Programming Techniques — Pointers and Derived Types

Lecture: Pointers, Dynamic Data, and Derived Types

Hannu Parviainen

Universidad de la Laguna

September 29, 2025

Dynamic Memory Allocation

Static memory allocation

Concept

- Array size is fixed at compile time.
- Memory allocated once when the program starts.
- Lifetime = entire execution of the program (or scope).
- Efficient, but inflexible when array size is not known in advance.

```fortran
program static_demo
  implicit none
  real, dimension(100) :: x
  integer :: i

  do i = 1, 100
    x(i) = i * 0.1
  end do

  print *, "x(100) = ", x(100)
end program static_demo
```

Dynamic memory allocation

Concept

- Size and shape chosen at runtime.
- Memory allocated with `allocate`, released with `deallocate`.
- Lifetime = until explicitly deallocated.
- More flexible, but requires careful management.

```
program dynamic_demo
  implicit none
  real, dimension(:), allocatable :: y
  integer :: n, i, ierr

  print *, "Enter size:"
  read *, n
  allocate(y(n), stat=ierr)
  if (ierr /= 0) then
    print*, 'Memory allocation failed'
    stop
  end if

  do i = 1, n
    y(i) = i * 0.5
  end do

  print *, "y(n) = ", y(n)
  deallocate(y)
end program dynamic_demo
```

Allocatable Arrays in Fortran

Declaration

```
integer, dimension(:),   allocatable :: ages
real,    dimension(:,:), allocatable :: speed
```

Allocation

```
integer :: isize, ierr
read *, isize

allocate(ages(isize), stat=ierr)
if (ierr /= 0) print *, 'ages: allocation failed'

allocate(speed(0:isize-1, 10), stat=ierr)
if (ierr /= 0) print *, 'speed: allocation failed'
```

Note

stat is optional but recommended to detect errors.

Deallocation and Status

Deallocate

```
integer :: ierr
if (allocated(ages)) deallocate(ages, stat=ierr)
if (allocated(speed)) deallocate(speed, stat=ierr)
```

Guidelines

- Only deallocate allocatable arrays that are currently allocated.
- Prefer checking `allocated(array)` before deallocation.
- In procedures, deallocate before exit.

Pointers

Fortran Pointers: Introduction

What is a pointer?

- A **pointer** is a variable that does not store a value directly, but instead **points to a memory location**.
- In Fortran, pointers can be associated with
  - other variables,
  - array sections,
  - dynamically allocated memory.

Why are they useful?

- Enable **dynamic memory management** (allocate and free memory at runtime).
- Allow multiple variables to **share the same data**.
- Needed for building **linked data structures** (lists, trees, graphs).
- Provide flexibility similar to references in modern languages.

Python vs. Fortran Variables

Key difference

- **Python variables** are *references* to objects in memory.
  - Assigning one variable to another does not copy the object.
  - Example: `b = a` makes `b` point to the same object as `a`.
- **Fortran variables** are normally *values* stored directly.
  - Assigning one variable to another copies the value.
  - No implicit references like in Python.

Python variables act like Fortran pointers by default, while plain Fortran variables are independent copies.

Pointer Declaration

Definition

- A variable with the `pointer` attribute.
- Has static type, kind, and rank determined by its declaration.

Examples

```
real, pointer :: ptor
real, dimension(:,:), pointer :: ptoa
```

- `ptor` is a pointer to a scalar real target.
- `ptoa` is a pointer to a 2D array of reals.

Target Declaration

Targets must have the `target` attribute

```
real, target :: x, y
real, dimension(5,3), target :: a, b
real, dimension(3,5), target :: c
```

- `x` or `y` may become associated with `ptor`.
- `a`, `b`, or `c` may become associated with `ptoa`.

Pointer Manipulation

Operators

- => pointer assignment: alias pointer with a target.
- = normal assignment: assign value to space pointed at.

Pointer assignment makes the pointer and target reference the same space, while normal assignment changes the value.

Pointer Assignment

```
real, target :: x, y
real, pointer :: ptor

x = 3.14159
ptor => y
ptor = x
```

- `x` and `ptor` have the same value.
- `ptor` is an alias for `y`, so the assignment sets `y = 3.14159`.
- Changing `x` later does not change `ptor` or `y`.

Dynamic Targets

Targets can be created dynamically by allocation

```
allocate(ptor, stat=ierr)
allocate(ptoa(n*n, 2*k-1), stat=ierr)
```

- First: allocates a scalar real as target of `ptor`.
- Second: allocates a rank-2 real array as target of `ptoa`.
- Re-allocating an already associated pointer is not an error.

Association Status

Test with associated()

```
associated(ptoa)
associated(ptoa, arr)
```

- Returns `.true.` if pointer is defined and associated.
- Second form tests association with a specific target.

Pointer Disassociation

Nullification

```
nullify(ptor)
```

- Breaks pointer–target connection.
- Good practice to nullify before use.

Deallocation

```
deallocate(ptoa, stat=ierr)
```

- Breaks connection and deallocates target.

Practical Example

```fortran
real, dimension(100,100), target :: app1, app2
real, dimension(:,:), pointer :: prev_app, next_app, swap

prev_app => app1
next_app => app2
prev_app = initial_app(...)

do
  next_app = iteration_function_of(prev_app)
  if (abs(maxval(next_app-prev_app))<0.0001) exit
  swap => prev_app
  prev_app => next_app
  next_app => swap
end do
```

Note

Pointers avoid copying large matrices in iterative algorithms.

Derived Types

Derived Types

Example: 3D vector type

```
module geometry

type vector3d
  real :: x, y, z
end type vector3d

end module geometry
```

- Basic Fortran types can be combined to create more complex *derived* types.
- Derived type definitions should be placed in a module.

Example: type usage

```
program test_geometry
    use geometry
    implicit none
    type(vector3d) :: a
    a = vector3d(0.0, 1.0, 0.0)
    print *, a
end program test_geometry
```

- Variables are declared using the type statement.
- Type values can be initialised in the same order as they are defined inside the type.

Supertypes

Previously defined types can be used as components of other derived types.

Example: sphere

```
module geometry
type vector3d
  real :: x, y, z
end type vector3d

type sphere
  type(vector3d) :: centre
  real :: radius
end type sphere
end module geometry
```

Example: sphere usage

```
program test_geometry
  use geometry
  implicit none
  type(sphere) :: s
  s = sphere(vector3d(0.0, 1.0, 0.0), 5.0)
  print *, s
end program test_geometry
```

Derived type assignment

Ways to assign

- Component-by-component using %.
- As a whole object using a constructor.
- Assignment between two objects of the same derived type is intrinsic.

The derived-type component of a composite (e.g., sphere%centre) must also be set via a constructor.

```
! component by component
pt1%x = 1.0
p%radius = 3.0
p%centre%x = 1.0

! whole-object with constructors
pt1 = vector3d(1., 2., 3.)
p%centre = vector3d(1., 2., 3.)
p = sphere(p%centre, 10.)
p = sphere(vector3d(1.,2.,3.), 10.)

! intrinsic assignment between objects
ball = p
```

Derived type I/O

Unformatted I/O

- If a derived type has no pointer or private components, it can be printed/read "normally."
- I/O proceeds component-by-component in order.

Example equivalence shown at right.

```
print *, p

print *, p%centre%x, p%centre%y, &
         p%centre%z, p%radius
```

Pointer components of derived types

Allocatable vs pointer

- allocatable arrays cannot be components in a derived type; pointer components can.
- Enables dynamic-size structures such as growable strings.

Example: a vector-of-characters pointer for a simple string type.

```
type vstring
  character, dimension(:), pointer ::
        chars
end type vstring

type(vstring) :: pvs1
! ...
allocate(pvs1%chars(5))
pvs1%chars = (/"H","e","l","l","o"/)
! prints: H e l l o
```

Arrays of pointers

Notes

- You can make arrays whose elements are themselves pointers.
- You cannot reference a whole array of pointer components in one go.
- If desired, the array-of-pointers container could be `allocatable`.

See valid vs. invalid examples at right.

```
type iptr
  integer, pointer :: compon
end type iptr

type(iptr), dimension(100) :: ints

! ok:
ints(10)%compon

! not ok (whole array component):
! ints(:)%compon
```

Custom Operators

Custom Operators

- Fortran allows you to define complex derived types.
- But using them directly for calculations leads to cumbersome code (and bugs, lots and lots of bugs)

```fortran
module geometry
  implicit none
  type :: vector2d
     real :: x, y
  end type vector2d
end module geometry
```

```fortran
program test
  use geometry
  implicit none
  type(vector2d) :: a, b, c
  real :: s = 2.3
  a = vector2d(0.0, 1.0)
  b = vector2d(1.0, 1.0)
  c = vector2d(s*a%x+b%x, s*a%y+b%y)
end program test
```

Custom Operators

Code can be simplified (and made more robust) by using functions.

```
module geometry
  implicit none
  type :: vector2d
     real :: x, y
  end type vector2d

contains
  pure type(vector2d) function sumvv(a, b)
    type(vector2d), intent(in) :: a, b
    sumvv = vector2d(a%x + b%x, a%y + b%y)
  end function sumvv

  pure type(vector2d) function mulrv(a, b)
    real, intent(in) :: a
    type(vector2d), intent(in) :: b
    mulrv = vector2d(a*b%x, a*b%y)
  end function mulrv
end module geometry
```

```
program test
  use geometry
  implicit none
  type(vector2d) :: a, b, c
  real :: s = 2.3
  a = vector2d(0.0, 1.0)
  b = vector2d(1.0, 1.0)
  c = sumvv(mulrv(s, a), b)
end program test
```

Custom Operators

And defining custom operators can make your code even cleaner.

```
module geometry
  implicit none
  type :: vector2d
     real :: x, y
  end type vector2d

  interface operator(+)
     module procedure sumvv, sumrv
  end interface

  interface operator(*)
     module procedure mulrv
  end interface

contains
  pure type(vector2d) function sumvv(a, b)
    type(vector2d), intent(in) :: a, b
    sumvv = vector2d(a%x + b%x, a%y + b%y)
  end function sumvv

  pure type(vector2d) function mulrv(a, b)
    real, intent(in) :: a
```

```
program test
  use geometry
  implicit none
  type(vector2d) :: a, b, c
  real :: s = 2.3
  a = vector2d(0.0, 1.0)
  b = vector2d(1.0, 1.0)
  c = s*a + b
end program test
```

How to Define a Custom Operator

Custom operators are defined inside a module using the `operator` keyword.

- Standard operators (+, -, *, /)

```
interface operator(+)
   module procedure newsum
end interface
```

- A new operator named `.op.`

```
interface operator(.op.)
   module procedure newop
end interface
```

The functions implementing the operators are defined later in the module after the `contains` keyword.