: M
M(1, 1:3) = (/ 11, 12, 13 /) ! a11 a12 a13
M(2, 1:3) = (/ 21, 22, 23 /) ! a21 a22 a23

write(*, *) M
>>> 11.0   21.0   12.0   22.0   13.0   23.0
```

Arrays – Wichtige Funktionen

- Abfragen von Arraydimensionen und -größe mit `shape()`
- Zuweisen von n-dimensionalen Arrays mit `reshape()`
- Abfragen der Länge mit `size()`
- Abfragen von min. und max. Indices mit `lbound()` und `ubound()`

```
real, dimension(2,3) :: M

shape(M)
>>> (2, 3)

M = reshape((/ 11, 21, 12, 22, 13, 23 /), shape(M))


real, dimension(-2:2) :: vecD

size(vecD)
>>> 5

lbound(vecD)
>>> -2

ubound(vecD)
>>> 2
```

Arrays – Matrixoperationen

- Operatoren wie +, -, *, /, ** , sowie Funktionen wie `sqrt()`, `abs()`, etc funktionieren problemlos mit n-dimensionalen Arrays, solange die jeweilige Operation mathematisch zulässig ist
- Matrixmultiplikation mit `matmul()`
- Transponieren (Vertauschen von Zeilen und Spalten) einer Matrix mit `transpose()`

```
real, dimension(3) :: vec = (/ 1, 1, 1 /)
real, dimension(2, 3) :: M

M = reshape((/ 11, 21, 12, 22, 13, 23 /), shape(M))

write(*, *) transpose(M)
>>> 11.0    12.0    13.0    21.0    22.0    23.0

M = sqrt(M - 10)
write(*, *) M
>>> 1.0   3.31..   1.41..   3.46..   1.73..   3.60..

write(*, *) matmul(M, vec)
>>> 4.14...    10.38...
```

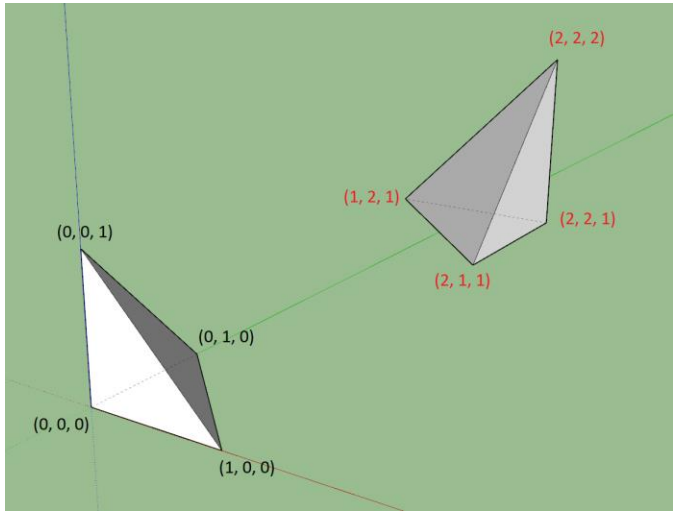
hochschule mannheim

Übung: Modularisierung und Arrays – 35 min

Übung zu Modularisierung und Arrays

- **Teil 1**: Implementiert ein **Modul** *"LinearAlgebra"* mit den Funktionen
  - **`makeIdentityMatrix`**`(int size) => real[,]` | Erzeugt eine Einheitsmatrix der Größe `size`
  - **`vecToHomogCoords`**`(real[] vector) => real[]` | Erzeugt ein Array der Länge n+1, wobei die ersten n Elemente kopiert werden und das Element n+1 gleich 1 ist
  - **`homogCoordsToVec`**`(real[] coords) => real[]` | Erzeugt ein Array der Länge n-1, wobei die Einträge kopiert und durch das n-te Element dividiert werden
- **Teil 2**: Implementiert mit Hilfe des Moduls ein **Programm**, das ein Objekt mit Punkten `(0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1)` wie folgt transformiert:
  - Verschiebung um `(2, 2, 1)`
  - Rotation (um die z-Achse) mit θ = 180° (θ = π)
- Benutzt dazu diese Transformationsmatrix:
- *Hinweis 1*: Mat(4, 4) * vec(4) = vec'(4)
- *Hinweis 2*: `sin()`, `cos()`, etc können ohne weiteres verwendet werden

$$\begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & x \\ \sin(\theta) & \cos(\theta) & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



hochschule mannheim

Objektorientierte Programmierung

Objektorientierte Programmierung - Einführung

- In Fortran gibt es eine etwas andere Umsetzung von OOP im Vergleich zu anderen bekannten Sprachen (C++, Java, Python, etc.)
- Aus zwei sprachlichen Features wird OOP "nachgebaut":
  - Derived Types – Datencontainer, die man beliebig häufig instanziieren kann. Ähnlich wie "Structs" in C

```
program x
    type shape
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
    end type shape

    type(shape) :: shp1, shp2

    shp1 = shape(color = 1, filled = .false., x = 2, y = 3)
    shp2 = shape(color = 2, filled = .false., x = 5, y = 6)
    print *, "shp1 color: ", shp1%color, "shp2 color: ", shp2%color
end program x
```

Definition des Derived Types

Deklaration der Variablen - Variablenname != DT-Name!

Instanziierung

Objektorientierte Programmierung - Einführung

- In Fortran gibt es eine etwas andere Umsetzung von OOP im Vergleich zu anderen bekannten Sprachen (C++, Java, Python, etc.)
- Aus zwei sprachlichen Features wird OOP "nachgebaut":
  - Derived Types – Datencontainer, die man beliebig häufig instanziieren kann. Ähnlich wie "Structs" in C
  - Modules – eine Art Packages, die aus einem Daten- und einem Methodenbereich bestehen

```
module mod_bsp
    ! Datenbereich
    real, private :: x = 1.2
    real          :: y = 9.8

    contains ! kündigt Methodenbereich an
    ! Methodenbereich
    real function addX (a)
        real, intent (in) :: a

        addX = x + a
    end function addX
end module mod_bsp
```

"*use mod_bsp*" im Programm ermöglicht Zugriff auf die Daten und Methoden

Objektorientierte Programmierung – Unsere erste Klasse

- Definition des Moduls
- Definition des Derived Types
  - Der Name des Derived Types ist auch der Name der Klasse!
  - Attribute werden angegeben
  - Methoden des Moduls, die zu Instanzmethoden werden "type-bound procedures". Compiler-Fehler, wenn eine Methode auf einem Objekt aufgerufen wird, das hier nicht angegeben ist
- Implementierung der Modulmethoden

```
module class_Circle
  implicit none
  private
  real :: pi = 3.1415926535897931d0 ! Class-wide private constant

  type, public :: Circle
     real :: radius
   contains
     procedure :: area => circle_area
     procedure :: print => circle_print
  end type Circle
contains
  function circle_area(this) result(area)
    class(Circle), intent(in) :: this
    real :: area
    area = pi * this%radius**2
  end function circle_area

  subroutine circle_print(this)
    class(Circle), intent(in) :: this
    real :: area
    area = this%area()  ! Call the type-bound function
    print *, 'Circle: r = ', this%radius, ' area = ', area
  end subroutine circle_print
end module class_Circle
```

Objektorientierte Programmierung – Erstellung unseres ersten Objekts

- Erstellung des Objekts *c* vom Typ *Circle*
- Zugriff auf Methoden und Attribute mit "%" statt mit ".", wie in vielen anderen Sprachen

```
program circle_test
  use class_Circle
  implicit none

  type(Circle) :: c     ! Declare a variable of type Circle.
  c = Circle(1.5)       ! Use the implicit constructor, radius = 1.5.
  call c%print          ! Call the type-bound subroutine
end program circle_test
```

- Ausgabe: "Circle: r = 1.5 area = 7.069"

Objektorientierte Programmierung – Type bound procedures (= TBP)

- Zuordnung zwischen Modul-Methoden zu dem Derived Type geschieht hier
- Das bedeutet: versucht man auf einem Objekt eine Methode im Modul aufzurufen, die hier nicht definiert ist, kommt es zum Compiler-Fehler
- Diese Zuordnung macht diese Methoden zu "Type bound procedures"
- Interessantes Feature, das nicht jede Sprache bietet: man ruft die Methode mit dem *binding-name* auf, der optional angegeben werden kann
  - Beispiel letzte Folie: es wird *print* aufgerufen, die eigentliche Methode heißt aber *circle_print*
  - Möchte man bspw. *circle_print* unter diesem Namen aufrufbar machen, schreibt man *procedure :: circle_print*
- Ermöglicht, wie wir sehen werden, das Überschreiben von Methoden
- Nachteil der ganzen Geschichte: Verwaltungsaufwand

```
  type, public :: Circle
     real :: radius
   contains
     procedure :: area => circle_area
     procedure :: print => circle_print
  end type Circle
contains
  function circle_area(this) result(area)
    class(Circle), intent(in) :: this
    real :: area
    area = pi * this%radius**2
  end function circle_area
```

Objektorientierte Programmierung – Instanzmethoden

- Werden eingeleitet durch ein *contains* nach der Definition des Derived Types
- Wie z. B. in Python: die Referenz auf das aufrufende Objekt *this* steht immer an erster Stelle in der Parameterliste
- Ein übergebenes Objekt muss einem Datentypen zugewiesen werden
  - Datentyp == Name des Derived Types

```
  type, public :: Circle
     real :: radius
   contains
     procedure :: area => circle_area
     procedure :: print => circle_print
  end type Circle
contains
  function circle_area(this) result(area)
    class(Circle), intent(in) :: this
    real :: area
    area = pi * this%radius**2
  end function circle_area
```

Objektorientierte Programmierung – Konstruktor - 1. Möglichkeit

- Das Objekt wird erst deklariert 1)
- Und anschließend initialisiert 2)
- Dieses Vorgehen funktioniert im Gegensatz zu anderen Sprachen, bei denen bei reiner Deklaration das Objekt *null* wäre
- Eine ähnliche Implementierung liefert in Java einen Compilerfehler:
  - *Exception in thread "main" java.lang.Error: Unresolved compilation problem: The local variable circle may not have been initialized*
- Ein richtiger Konstruktur ist das aber nicht – die Funktion initialisiert für uns das Objekt, anstatt ein fertiges Objekt zurückzugeben
- Nichtsdestotrotz ein valides Vorgehen in Fortran
- Möglicherweise inspiriert von den Sprachen Modula-2 und Oberon-2
  - Beide Sprachen haben ebenfalls keine Konstruktoren
  - SmallTalk ebenfalls nicht!

```
module workshop
    type shape
        integer :: x
        integer :: y
    contains
        procedure :: init => initShape
    end type shape

    type, extends(shape) :: circle
        real :: radius
        integer :: padding
    contains
        procedure :: init => initCircle
    end type circle

    contains
    subroutine initShape(this, x, y, radius, padding)
        class(shape) :: this
        integer :: x
        integer :: y
        integer :: padding
        real :: radius
        this%x = x
        this%y = y
    end subroutine initShape

    subroutine initCircle(this, x, y, radius, padding)
        class(circle) :: this
        real :: radius
        integer :: x
        integer :: y
        integer :: padding

        call this%shape%init(x, y, radius, padding)
        this%radius = radius
        this%padding = padding
    end subroutine initCircle
end module

program x
    use workshop
1)  type(circle) :: cir
2)  call cir%init(5, 5, 1., 2)

    print*, cir%x, cir%y, cir%radius, cir%padding
end program x
```

Objektorientierte Programmierung – Konstruktor - 2. Möglichkeit

- Factory-Pattern
- Bei Aufruf wird ein frisches Objekt zurückgegeben
- Ähnelt mehr dem Vorgehen von z. B. Java
- Nachteil:
  - Um Codedopplung zu vermeiden, werden intern trotzdem Init-Methoden benötigt, damit z. B. die Attribute von Shape in einer einzigen Methode festgelegt werden
  - Sonst Redundanz in constructor_shape() und constructor_circle()
    - Bei Erstellung eines Shape-Objekts: constructor_shape() => init_shape()
    - Bei Erstellung eines Circle-Objekts: constructor_circle() => init_circle() => init_shape()
- An sich also dasselbe wie die 1. Möglichkeit, allerdings mit zusätzlichen Factory-Methoden
  - Der dynamische Typ muss trotzdem anschließend per Hand festgelegt werden (dazu später)

Objektorientierte Programmierung – Vererbung

- Vererbung verhält sich wie in Java – Einfachvererbung
  - Konzepte und Ideen sind dieselben
- Vererbung geschieht bei den Derived Types mit dem Schlüsselwort *extends(<<superklasse>>)*
- Unterklasse erhält damit Zugriff auf Attribute und Methoden der Superklasse

```
program x
    type shape
        integer :: x
        integer :: y
    end type shape

    type, extends(shape) :: circle
        real :: radius
    end type circle

type(circle) :: cir
cir = circle(x=1, y=2, radius=3.5)
print *, cir%x, cir%y, cir%radius

end program x
```

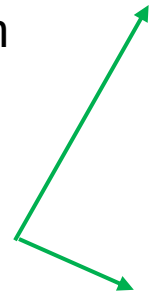
Objektorientierte Programmierung – Überschreiben von Methoden (1)

- Methoden werden überschrieben, indem sie denselben *binding-name* haben, aber letztlich verschiedene Methodennamen
  - In diesem Beispiel: beide Methoden implementieren *draw*
  - *binding-name* ist *draw,* d. h. die Objekte rufen die Methode unter diesem Namen auf
  - Wie man sieht, überschreibt der Derived Type *Circle* diese Methode
  - Je nach Klasse wird also *drawShape* oder *drawCircle* aufgerufen
  - Ausgabe bei diesem Code:
    drawShape
    drawCircle
  - Um explizit auf Attribute / Methoden der Superklasse zuzugreifen, kennt Java z. B. das Schlüsselwort *super*
    - in Fortran ist es *this%<<Name der Superklasse>>%<<Attr./Methode>>*

```
module workshop
    type shape
    contains
        procedure :: draw => drawShape
    end type shape

    type, extends(shape) :: circle
    contains
        procedure :: draw => drawCircle
    end type circle

    contains

    subroutine drawShape(this)
        class(shape) :: this
        print*, "drawShape"
    end subroutine drawShape

    subroutine drawCircle(this)
        class(circle) :: this
        print*, "drawCircle"
    end subroutine drawCircle
end module

program x
    use workshop
    type(shape) :: shp
    type(circle) :: cir
    call shp%draw()
    call cir%draw()
end program x
```

Objektorientierte Programmierung – Überschreiben von Methoden (2)

- Eigenheit von Fortran: überschriebene Methoden müssen **dieselbe** Parameterliste haben, sogar die Namen der Variablen müssen gleich sein
- Ansonsten Compilerfehler
- *initShape()* und *initCircle()* haben deswegen dieselbe Parameterliste, obwohl *initShape()* nur x und y benötigt
- Grund: es solle möglich sein, beide Methoden auf dieselbe Art und Weise aufzurufen
- TBP kamen in F2003 als Feature



```
module workshop
    type shape
        integer :: x
        integer :: y
    contains
        procedure :: init => initShape
    end type shape

    type, extends(shape) :: circle
        real :: radius
        integer :: padding
    contains
        procedure :: init => initCircle
    end type circle

    contains
    subroutine initShape(this, x, y, radius, padding)
        class(shape) :: this
        integer :: x
        integer :: y
        integer :: padding
        real :: radius
        this%x = x
        this%y = y
    end subroutine initShape

    subroutine initCircle(this, x, y, radius, padding)
        class(circle) :: this
        real :: radius
        integer :: x
        integer :: y
        integer :: padding

        call this%shape%init(x, y, radius, padding)
        this%radius = radius
        this%padding = padding
    end subroutine initCircle
end module

program x
    use workshop
    type(circle) :: cir
    call cir%init(5, 5, 1., 2)

    print*, cir%x, cir%y, cir%radius, cir%padding
end program x
```

Objektorientierte Programmierung – Polymorphismus (1)

- Es gibt mehrere und komplizierte Wege, um polymorphe Variablen zu erstellen
- Wir schauen uns eine Möglichkeit an, die z. B. den Konzepten von Java am ähnlichsten ist
- Vorherige Folien: Objekte mit *type(<<Name>>)* erstellt
  - Diese Objekte sind allerdings nicht polymorph!
  - Für richtiges OO-Verhalten benötigen wir *class(<<statischer Typ>>), allocatable :: Variablenname*
  - Der dynamische Typ wird mit *allocate()* festgelegt
    - Statischer Typ == Dynamischer Typ, dann reicht der Var.-Name
    - Ansonsten *allocate(<<dynamischer Typ>> :: Var.-Name)*
  - Ausgabe: variiert je nach Eingabe, Verhalten ist polymorph

```
module workshop
    type dackel
    contains
        procedure :: bellen => bellenDackel
    end type dackel

    type, extends(dackel) :: kampfdackel
    contains
        procedure :: bellen => bellenKampfdackel
    end type kampfdackel

    contains

    subroutine bellenDackel(this)
        class(dackel) :: this
        print *, "bellenDackel"
    end subroutine bellenDackel

    subroutine bellenKampfdackel(this)
        class(kampfdackel) :: this
        print *, "bellenKampfdackel"
    end subroutine bellenKampfdackel
end module

program jamoin
    use workshop
    integer :: i
    class(dackel), allocatable :: irgendeinHund
    read (*,*) i
    if(modulo(i,2) == 0) then
        allocate(irgendeinHund)
    else
        allocate(kampfdackel :: irgendeinHund)
    end if
    call irgendeinHund%bellen()
end program jamoin
```

Objektorientierte Programmierung – Polymorphismus (2)

- Mit *deallocate(<<Var-Name>>* wird der dynamische Typ entfernt
  - Notwendig, wenn man den dynamischen Typ ändern möchte
- Erinnert an Sprachen, bei denen Speicher manuell gemanaged wird
- Das class keyword kam ebenfalls mit F2003 hinzu
- Empfehlung: wenn objektorientiert programmiert wird, *type()* nicht benutzen, um Objekte zu erstellen, sondern *class(), allocatable*
  - Denn: *class()* bietet einen dynamischen Typ
  - Möglichkeit, ein Objekt ohne dynamischen Typen zu erstellen, kennt man aus modernen Sprachen nicht

```
module workshop
    type dackel
    contains
        procedure :: bellen => bellenDackel
    end type dackel

    type, extends(dackel) :: kampfdackel
    contains
        procedure :: bellen => bellenKampfdackel
    end type kampfdackel

    contains

    subroutine bellenDackel(this)
        class(dackel) :: this
        print *, "bellenDackel"
    end subroutine bellenDackel

    subroutine bellenKampfdackel(this)
        class(kampfdackel) :: this
        print *, "bellenKampfdackel"
    end subroutine bellenKampfdackel
end module

program jamoin
    use workshop
    integer :: i
    class(dackel), allocatable :: irgendeinHund
    read (*,*) i
    if(modulo(i,2) == 0) then
        allocate(irgendeinHund)
    else
        allocate(kampfdackel :: irgendeinHund)
    end if
    call irgendeinHund%bellen()
end program jamoin
```

Objektorientierte Programmierung – Information Hiding (1)

- Es gibt zwei Keywords: public und private
- Können ausschließlich in Modulen verwendet werden
  - 1: Alle Felder des Derived-Types sind private
  - 2: Alle TBPs des Typs sind private
  - 3: Man kann allerdings Ausnahmen anlegen, das geht ebenso für die Attribute
  - 4: Alle Methoden des Moduls sind private (außer *shape* und *constructor*). Es kann Methoden geben, die unabhängig von Objekten sind, diese werden hier eingestellt

```
module shape_mod

private 4 ! hide the type-bound procedure implementation procedures
public :: shape, constructor   ! allow access to shape & constructor procedure

type shape
    private      1       ! hide the underlying details
    integer :: color
    logical :: filled
    integer :: x
    integer :: y
contains
    private 2                          ! hide the type bound procedures by default
    procedure         :: initShape  ! private type-bound procedure
    procedure, public :: isFilled   ! allow access to isFilled type-bound procedure
    procedure, public :: print      ! allow access to print type-bound procedure
end type shape      3
```

Objektorientierte Programmierung – Information Hiding (2)

- Erinnert an C++, wo für viele Methoden oder Attribute gleichzeitig ein Zugriffsmodifizierer festgelegt wird
- Nichtsdestotrotz kann für jedes Attribut und jedes TBP einzeln ein Modifizierer festgelegt werden
- Im Unterschied zu Java: ein Objekt kann direkt auf ein privates Attribut der Oberklasse zugreifen

```
public:
   double length;
   void setWidth( double wid );
   double getWidth( void );

private:
   double width;
```

Objektorientierte Programmierung – Zusammenfassung

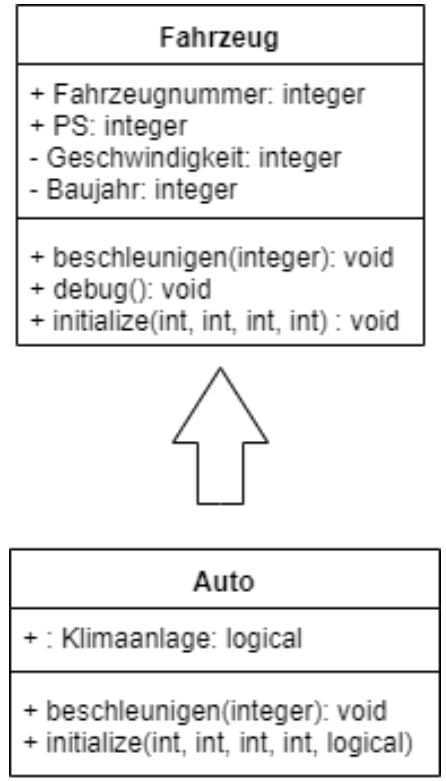
- Klassen werden zusammengebaut aus Derived Types, die in Modulen sitzen
- Es gibt keine echten Konstruktoren, stattdessen sollte man Initialisierungsmethoden nutzen
- Vererbungsmechanismen wie man es kennt. Einfachvererbung
- Überschreiben von Methoden, indem der *binding-name* derselbe ist, aber die ausgeführte Methode eine andere
- Information Hiding möglich mit public und private
- Es gibt mehrere Wege, ein Objekt zu erstellen
  - Empfehlenswert mit dem Schlüsselwort *class*, um einen dynamischen Typ zuweisen zu können

hochschule mannheim

Übung: Objektorientierte Programmierung – 20 min

Übung zu objektorientierter Programmierung

- Folgendes Klassendiagramm soll in Fortran-Code umgesetzt werden
- Zur Einfachheit: alles in einer Datei und beide Klassen in einem Modul
- Anmerkungen zu den Methoden:
  - Die beiden "beschleunigen"-Methoden sollen lediglich "Beschleunigen Fahrzeug" bzw. "Beschleunigen Auto" ausgeben
  - Die "debug"-Methode soll die vier Instanzvariablen von Fahrzeug ausgeben
    - Diese können bei der Initialisierung frei gewählt werden, entsprechend den Datentypen
  - Die "initialize"-Methode in Auto soll als erste Anweisung die "initialize"-Methode von Fahrzeug aufrufen



hochschule mannheim

Pause - 40 min

hochschule mannheim

Parallelisierung in Fortran

Parallelisierung

MPI

- Erster Standard in 1994
- Nicht Teil der Sprache
- Separate Runtime für Fortran benötigt
- Integriert über Bibliotheken
- Optimiert für Cluster

CoArray Fortran

- Teil des Fortran 2008 Standard
- Gleiches Programm mehrfach ausführen
  - => Images
- Spezielle Arrays werden über diese geteilt
- gfortran benötigt externe Bibliothek

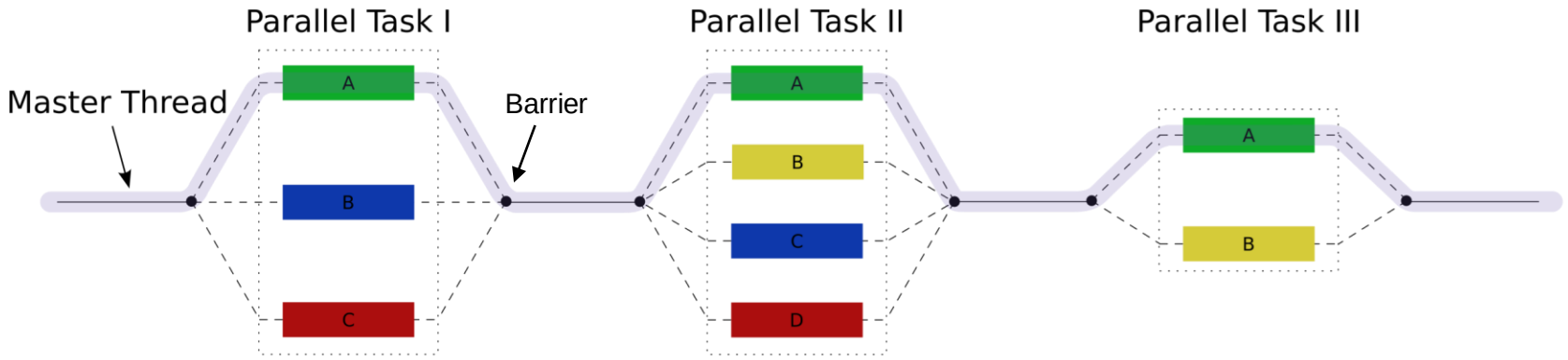
OpenMP

- Erster Release 1997
- Auch für C/C++
- Funktioniert über Compiler Direktiven und Bibliotheken
- Integriert in den gfortran Compiler

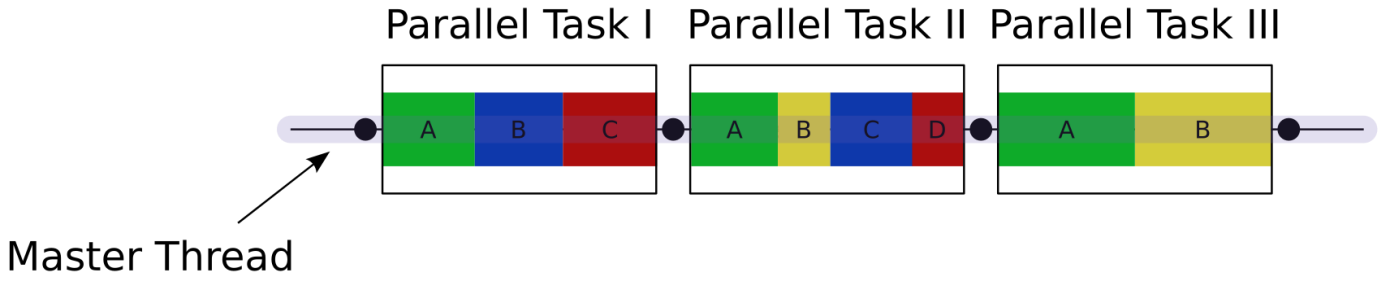
+ weitere Frameworks

Einführung OpenMP

"Fork-Join Modell"



→ Ermöglicht auch eine sequentielle Ausführung



Einführung OpenMP – Parallel DO

Verwendung der OMP-Bibliothek →

Auslesen und Drucken der max. Threads →

Markieren des “do” Blocks als Parallelisierung
→ Verwendung von Kommentaren →

```
program omp_par_do
  use omp_lib

  implicit none
  integer, parameter :: n = 10
  integer :: i
  integer :: nthreads

  nthreads = omp_get_max_threads()
  write(*, *) ' Number of threads = ', nthreads

  !$OMP PARALLEL DO
  do i = 1, n
    write(*, *) i
  end do
  !$OMP END PARALLEL DO

end program omp_par_do
```

Einführung OpenMP – Parallel DO

Kompilieren des Codes mit der '-fopenmp' Option:

```
gfortran parallel_do.f90 -o parallel_do -fopenmp
```

Ausführung:

```
~/PrivateProjects/Fortran » ./parallel_do
  Number of threads =          12
           4
          10
           7
           5
           3
           9
           1
           8
           2
           6
```

Einführung OpenMP – Verwendung von Variablen

```
program omp_par_do
  implicit none

  integer, parameter :: n = 10
  integer :: i
  integer :: x

  !$OMP PARALLEL DO
  do i = 1, n
    x = i
    write(*, *) x
  end do
  !$OMP END PARALLEL DO

end program omp_par_do
```

```
~/PrivateProjects/Fortran » ./parallel_do_wrong
          10
           1
           1
           1
           1
           1
           1
           1
           1
           1
           1
```

’x’ ist keine DO-Variable & nicht als thread-privat deklariert
→ Implizit eine shared Variable

Einführung OpenMP – Verwendung von Variablen

```
program omp_par_do
  implicit none

  integer, parameter :: n = 10
  integer :: i
  integer :: x

  !$OMP PARALLEL DO PRIVATE(x)
  do i = 1, n
    x = i
    write(*, *) x
  end do
  !$OMP END PARALLEL DO

end program omp_par_do
```

```
~/PrivateProjects/Fortran » ./parallel_do_fixed
          10
           1
           4
           5
           8
           7
           9
           2
           3
           6
```

Einführung OpenMP – Parallel Block

Beginn Parallel Block &
Explizites Deklarieren der Variablen

Parallele DO Schleife innerhalb des Blocks
→ Aufteilen der Arbeit auf Threads

Kritische Sektion
→ Nur eine Ausführung gleichzeitig

Ende Parallel Block &
Ausgabe des Ergebnis

```
3   integer :: i, partial_Sum, total_Sum
4   total_Sum = 0;
5
6   !$OMP PARALLEL PRIVATE(partial_Sum) SHARED(total_Sum)
7       partial_Sum = 0;
8
9       !$OMP DO
10      DO i=1,1000
11          partial_Sum = partial_Sum + i
12      END DO
13      !$OMP END DO
14
15      !$OMP CRITICAL
16          total_Sum = total_Sum + partial_Sum
17      !$OMP END CRITICAL
18
19  !$OMP END PARALLEL
20  PRINT *, 'Total Sum: ', total_Sum
```

Einführung OpenMP – Parallel Block

Kompilieren des Codes mit der '-fopenmp' Option:

```
gfortran parallel_factorial.f90 -o parallel_factorial -fopenmp
```

Ausführung:

```
~/PrivateProjects/Fortran » ./parallel_factorial

 Add        58930  to           0
 Add        10626  to       58930
 Add        24738  to       69556
 Add        17682  to       94294
 Add        38263  to      111976
 Add        52041  to      150239
 Add        65819  to      202280
 Add        45152  to      268099
 Add         3570  to      313251
 Add        31374  to      316821
 Add        72708  to      348195
 Add        79597  to      420903
 Total Sum:       500500
```

Im Vergleich **ohne** OpenMP:

```
~/PrivateProjects/Fortran » ./parallel_factorial
 Add       500500  to           0
 Total Sum:       500500
```

Einführung OpenMP – Weitere Parallele Konstrukte

Ausführung eines Codeblocks nur auf einem Thread:

```
!$omp single
...
!$omp end single
```

Ausführung eines Codeblocks nur auf dem Master Thread:

```
!$omp master
...
!$omp end master
```

Atomic Read/Write/Update einer Variable:

```
!$omp atomic [read/write/update]
...
```

Synchronisierung aller Threads:

```
!$omp barrier
```

+ vieles mehr: https://www.openmp.org//wp-content/uploads/OpenMP3.1-FortranCard.pdf

hochschule mannheim

Übung: Parallelisierung – 25 min

Übung zu Parallelisierung

Aufgabe: Parallelisiert zwei Matrizen multiplizieren und die Spur der Ergebnismatrix berechnen

```java
for(int i = 0; i < 3; i++) {
  for(int j = 0; j < 3; j++) {
    c[i][j] = 0;
    for(int k = 0; k < 3; k++) {
      c[i][j] += a[i][k] * b[k][j];
    } //end of k loop
  } //end of j loop
} //end of i loop
```

Matrix-Multiplikation in Java

$$tr(A) = \sum_{i=1}^{n} a_{ii}$$

Berechnung der Spur

```fortran
integer, dimension(3,3) :: a
a = reshape([1, 2, 3, 1, 2, 3, 1, 2, 3], shape(a))
write( * , "(*(g0))" ) ( (a(i,j), " ", j=1, n), new_line("A"), i=1, n )
```

Erstellen & Ausgeben einer Matrix in Fortran

hochschule mannheim

Fortran im High-Performance-Computing (HPC)

Performance - Gründe

Parallelität:

- Bietet Parallelität mit OpenMP wie C und C++
- Coarray - dafür gibt es in C++ und Python externe Bibliotheken
  - Ab Fortran 2008 im Standard, also der Sprache selbst, integriert

Andere Gründe:

- Compiler ist sehr optimiert
- Array-Operationen vergleichbar mit MatLab und Numpy, aber teils schneller (s. Performanz-Test)
- Aliasing ist nicht erlaubt => Compiler kann wesentlich besser optimieren

Performance – Aliasing Beispiel

- Rechts ist ein C Code gegeben
- Aliasing: > 1 Pointer zeigen auf dieselbe Speicheradresse
- Das könnte hier der Fall sein: "output" und "matrix" könnten auf dieselbe Adresse zeigen
- Somit würde bei jeder Iteration auch der Wert von "matrix" geändert werden

```
void transform (float *output, float const * input, float const * matrix, int *n)
{
    int i;
    for (i=0; i<*n; i++)
    {
        float x = input[i*2+0];
        float y = input[i*2+1];
        output[i*2+0] = matrix[0] * x + matrix[1] * y;
        output[i*2+1] = matrix[2] * x + matrix[3] * y;
    }
}
```

- Das Programm ist also gezwungen, bei jeder Iteration die Werte neu aus dem RAM zu lesen
  - In Fortran hingegen könnte der Compiler "matrix" direkt in Register schieben
- I. A. ist Compiler-Optimierung ohne Aliasing besser möglich, auch was Umordnung von Befehlen durch den Compiler betrifft

Benchmarking-Seiten

- Es gibt Benchmark-Seiten, die Sprachen miteinander vergleichen
- Benutzer liefern Algorithmen und Ergebnisse
  - Als etwas kritisch im Vergleich zu Fortran zu sehen, da es mehr "Hardcore-Programmer" im C/C++ Bereich gibt als in Fortran
  - Bsp.: C++ ist ca. 4x schneller bei Berechnung der Mandelbrotmenge
  - C++ wurde allerdings manuell vektorisiert
    - d. h., eine viermalige Addition geschieht in einem Schritt statt in vier Schritten
- Somit solche Seiten manchmal mit Vorsicht zu genießen

$$
\begin{aligned}
c_1 &= a_1 + b_1 \\
c_2 &= a_2 + b_2 \\
c_3 &= a_3 + b_3 \\
c_4 &= a_4 + b_4
\end{aligned}
$$

```
for (i=0; i<n; i++)
    c[i] = a[i] + b[i];
```

Benchmarking-Seiten

- Ergebnisse zur Mandelbrotmenge
- Mehrere Lösungen zu einer Sprache + kompetitiv
- Zwar gut für einen ungefähren Vergleich, aber nicht wirklich wissenschaftlich

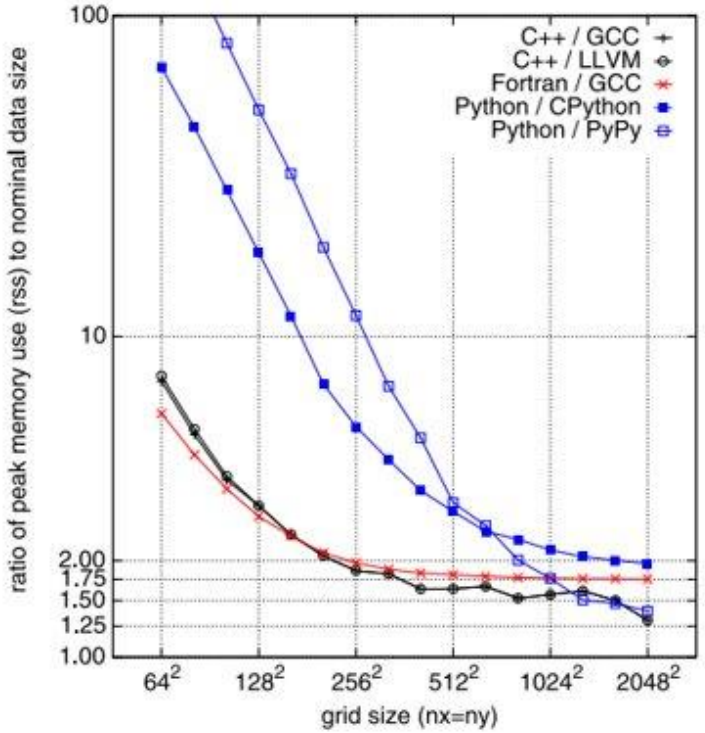
×	source	secs	mem	gz	busy	cpu load
1.0	**C++** g++ #4	**0.84**	34,604	3542	3.28	98% 99% 98% 95%
1.1	C++ g++	0.90	31,724	1791	3.51	98% 100% 98% 97%
1.1	**Rust** #8	**0.93**	32,788	763	3.69	100% 100% 99% 100%
1.1	Rust #7	0.93	32,644	757	3.71	100% 100% 100% 100%
1.1	C++ g++ #6	0.97	32,236	1002	3.84	99% 100% 99% 99%
1.2	Rust #3	0.99	12,920	1007	3.94	99% 100% 100% 100%
1.4	Rust #5	1.16	33,608	719	4.56	98% 98% 98% 99%
1.5	**C** gcc #6	**1.27**	31,692	1135	5.08	100% 100% 100% 99%
1.6	Rust #6	1.33	33,768	1332	5.28	100% 98% 98% 99%
1.6	**Julia** #8	**1.38**	235,276	621	4.78	83% 83% 83% 99%
1.7	Intel **Fortran** #8	**1.42**	36,184	957	5.63	99% 100% 99% 99%
1.7	**Haskell** GHC #3	**1.47**	38,268	1975	5.82	99% 100% 99% 99%
2.2	Julia #7	1.84	226,908	619	6.70	98% 89% 88% 89%
2.2	**Swift** #3	**1.87**	39,072	1138	7.44	100% 99% 98% 99%
2.3	Julia #5	1.93	267,708	571	7.06	98% 90% 89% 89%
2.4	Julia #3	2.05	267,188	579	7.56	100% 90% 89% 89%
2.8	Julia #6	2.40	240,848	562	6.91	59% 90% 90% 49%
3.1	Julia #4	2.63	256,060	574	7.46	90% 47% 57% 90%
3.3	Rust	2.78	38,744	868	10.92	98% 98% 99% 98%
3.7	**C#** .NET	**3.14**	64,716	1974	12.28	99% 97% 97% 97%
3.7	C# .NET #9	3.15	64,520	816	12.29	97% 97% 97% 99%
3.7	Intel Fortran #6	3.16	75,852	661	10.64	79% 79% 79% 100%

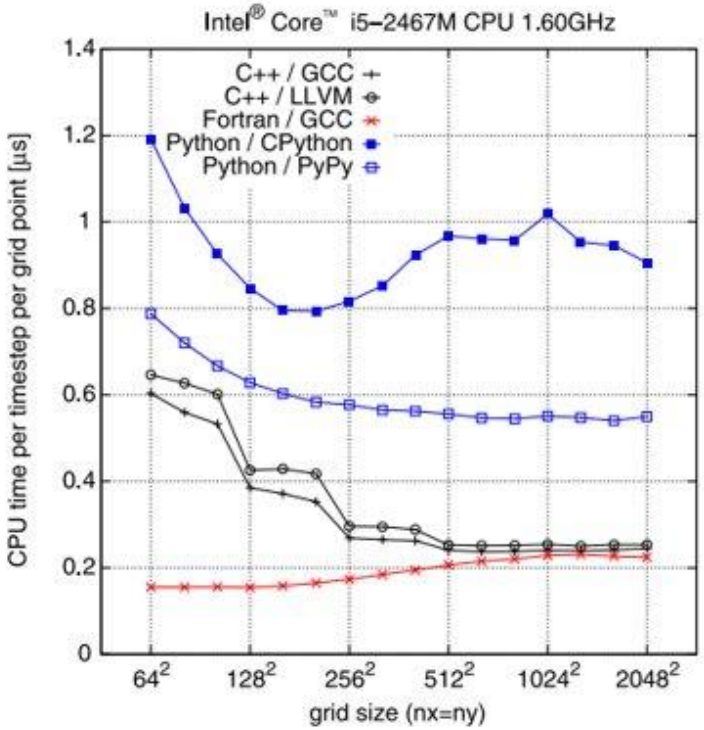
Performanz-Test

- Paper "Formula translation in Blitz++, NumPy and modern Fortran: A case study of the language choice tradeoffs"
- Vier Mitarbeiter der Universität von Warschau in Polen, Fak. für Physik
- Haben "MPDATA" implementiert, ein Algorithmus, der für Wetter-, Klima- und Ozeansimulationen verwendet wird
  - Berechnungen sehr Array-lastig
- Haben Fortran verglichen mit zwei C++ Compiler-Implementierungen und zwei Python-Implementierungen
- Python: NumPy, C++: Blitz++, Fortran: eigene Array-Sprachelemente
  - Blitz++: hoch-performante Arraybibliothek für C++
  - NumPy: Python-Bibliothek für Berechnungen mit Arrays

Performanz-Test

- Links: Speichernutzung
  - Bei kleinen Eingaben ungefähr gleichauf mit C++; gegenüber PY Vielfaches an weniger Verbrauch
  - Bei größeren Eingaben ab 1024^2 im Mittelfeld zwischen C++ / PyPy und CPython
- Rechts: CPU Zeit
  - Fortran bei jeder Größe 1. Platz





Fortran im HPC

- Fortran sei simpler zu erlernen als C, vorteilhafter für Nicht-Informatiker
- Viele Projekte in Domänen oder Bibliotheken in Fortran geschrieben
  - Es braucht sehr guten Grund, um als Nicht-Informatiker eine neue Sprache und ihr Umfeld zu erlernen
  - Wissenschaftler widmen sich lieber ihrem eigenen Themengebiet

hochschule mannheim

Diskussionsrunde
Fortran ←→ Moderne Sprachen

- Vorteile von Fortran, die es in modernen Sprachen nicht mehr gibt?
- War die Syntax von Fortran besser/schlechter?
- Was wurde vielleicht von Fortran in modern Sprachen übernommen?
- Gibt es Gründe heutzutage noch Fortran zu nutzen?
- …

hochschule mannheim

Danke für eure Aufmerksamkeit!

Wir würden uns sehr über euer Feedback freuen:
https://forms.gle/taRZ6LVKbJnGrdss7

Quellen

- https://stackoverflow.com/questions/146159/is-fortran-easier-to-optimize-than-c-for-heavy-calculations/146186#146186
- https://en.wikipedia.org/wiki/Automatic_vectorization#Manual_vectorization
- https://riptutorial.com/fortran/example/6897/arithmetic-if-statement
- https://riptutorial.com/Download/fortran.pdf
- https://pages.mtu.edu/~shene/COURSES/cs201/NOTES/chap08/io.html
- https://de.wikibooks.org/wiki/Fortran:_Fortran_95:_Verzweigungen_und_Schleifen#forall
- https://de.wikibooks.org/wiki/Fortran:_Fortran_95
- https://de.wikibooks.org/wiki/Fortran:_Fortran_77
- https://www.whoishostingthis.com/resources/fortran/
- http://www.ece.utep.edu/courses/web3376/Notes_files/ee3376-assembly.pdf
- https://www.tutorialspoint.com/fortran/fortran_data_types.htm
- https://www.tutorialspoint.com/fortran/fortran_variables.htm
- https://ourcodingclub.github.io/tutorials/fortran-intro/
- https://en.wikipedia.org/wiki/Fork%E2%80%93join_model#/media/File:Fork_join.svg
- https://en.wikipedia.org/wiki/Punched_card
- https://craftofcoding.wordpress.com/2017/01/28/read-your-own-punch-cards/
- https://de.wikibooks.org/wiki/Fortran:_OpenMP
- http://fortranwiki.org/fortran/show/OpenMP
- https://curc.readthedocs.io/en/latest/programming/OpenMP-Fortran.html
- https://www.openmp.org//wp-content/uploads/OpenMP3.1-FortranCard.pdf
- Introduction to Programming with Fortran, Ian Chivers und Jane Sleightholme, Springer 2018