Today‘s Goals

1. Learn about pointers, generic procedures and operators (examples of overloading)
2. Continue working on convection program.
Projects: start thinking about

Agree topic with me before final lecture
Project
(optional, 1 KP)

1. Chosen topic, agreed upon with me (suggestions given, also ask the advisor of your MSc or PhD project).
   - Due end of Semesterprüfung (15 Feb 2019)
   - Start planning soon!
Makefiles

• If your programs are split over several files, it may be easiest to write instructions for compiling them in a makefile (this applies mainly to unix-like systems)

• This also makes sure that the various options you use remain the same

• The makefile defines dependencies, so it only recompiles source files that have changed since the last compilation

• See manual pages for full information
Example makefile

% example makefile

FFLAGS = -O2
OBJECTS = main.o modules.o

myproject: $(OBJECTS)
gfortran -o myproject $(OBJECTS)

main.o : main.f90 stuff.mod
gfortran -c main.f90

stuff.mod : module.o

modules.o : modules.f90
gfortran -c modules.f90

USAGE: Just type in “make” or “make myproject”
Intrinsic functions

• There are many more intrinsic functions than we have used!
• It is good to browse a list so that you know what is available, for example at
  http://www.nsc.liu.se/~boein/f77to90/a5.html
• This doesn’t list cpu_time(), which is useful for checking how long parts of the code take
Pointers

• Point to either
  – (i) data stored by a variable declared as a target, or
  – (ii) another pointer, or
  – (iii) an area of allocated memory

• (i) and (ii) are mainly useful for creating interesting data structures (like linked lists); for us (iii) will be the most useful.

• “=>” is used to point a pointer at something

• See examples next slide

• Functions can also return pointers (“pointer functions”)
Pointers to scalar variables

program pointless
  implicit none
  real, pointer:: p
  real, target:: a=3.1, b=4.7

  p=>a          ! point p to a
  print*, p     ! prints 3.1 (value of a)
  p=4.0         ! changes a to 4.0
  print*, a     ! prints 4.0
  p=>b          ! now points to b
  print*, b     ! prints 4.7
  print*, associated (p) ! test status: prints true
  nullify(p)    ! points p at nothing
  print*, associated (p) ! prints false
end program pointless
program pointarray
  implicit none
  integer, parameter :: n=7
  real, target :: T(n,n)
  real, pointer :: bot_boundary(:,), &
                  top_boundary(:,), center4(:,,:) 

  top_boundary => T(:,1)
  bot boundary => T(:,n)
  center4 => T(n/2:n/2+1,n/2:n/2+1)

  call random_number(T)
  top_boundary = 0.0 ! easy way to do
  bot_boundary = 1.0 ! boundary conditions
  center4 = 1.0

  print*, T 
end program pointarray
Pointers in defined types
similar to allocatable array (i.e., not pointing to a variable)

program pointertype
  implicit none

  type array2D
    real, pointer:: a(:, :) ! 2D array
  end type array2D

  type(array2D), allocatable:: T(:) ! 1D array of 2D arrays
  integer i, n, ng

  print '(a, $)', "Number of multigrid levels:"
  read*, ng
  allocate (T(ng)) ! allocate number of grids
  do i = 1, ng
    n = 2**(i+1) ! #grid points in this grid
    print*, 'Allocating grid', n, n
    allocate (T(i) % a(n, n)) ! each grid
  end do

  !..... rest of program

end program pointertype
Generic procedures
(i.e., using a generic name to access different procedures)

• e.g., the intrinsic (built-in) $\texttt{sqrt}$ function. There are several versions: $\texttt{sqrt}$(real), $\texttt{dsqrt}$(double precision), $\texttt{csqrt}$(complex). Use the generic name and the correct one will automatically be used.

• You can define the same thing. Define similar sets of procedures then define a generic $\texttt{interface}$ to them (see example next slide).

• Easiest way: in a $\texttt{module}$
module goodstuff
  implicit none

  interface apbxc  ! define generic interface
    module procedure rapbxc,iapbxc
  end interface

  contains

  real function rapbxc(a,b,c)  ! actual function
    real,intent(in):: a,b,c
    rapbxc = a+b*c
  end function rapbxc

  integer function iapbxc(a,b,c)
    integer,intent(in):: a,b,c
    iapbxc = a+b*c
  end function iapbxc

end module goodstuff

!---------------------------------------------------

program generic
  use goodstuff
  implicit none
  real:: a=1.2,b=3.4,c=5.6
  integer:: i=2,j=3,k=4

  print*,apbxc(a,b,c)  ! real arguments
  print*,apbxc(i,j,k)  ! integer arguments

end program generic
Overloading

- Overloading means that one operator or procedure name is used to refer to several procedures: which one is used depends on the variable types.

- Examples
  - \texttt{apbxxc} is overloaded with \texttt{rapbxxc} and \texttt{iapbxxc}
  - \texttt{*,+,-,\textbackslash} are overloaded with integer, real, complex versions in different precisions

- You can overload existing operators, or define new overloaded operators or procedures
Overloading existing + and –

module coordstuff
  implicit none

  type point, intent(in) :: a, b
  type(point):: pointplus

contains
  function pointplus(a,b)
    type(point):: pointplus
    pointplus%x = a%x + b%x
    pointplus%y = a%y + b%y
    pointplus%z = a%z + b%z
  end function pointplus

  function pointminus(a,b)
    type(point):: pointminus
    pointminus%x = a%x - b%x
    pointminus%y = a%y - b%y
    pointminus%z = a%z - b%z
  end function pointminus
end module coordstuff

Useful for defined types

program test
  use coordstuff
  type(point):: p1,p2,p3
  p1%x=1.2; p1%y=0. ; p1%z=3.1
  p2%x=0. ; p2%y=1.2; p2%z=1.7
  p3 = p1 - p2 ! using overloaded -
  print*,p3
  p3 = p1 + p2 ! using overloaded +
  print*,p3
end program test
Defining new operator distance.

```fortran
module coordsagain
  contains
  interface operator (distance)
    new operator
      contains
      module procedure pointseparation
    end operator
  end interface
end module coordsagain

real function pointseparation(a, b)
  type(point), intent(in) :: a, b
  real:: pointseparation = sqrt((a%x-b%x)**2+(a%y-b%y)**2+(a%z-b%z)**2)
end function pointseparation
```

```
! using new operator
!

d = p1.distance(p2)
```

Overloading "=". Useful for conversion between different types: in this case point to real.

Must use subroutine with 1st argument intent(out) and 2nd argument intent(in).

```fortran
module coords3
  implicit none
  type point ! defined type
    real:: x,y,z
  end type point

  interface assignment (=) ! new "="
    module procedure absvec ! converts point to real
  end interface contains

  subroutine absvec(a,b) ! calculates distance
    real,intent(out):: a ! from origin
    type(point),intent(in):: b
    a = sqrt(b%x**2+b%y**2+b%z**2)
  end subroutine absvec

end module coords3

!---------------------------------------------
program test
  use coords3
  type(point):: p1
  real:: d
  p1%x=1.2; p1%y=0.; p1%z=3.1

  d = p1 ! using new =,
  print*,d
end program test
```
Combine operators into 1 module

• Generic procedures
• New operators
• Overloaded operators (e.g., +, -, =)
module coords
  implicit none

  type point ! defined type
    real :: x,y,z
  end type point

interface apbxc ! define generic interface
  module procedure rapbxc,iapbxc
end interface

interface operator (.distance.) ! new operator
  module procedure pointseparation
end interface

interface operator (+) ! new version of +
  module procedure pointplus
end interface

interface operator (-) ! new version of -
  module procedure pointminus
end interface

interface assignment (=) ! new "="
  module procedure absvec ! converts
t ! point to real
end interface

contains
The procedures

```fortran
real function rapbxc(a,b,c) ! real version of fn
 real, intent(in):: a,b,c
 rapbxc = a+b*c
end function rapbxc

integer function iapbxc(a,b,c) ! int version
 integer, intent(in):: a,b,c
 iapbxc = a+b*c
end function iapbxc

real function pointseparation(a,b) ! for .distance.
 type(point), intent(in):: a,b
 pointseparation = sqrt( &
 (a%x-b%x)**2+(a%y-b%y)**2+(a%z-b%z)**2)
end function pointseparation

function pointplus(a,b) ! for +
 type(point):: pointplus
 type(point), intent(in):: a,b
 pointplus%x = a%x + b%x
 pointplus%y = a%y + b%y
 pointplus%z = a%z + b%z
end function pointplus

function pointminus(a,b) ! for -
 type(point):: pointminus
 type(point), intent(in):: a,b
 pointminus%x = a%x - b%x
 pointminus%y = a%y - b%y
 pointminus%z = a%z - b%z
end function pointminus

subroutine absvec(a,b) ! for = (distance
 real, intent(out):: a ! from origin)
 type(point), intent(in):: b
 a = sqrt(b%x**2+b%y**2+b%z**2)
end subroutine absvec

end module coords
```
The test program

program test
  use coords
  real :: a=1.2, b=3.4, c=5.6, d
  integer :: i=2, j=3, k=4
  type(point) :: p1, p2, p3
  p1%x=1.2; p1%y=0.; p1%z=3.1
  p2%x=0.; p2%y=1.2; p2%z=1.7

  print*, apbxc(a, b, c)  ! real arguments
  print*, apbxc(i, j, k)  ! integer arguments

  p3 = p1 - p2  ! using overloaded -
  p3 = p1 + p2  ! using overloaded +

  d = p1  ! using overloaded =,

  d = p1.distance.p2  ! using new operator
end program test