Wissenschaftliches Programmieren

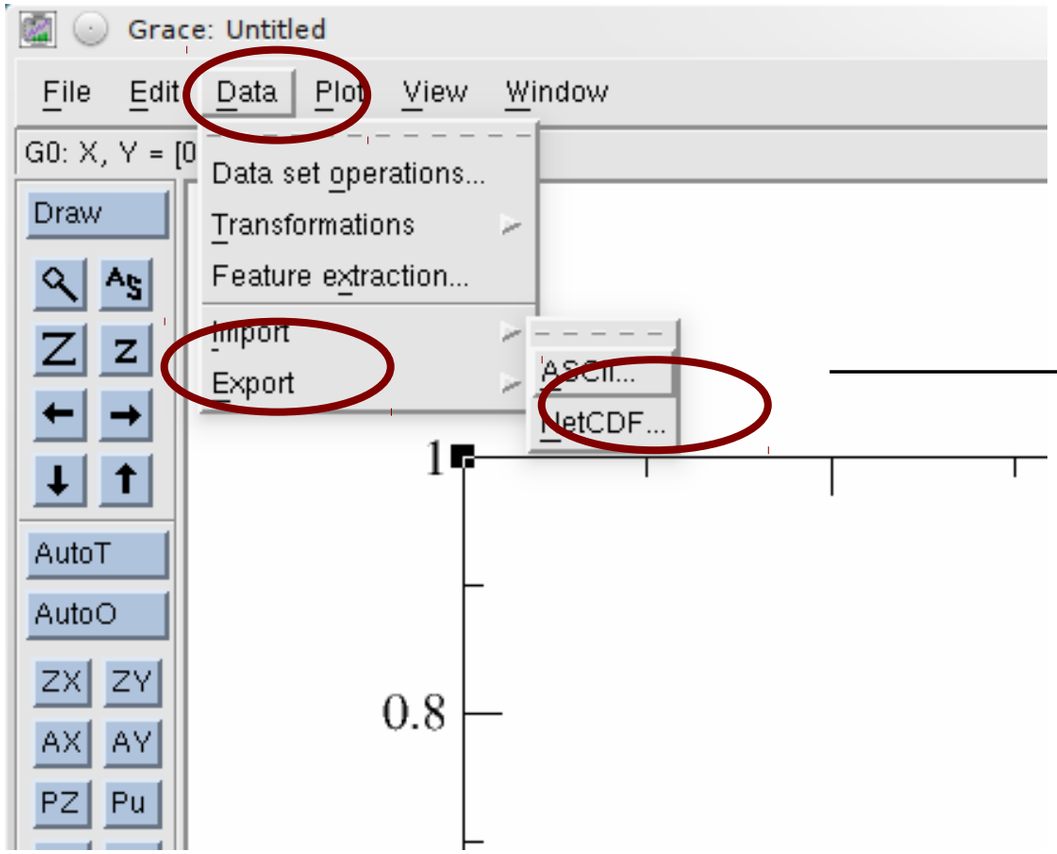
Bálint Aradi
http://www.bccms.uni-bremen.de/cms/people/b-aradi

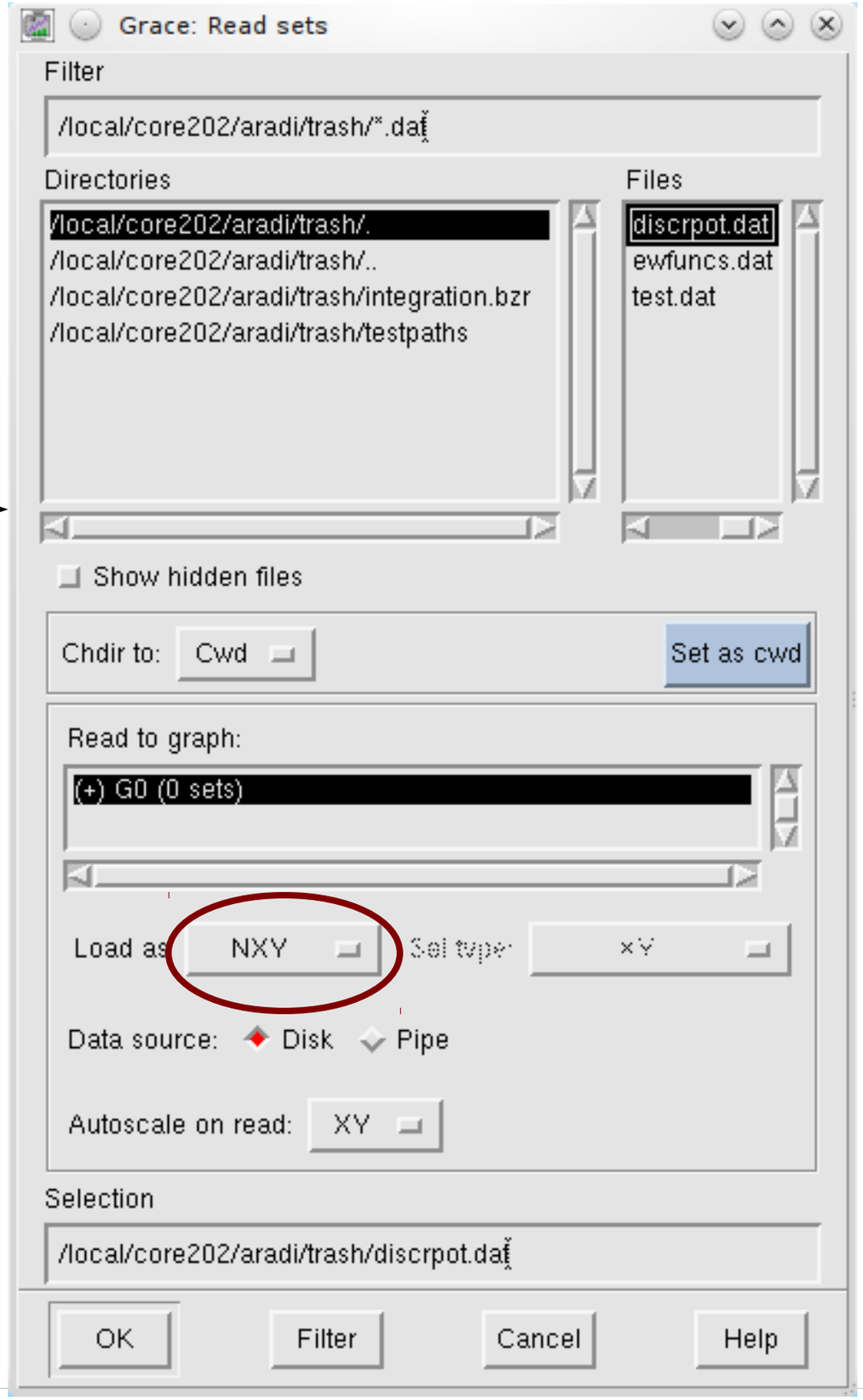
13. Benutzerdefinierte Datentypen (derived types)

Visualisierung von XY und NXY-Daten mit Grace

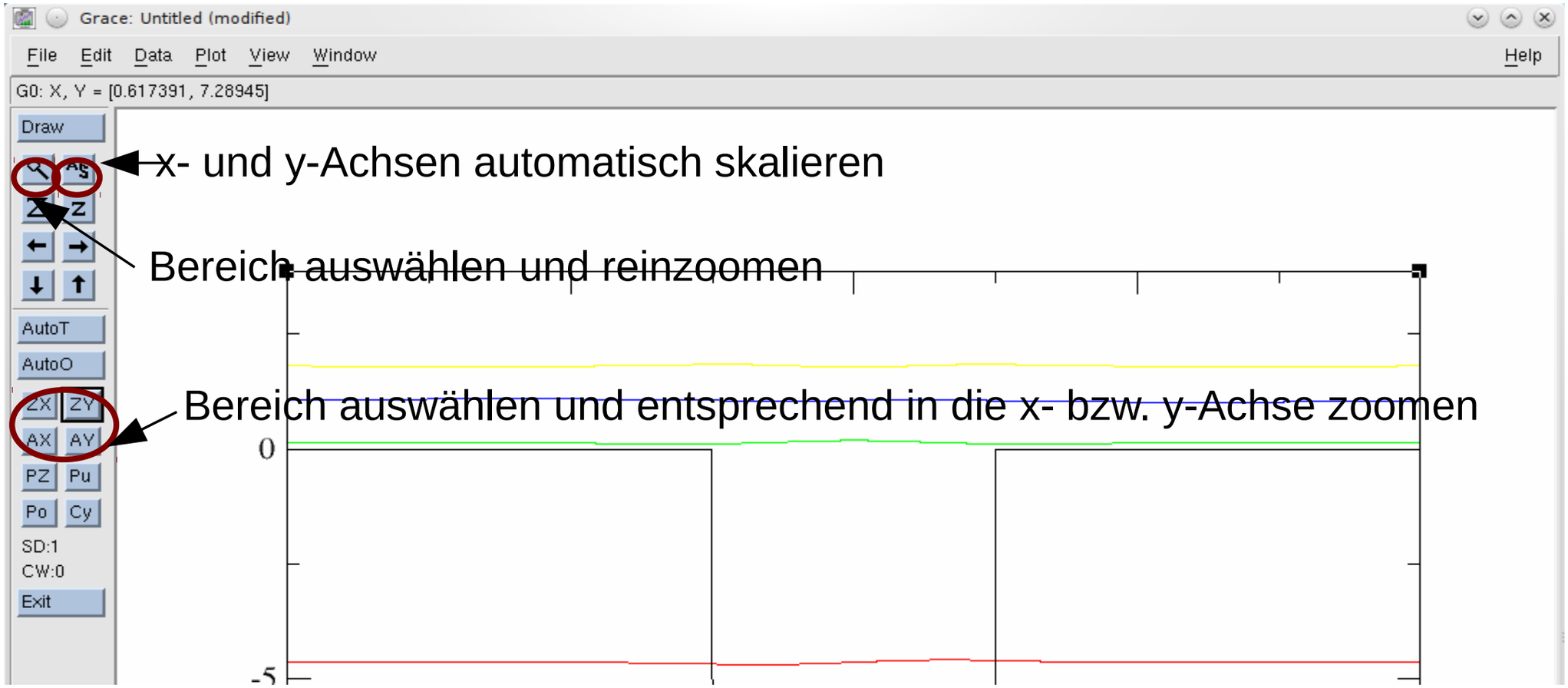
```
user:~> xmgrace
```

Datei laden:





Visualisierung von XY und NXY-Daten mit Grace



Weitere interessante Menüpunkte:

Plot | Axis properties

Plot | Set properties

Benuterzhandbuch:

http://plasma-gate.weizmann.ac.il/Grace/doc/UsersGuide.html

Benutzerdefinierte Typen

Variablentypen:

Intrinsiche Typen: Wird von der Programmiersprache (vom Compiler) bereitgestellt (integer, real, complex, logical, character)

Benutzerdefinierte Typen: Wird im Programm (oder Modul) mit Hilfe von intrinsischen Typen (und eventuell schon eingeführten benutzerdefinierten Typen) definiert.

- Mit benutzerdefinierte Typen können logisch zusammengehörende Variablen zu einer Einheit zusammgefasst werden

  Beispiel:

  Rationale Zahlen, Nenner und Zähler durch jeweils ein Integer repräsentiert

```
integer :: num, denom
```

Arithmetische Operationen mit rationalen Zahlen

```
integer :: num1, denom1, num2, denom2, numres, denomres
:
call add(num1, denom1, num2, denom2, numres, denomres)
```

Viel besser wäre:

```
qres = add(q1, q2)
```

bzw.

```
qres = q1 + q2
```

Benutzerdefinierte Typen (Beispiel)

```
module rationals
  implicit none
  private

  public :: rational, rational_add

  type rational
    integer :: num
    integer :: denom
  end type rational

contains

  function rational_add(q1, q2) result(res)
    type(rational), intent(in) :: q1, q2
    type(rational) :: res

    res%num = q1%num * q2%denom &
        + q2%num * q1%denom
    res%denom = q1%denom * q2%denom

  end function rational_add
end module rationals
```

```
program TestRationals
  use rationals
  implicit none

  type(rational) :: q1, q2, q3

   q1 = rational(1, 2)
   q2 = rational(3, 4)
   q3 = rational_add(q1, q2)
   write (*, "(2I10)") q3

end program TestRationals
```

Deklaration von Benutzerdefinierten Typen

- Benutzerdefinierte Typen werden mit der Type-Deklarationsanweisung deklariert

```
type rational
  integer :: num
  integer :: denom
end type rational
```

← Name des neuen Typs

Enthaltene Komponenten (intrinsiche Datentypen)

- Variablendeklaration vom benutzerdefinierten Typ:

```
type(rational) :: q1, q2, q3
```

- Analog zu intrinsischen Typen können für Variablen mit benutzerdefiniertem Typ Arrays definiert werden:

```
type(rational) :: array(12)
type(rational), allocatable :: array2(:,:)
:
allocate(array(10, 10))
array(3,3) = rational(3,2)
```

- Konstanten mit benutzerdefiniertem Typ:

```
type(rational), parameter :: onehalf = rational(1, 2)
```

Operationen mit benutzerdefinierten Typen

- Die Komponenten einer Variable mit benutzerdefiniertem Typ können via „%“-Operator erreicht werden (und als normale Variablen verwendet werden)

```
type(Rational) :: q1, q2, q3

# q1 = 1/2
q1%num = 1
q1%denom = 2
# q2 = 3/4
q2%num = 3
q2%denom = 4
# a/b + c/d = (a*d + c*b) / (b*d)
q3%num = q1%num * q2%denom + q2%num * q1%denom
q3%denom = q1%denom * q2%denom
```

Komponenten von intrinsischem Typ können wie normale intrinsiche Variablen verwendet werden

- Wertzuweisung kann auch für alle Komponenten gleichzeitig erfolgen:

```
type(rational) :: q1, q2

q1 = rational(1, 2)
q2 = rational(3, 4)
```

Name des Typs

Wert der einzelnen Komponenten, in der selben Reihenfolge, wie sie in der Typdeklaration vorkommen.

Wertzuweisung

- Benutzerdefinierte Variablen werden bei Wertzuweisung komponentenweise gleichgesetzt:

```
type(rational) :: q1, q2

q1 = rational(1, 2)
q2 = q1
```

Äquiv.

```
q2%num = q1%num
q2%denom = q1%denom
```

- Benutzerdefinierte Typen können weitere benutzerdefinierte Typen enthalten, aber nicht sich selbst (nur Zeiger auf sich):

```
type point
  integer :: row, col
end type point

type pixel
  type(point) :: coords
  integer :: color
end type pixel
```

```
type(pixel) :: px

px%coords%row = 3
px%coords%col = 2
px%color = 5

px%coords = point(3, 2)
px%color = 5

px = pixel(point(3, 2), 5)
```

Äquivalente Wertzu-weisungen

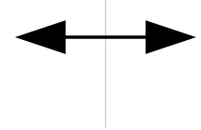
Input/Output von benutzerdefinierten Typen

- Eingabe und Ausgabe von benutzerdefinierten Typen passiert komponentenweise als stünden die einzelnen Komponenten hintereinander in der IO-Liste

```
type(rational) :: q3

write(*,*) "Enter numerator and&
    & denominator:"
read(*, *) q3
write(*,*) "You entered the following&
    & numerator and denominator:"
write (*, "(2I10)") q3
```

```
read(*, *) q3%num, q3%denom
```



```
write (*, "(2I10)")&
    & q3%num,
q3%denom
```

- In Fortran 2003 läßt sich für benutzerdefinierte Typen beliebige I/O-Formattierung definieren. (Nur in sehr wenigen Compilern implementiert.)

Benutzerdefinierte Typen (Beispiel)

```
module rationals
  implicit none
  private

  public :: rational, rational_add

  type rational
    integer :: num
    integer :: denom
  end type rational

contains

  function rational_add(q1, q2) result(res)
    type(rational), intent(in) :: q1, q2
    type(rational) :: res

    res%num = q1%num * q2%denom &
        + q2%num * q1%denom
    res%denom = q1%denom * q2%denom

  end function rational_add
end module rationals
```

```
program TestRationals
  use rationals
  implicit none

  type(rational) :: q1, q2, q3

   q1 = rational(1, 2)
   q2 = rational(3, 4)
   q3 = rational_add(q1, q2)
   write (*, "(2I10)") q3

end program TestRationals
```

Überladen von Operatoren (Beispiel)

```
module rationals
  implicit none
  private

  public :: rational, operator(+)

  type rational
    integer :: num
    integer :: denom
  end type rational

  interface operator(+)
    module procedure rational_add
  end interface operator(+)

contains

  function rational_add(q1, q2) result(res)
    type(rational), intent(in) :: q1, q2
    type(rational) :: res
    :
  end function rational_add
end module rationals
```

```
program TestRationals
  use rationals
  implicit none

  type(rational) :: q1, q2, q3

   q1 = rational(1, 2)
   q2 = rational(3, 4)
   q3 = q1 + q2
   write (*, "(2I10)") q3

end program TestRationals
```

Überladen von Operatoren

- Die Sprache definiert Funktionen und Operatoren nur für intrinsische Typen:

```
integer :: i1, i2, i3
type(rational) :: q1, q2, q3
:
i1 = i2 + i3
q1 = q2 + q3
```

OK!

Problem! „This binary operation is invalid for this data type.“

- Mit Hilfe von **Interface-Blocks** kann die Bedeutung von bereits definierten Funktionen, Subroutinen und Operatoren erweitert werden.

- Diese Interface-Blocks müssen sich innerhalb von Modulen befinden:

```
module rationals
  :
  interface operator(+)
    module procedure rational_add
  end interface operator(+)

contains
  function rational_add(q1, q2)
  :
```

Der Operator + wird mit der Prozedur Rational_add (im selben Modul) erweitert

Operatorerweiterung

- Bei der Auswertung einer Operation mit benutzerdefinierten Typen wird nach einer Erweiterung mit den entsprechenden Typen gesucht:

```
program rational_test
  use Rationals
  implicit none

  type(rational) :: aa, bb, cc
  :
  cc = aa + bb
```

Braucht eine Funktion mit:

- 2 type(rational) Argumente (intent(in))
- type(rational) Rückkehrwert

```
module rationals
  :
  interface operator(+)
    module procedure rational_add
  end interface operator(+)

contains
  function rational_add(q1, q2) result(res)
     type(rational), intent(in) :: q1, q2
     type(rational) :: res
  :
```

Funktion mit aa und bb als Parameter ausgeführt, Ergebnis in cc gespeichert

Typübereinstimmung

- Funktionen für Erweiterung von binären Operatoren müssen zwei intent(in) Parameter haben. (z.B.: Addition verändert die Werte der Summanden nicht)
- Sie können Parameter mit unterschiedlichem Typ besitzen, wenn die entsprechende Operation zwischen diesen Datentypen definiert werden soll:

```
type(rational) :: q1, q2
integer :: ii
:
q2 = q1 + ii
```

```
module rationals
  :
  interface operator(+)
    module procedure rational_addint
  end interface operator(+)

contains
  function rational_addint(q1, i2) result(res)
    type(rational), intent(in) :: q1
    integer, intent(in) :: i2
    type(rational) :: res
  :
```

Eindeutigkeit

- Ein Operator kann um beliebig viele Funktionen erweitert werden, *solange die Eindeutigkeit gewährleistet ist*:

```
interface operator(+)
  module procedure rational_add
  module procedure rational_addint1, rational_addint2
end interface operator(+)

contains
  function rational_add(q1, q2) result(res)
     type(rational), intent(in) :: q1, q2
     type(rational) :: res
     :
function rational_addint1(q1, i2) result(res)
     type(rational), intent(in) :: q1
     integer, intent(in) :: i2
     type(rational) :: res
:
function rational_addint2(q1, i2) result(res)
     type(rational), intent(in) :: q1
     integer, intent(in) :: i2
     type(rational) :: res
:
```

Falsch!
Nicht eindeutig, welche Funktion bei
q2 = q1 + 1
aufgerufen werden soll!

Kommutativität

- Wenn Kommutativität erwünscht ist, muss für Operationen mit gemischten Typen für beide Typreihenfolgen eine Erweiterungsfunktion implementiert werden.

```
interface operator(+)
  module procedure rational_add
  module procedure rational_addint1, rational_addint2
end interface operator(+)

contains
  :
  function rational_addint1(q1, ii) result(res)
     type(rational), intent(in) :: q1
     integer, intent(in) :: ii
     type(rational) :: res
  :
  function rational_addint2(ii, qq) result(res)
     integer, intent(in) :: ii
     type(rational), intent(in) :: qq
     type(rational) :: res
  :
```

```
type(Rational) :: q1, q2
:
q1 = q2 + 1
q2 = 1 + q1
```

Überladen von Operatoren (Beispiel)

```
module rationals
  implicit none
  private

  public :: rational, operator(+)

  type rational
    integer :: num
    integer :: denom
  end type rational

  interface operator(+)
    module procedure rational_add
  end interface operator(+)

contains

  function rational_add(q1, q2) result(res)
    type(rational), intent(in) :: q1, q2
    type(rational) :: res
    :
  end function rational_add
end module rationals
```

```
program TestRationals
  use rationals
  implicit none

  type(rational) :: q1, q2, q3

   q1 = rational(1, 2)
   q2 = rational(3, 4)
   q3 = q1 + q2
   write (*, "(2I10)") q3

end program TestRationals
```

Statische Felder in benutzerdefinierten Datentypen

- Benutzerdefinierte Typen können auch statische Felder enthalten:

```
type rationals
  integer :: numdenum(2)
end type rationals
```

- Bei der Initialisierung muss für die Feldkomponente ein Feld mit der richtigen Größe übergeben werden:

```
type(rationals) :: q1
:
q1 = Rationals([  1, 2 ])
```

Falsch!

```
type(rationals) :: q1
:
q1 = rationals([ 1, 2, 3 ])
```

- Bei der Zuweisung wird der Feldinhalt **kopiert**:

```
type(rationals) :: q1, q2

q1 = rationals([ 1, 2 ])
q2 = q1
write(*,*) q2%numdenum ! 1 2
```

- Feldkomponente kann wie ein normales Feld behandelt werden

```
type(rationals) :: q1

q1%numdenum(1) = 1
q1%numdenum(2) = 2
```

Dynamische Felder in benutzerdefinierten Datentypen (F2003)

- Fortran 2003 Feature:

```
type polynomial
  real(dp), allocatable :: coeffs(:)
end type polynomial
```

- Kann, wie normales dynamisches Feld behandelt werden:

```
type(polynomial) :: p1

allocate(p1%coeffs(10))
coeffs(1:10) = 2.0_dp
deallocate(p1%coeffs)
```

- Wenn der Geltungsbereich des benutzerdefinierten Datentyps verlassen wird (und der Datentyp vernichtet wird), wird die dynam. Feldkomponente automatisch freigegeben:

```
subroutine test()
  type(polynomial) :: p1

  allocate(p1%coeffs(100))
  :
end subroutine test()
```

Beim Verlassen des Unterprogrammes wird `deallocate(p1%coeffs)` automatisch ausgeführt

Dynamische Felder in benutzerdefinierten Datentypen (F2003)

- Bei Zuweisungen werden dynamische Feldkomponenten auf der rechten Seite deallokiert, und bei Bedarf (und nur bei Bedarf!) allokiert

```
type(polynomials) :: p1, p2

p1 = polynomials(&
  &[ 1.0_dp, -2.0_dp, 0.5_dp ])
p2 = polynomials(&
  &[ 3.0_dp, -1.0_dp ])

p2 = p1

deallocate(p1%coeffs)
p2 = p1
```

Äquivalente:

```
allocate(p1%coeffs(3))
p1%coeffs(:) = [ 1.0_dp, -2.0_dp, 0.5_dp ]

allocate(p2%coeffs(2))
p2%coeffs(:) = [ 3.0_dp, -1.0_dp ]

deallocate(p2%coeffs)
allocate(p2%coeffs(size(p1%coeffs))
p2%coeffs = p1%coeffs

deallocate(p2%coeffs)
```

- Wenn benutzerdefinierte Typen mit dynamischen Datenfelder als **intent(out)** Parametern vorkommen, werden die entsprechenden dynamische Felder beim Eintreten in das Unterprogramm deallokiert (analog zu normalen dynamischen Felder)

Zugriffkontrolle für Variablen in benutzerdefinierten Typen

- Wenn benutzerdefinierter Typ den internen Status eines Objektes beschreibt, kann Manipulation außerhalb des Modules unerwünscht sein:

```
module GreatIterator
  implicit none
  private

  public :: grit, init_iterator

  type grit
    integer :: status ! should be > 0
  end type grit

contains

  subroutine init_iterator(iter)
    type(grit), intent(inout) :: iter

    iter%status = 1

  end subroutine
  :
```

```
program GreatIteratorTest
  use GreatIterator
  implicit none

  type(grit) :: myiterator

  call init_iterator(myiterator)
  myiterator%status = -2
```

Könnte zu unerwünschten Effekten führen

Zugriffkontrolle für Variablen in benutzerdefinierten Typen (#2)

- Die Anweisung **private** macht *alle* Komponenten des benutzerdefinierten Types privat
- Es können nur Unterprogramme des Modules, das den **Typ deklariert**, auf die Komponenten zugreifen, egal wo die Variable mit diesem Typ deklariert wurde

```
module GreatIterator
  implicit none
  private

  public :: grit, init_iterator

  type grit
    private
    integer :: status
    integer :: index
  end type grit

contains

  subroutine init_iterator(iter)
    type(grit), intent(inout) :: iter

    iter%status = 0
    iter%index = 1
```

```
program GreatIteratorTest
  use GreatIterator
  implicit none

  type(grit) :: myiterator

  call init_iterator(myiterator)
  write(*,*) iter%index
  myiterator%status = -2
```

Falsch, außerhalb des Modules kann nicht direkt auf die Komponenten zugegriffen werden

OK, Unterprogramme im Modul dürfen die Komponenten lesen und schreiben.

Aufgabe – fortgeschrittene Programmierung

- Schreiben Sie ein Modul, das die grundlegende Arithmetik mit rationalen Zahlen ermöglicht.
- Es sind die vier Grundoperationen (+, -, *, /) und die Funktionen min(a, b) und max(a, b) zu implementieren.
- Die Operationen und die min() und max() Funktionen müssen auch mit einer rationalen Zahl und einem Integer ausgeführt werden können.
- Das folgende Testprogramm muss mit dem Modul ausgeführt werden können:

Aufgabe – fortgeschrittene Programmierung

```
program test_rationals
  use rationals
  implicit none

  type(rational) :: q1, q2, q3
  character(len=*), parameter :: formstr = "('(',I4,'/',I4,')')"

  q1 = rational(3, 4)
  q2 = rational(1, 3)
  q3 = -2

  write(*, formstr) q1, q2, q3
  write(*, formstr) q1 + q2
  write(*, formstr) q1 - q2
  write(*, formstr) -q2
  write(*, formstr) 1 + q1 + 3
  write(*, formstr) 2 * q2 * 3
  write(*, formstr) 2 / q3 / 3
  write(*, formstr) max(q1, q2)
  write(*, formstr) min(q3, 0)
  write(*, formstr) max(0, q2)

end program test_rationals
```