Modern Programming Languages: Fortran90/95/2003/2008

Why we need modern languages (Fortran/C++)

How to write code in modern Fortran

Lars Koesterke

Texas Advanced Computing Center
The University of Texas at Austin

November 10, 2011

This is an Intermediate Class

- You know already one computer language
- You understand the very basic concepts:
  - What is a variable, an assignment, function call, etc.?
  - Why do I have to compile my code?
  - What is an executable?
- You (may) already know some Fortran
- You are curious about what comes next
- What are the choices?
- How to proceed from old Fortran (or C), to much more modern languages like Fortran2003/2008 (and C++)?

Outline

- Motivation
- Modern Fortran
- Object-Oriented Programming: (Very) Short Version

Why do we (have to) learn advanced languages?

Basic features (BASIC)

- Variables — Data containers for Integers, Reals, Characters ,Logicals
  Arrays: Vectors ,Matrices
- Basic operators — arithmetic (+, −, *, /) logical, lexical, etc.
- Control constructs — if/else-if, case/switch, goto, ...
- Loops — do/for, while/repeat, etc.
- I/O — All languages provide sophisticated mechanisms for I/O (ASCII, binary, streams): **Not covered!**

Is that enough to write code?
My answer: No!

Subprograms: subroutines and functions

enables us to repeat operations on different data
enables us to savoid code replication

Starting with: Fortran77

- basic language (BASIC): allows to write 500 lines of code
- w/ subprograms: we can do much, much better

Old Fortran (Fortran77) provides **only** the absolute Minimum!

And these languages (Fortran77 and C) have flaws:

- Fortran77: No dynamic memory allocation (on the heap)
  - common blocks, equivalence statements

    old & obsolete constructs
    clunky style, missing blanks
    old (legacy) code is usually cluttered
- C: Call by value, no multidimensional arrays
  - Pointer (de)referencing everywhere, for no **good** reason

Fortran77 and C are simple languages
and they are (kind-of) easy to learn

If Fortran77 and C are so simple,

Why is it then so difficult to write good code?

Is simple really better?

- Using a language allows us to express our thoughts (on a computer)
- A more sophisticated language allows for more complex thoughts
- I argue: **Fortran77** and plain **C** are (way) too simple
- Basics $+ 1$ plus the flaws are not enough!

We need better tools!

- The basics without flaws
  - Language has to provide new (flawless) features
  - User has to avoid old (flawed) features
- more language elements to get organized
  $\Longrightarrow$ **Fortran90/95/2003** and C++

So, these languages (Fortran77 and C) are easy to learn?

... are you kiddin'? They are not!
We want to get our science done! Not learn languages!

How easy/difficult is it really to learn Fortran77 and C?

The concept is easy:
Variables, Arrays, Operators, If, Do, Subroutines/Functions

- I/O
- Syntax
- Rules & regulations, the fine print
- Conquering math, developing algorithms,
  the environment: OS, compiler, hardware, queues, etc.

- I/O details
  - print to screen
  - read/write from/to files
  - from ASCII to binary
  - from basic to efficient to parallel

- parallel computing: MPI, OpenMP, cudA, ...
- ... and the flaws $\Longrightarrow$ simple things will be complicated

Invest some time now, gain big later!

Remember: so far, we have only the Basics + Functions/Subroutines

Modern Fortran starts here!

- Modern style
  - Free format
  - Attributes
  - implicit none
  - do, exit, cycle, case
  - Single and double precision
- Fixing the flaws
  - Allocatable arrays
  - Structures, derived types
- Module-oriented Programming
  - internal subprograms
  - private, public, protected
  - contains
  - use
  - Explicite interfaces
  - Optional arguments & intent
- Formula translation
  - Array syntax, where and forall statement
  - Extended & user-defined operators
  - Functions: elemental, inquiry, mathematical
- Odds and Ends
  - Fortran pointers (References)
  - Command line arguments
  - Environment variables
  - Preprocessor
  - Interoperability with C (binding)
- Performance considerations
- Object oriented programming

Free Format

- Statement may start at the first column (0−132 characters)
- Exclamation mark (**!**) starts a comment (not in literal strings)
- Blanks are significant: Not allowed in keywords or variables
- Continuation with an ampersand (**&**) as the last character
- Mulitple statements in one line separated by a semicolon (**;**)

Style example

```
program style
print *, 'This statement starts in column 1'
i = 5; j = 7    ! Two statements in one line
                ! Comment with an exclamation mark
i = &           ! Line with continuation
    j * j + j
end
```

Blanks, blank lines, and comments

- Use blanks, blank lines, and comments freely
- Use indentation

Good

```
program square
! This program calculates ...

implicit none
real :: x, x2

x  = 5.
x2 = x * x
if (x == 13.) print *, 'Lucky'

end
```

Bad

```
program square
x=5.
x2=x*x
if(x.eq.13)print*,'Lucky'
end
```

Good

```
program square
! This program calculates ...

implicit none
integer :: i
real    :: x, x2

do i=1, 20
  x  = real(i)
  x2 = x * x
  if (x == 13.) print *, Lucky
enddo

end
```

Bad

```
      program square
      do 100 i=1,20
      x=i
      x2=x*x
      if(x.eq.13)print*,...
100   continue
      end
```

Attributes

Style example

```
program style
integer                              :: i, j
real                                 :: x
real, parameter                      :: pi = 3.1415
real, dimension(100)                 :: array
real, dimension(:,:), allocatable :: dyn_array_2d
```

- General form
  integer :: name
  real, <attributes> :: name
- attributes are:
  parameter, dimension, allocatable, intent, pointer, target, optional, private, public, value, bind, etc.

Implicit none

Implicit type declaration

```
program implicit
implicit none      ! use to disable the default
```

- Default type of undeclared variables:
  All variables starting with the letter i, j, k, l, m, n are integers
  All other variables are real variables
- Turn default off with: implicit none
- Strongly recommended (may not be right for everybody, though)

Loops: do, while, repeat

do-Loop

```
do i=1, 100, 8 ! No label
               ! loop-variable, start, increment
...
enddo
```

while-Loop

```
i = 0
do
  if (i > 20) exit
  i = i + 1
enddo
```

repeat-Loop

```
i = 0
do
  i = i + 1
  if (i > 20) exit
enddo
```

- Use the exit statement to “jump” out of a loop

Loops: exit and cycle

Exit anywhere

```
do i=1, 100
  x = real(i)
  y = sin(x)
  if (i > 20) exit
  z = cos(x)
enddo
```

Skip a loop iteration

```
do i=1, 100
  x = real(i)
  y = sin(x)
  if (i > 20) cycle
  z = cos(x)
enddo
```

- exit: Exit a loop
- cycle: Skip to the end of a loop
- Put exit or cycle anywhere in the loop body
- Works with loops with bounds or without bounds

Nested loops: exit and cycle

Exit Outer Loop

```
outer: do j=1, 100
  inner: do i=1, 100
    x = real(i)
    y = sin(x)
    if (i > 20) exit outer
    z = cos(x)
  enddo inner
enddo outer
```

Skip an outer loop iteration

```
outer: do j=1, 100
  inner: do i=1, 100
    x = real(i)
    y = sin(x)
    if (i > 20) cycle outer
    z = cos(x)
  enddo inner
enddo outer
```

- Constructs (do, if, case, where, etc.) may have names
- exit: Exit a nested loop
- cycle: Skip to the end of an outer loop
- Put exit or cycle anywhere in the loop body
- Works with loops with bounds or without bounds

Case

```
integer :: temp_c
! Temperature in Celsius!
select case (temp_c)
case (:-1)
  write (*,*) 'Below freezing'
case (0)
  write (*,*) 'Freezing point'
case (1:20)
  write (*,*) 'It is cool'
case (21:33)
  write (*,*) 'It is warm'
case (34:)
  write (*,*) 'This is Texas!'
end select
```

- case takes ranges (or one element)
- works also with characters
- read the fine-print

Variables of different kind values

```
integer :: i, my_kind
real    :: r
                                      ! Selection based on
                                      !   precision
print *, kind(i), kind(r)             ! prints 4 4 (most compilers)
my_kind = selected_real_kind(15)      ! select a real that has
                                      !   15 significant digits
print *, my_kind                      ! prints 8
```

```
integer, parameter :: k9 = selected_real_kind(9)
real(kind=k9)      :: r

r = 2._k9; print *, sqrt(r)       ! prints 1.41421356237309
```

Variables of different kind values: The sloppy way

- There are only 2(3) kinds of reals: 4-byte, 8-byte (and 16-byte)
- The kind-numbers are 4, 8, and 16 (most compilers!)
- Kind number may not be byte number!
- Selection based on the number of bytes

```
real*8        :: x8    ! Real with 8 bytes (double precision)
real(kind=8) :: y8    ! same, but not completely safe
real*4        :: x4    ! Real with 4 bytes (single precision)
integer*4     :: i4    ! Integer single precision
integer*8     :: i8    ! Integer double precision

x8 = 3.1415_8          ! Literal constant in double precision
i8 = 6_8               ! same for an integer
```

- `real*8`, `real*4`: works well with MPI_Real8 and MPI_Real4

Variables of different kind values

- Do not use 'double' in your definition
- double refers to something; it's double of what?
- double precision, dble(...)
- Select appropriate precision at compile time: ifort -r4, ifort -r8
- Compiler flag also elevates the unnamed constants

```
real*8                :: x8, y8
real*4                :: x4, y4
integer               :: i

y8 = 3.1415                 ! 3.1415 is an unnamed constant
                            !  with -r8: 8 bytes
x4 = real(i)
x8 = dble(i)                ! Old style, using dble
x8 = real(i, kind=8)        ! New style using the kind parameter
```

Fixing the Flaws

Allocatable arrays

- flexible size
- allocated on the heap
  - The size of the stack is severely limited (default: 2 GB)
  - Remedies are problematic (Intel: -mcmodel=medium -intel-shared)
- Always allocate large arrays on the heap!
  - Large arrays always have to be allocatable (heap) arrays, even if you do not need the flexibility to avoid problems with the limited size of the stack

Structures and derived types

- Organize your data
- Compound different variables into one type

Allocatable Arrays

- Variables live on the heap (vs. stack for scalars and static arrays)
- Declaration and allocation in 2 steps
- Declare an array as allocatable,
  use colons (:) as placeholders
- allocate/deallocate in the executable part
- Allocation takes time. Do not allocate too often.

```
program alloc_array
real, dimension(:),   allocatable :: x_1d  ! Attribute
real, dimension(:,:), allocatable :: x_2d  ! allocatable
...
read n, m
allocate(x_1d(n), x_2d(n,m), stat=ierror)  ! Check the
if (ierror /= 0) stop 'error'              ! error status!
...
deallocate(x)                              ! optional
```

Structures and Derived Types

- Declaration specifies a list of items (Derived Type)
- A Structure (a variable of a derived type) can hold
  - variables of simple type (real, integer, character, logical, complex)
  - arrays: static and allocatable
  - other derived types
  - A structure can be allocatable

```
program struct
type my_struct    ! Declaration of a Derived Type
  integer                               :: i
  real                                  :: r
  real*8                                :: r8
  real, dimension(100,100)              :: array_s ! stack
  real, dimension(:), allocatable       :: array_h ! heap
  type(other_struct), dimension(5)      :: os      ! structure
end type my_struct
```

Declaration of a Structure

Variables of Derived Type

```
program struct
type my_struct   ! Declaration of a Derived Type
...
end type my_struct

! Structures (Variables) of the the derived type my_struct
type(my_struct)                :: data
type(my_struct), dimension(10) :: data_array
```

Example: Structures

```
program people
type person
  character(len=10)           :: name
  real                        :: age
  character(len=6)            :: eid
end type person

type(person)                  :: you
type(person), dimension(10)   :: we

you%name = 'John Doe' ! Use (%)
you%age  = 34.2       ! to access
you%eid  = 'jd3456'   ! elements
```

```
we(1)%name = you%name
we(2)      = you

! Old style
! name, age, eid: arrays
call do_this(name,age,eid)
! Reduce parameter list
!   to one structure
call do_this_smart(we)
```

- Need more data $\Longrightarrow$ add a component to the derived type

From Functions to Modules

Let's step back for a second:

Why do we use Subprograms (Functions/Subroutines)?

Subroutines and Functions serve mainly 3 purposes:

- Re-use code blocks
- Repeat operations on different datasets

```
call do_this(data1)
call do_this(data2)
call do_this(data3)
```

- Hide local variables, so that the names can be re-used

```
subroutine do_this(data)
integer :: i, j       ! Local variables,
real    :: x, y, z    ! not accessible outside of the
                      !   subprogram
```

Modules are another, more flexible tool to Hide Content

Modules may contain all kind of things

- Derived Type declarations
- Variables and Arrays, etc.
  - Parameters (named constants)
  - Variables
  - Arrays
  - Structures
- Subprograms
  - Subroutines, Functions
  - other Modules
- Objects

Fortran 2008: Modules may contain Submodules.
Will make using Modules even nicer.
(Not implemented in Intel 12, yet)

Example: Constants and Variables

```
module mad_science
real, parameter :: pi = 3.   &
                   c  = 3.e8 &
                   e  = 2.7
real            :: r
end module mad_science

program go_mad
! make the content of module available
use mad_science
r = 2.
print *, 'Area = ', pi * r**2
end program
```

Example: Type Declarations

```
module mad_science
real, parameter :: pi = 3.   &
                   c  = 3.e8 &
                   e  = 2.7
real            :: r
type scientist
  character(len=10) :: name
  logical           :: mad
  real              :: height
end type scientist
end module mad_science
```

Example: Subroutines and Functions

```
module mad_science
real, parameter :: pi = 3.
type scientist
  character(len=10) :: name
  real              :: height
  logical           :: mad
end type scientist

contains
subroutine set_mad(s)
type(scientist) :: s
s%mad = .true.
end module mad_science
```

```
program go_mad
use mad_science

type(scientist) :: you
type(scientist), &
  dimension(10) :: we

you%name = 'John Doe'
call set_mad(you)
we(1)      = you
we%mad     = .true.
you%height = 5.
area       = you%height * pi
```

- Subprograms after the contains statement

Example: Public, Private Subroutine

```
module mad_science
contains

subroutine set_mad(s)
type(scientist) :: s
call reset(s)
s%mad = .true.

private
subroutine reset(s)
s%name = 'undef'
s%mad = .false.
```

- A module becomes accessible when the module is used
- Even more control: public and private components
- Default is public: all public content can be used from the outside of the module, i.e. by subprograms that use the module
- private items are only accessible from within the module
- Example: subroutine reset is only accessible by subroutine set_mad

Example: Public, Private Variables

```
module mad_science
real, parameter :: pi = 3.   &
                   c  = 3.e8 &
                   e  = 2.7
private
real, dimension(100) :: scratch
real, public         :: p_var

contains
subroutine swap(x, y)
real, dimension(100) :: x, y
scratch(1:100) = x(1:100)
x(1:100)       = y(1:100)
y(1:100)       = scratch(1:100)
```

- Default: public
- Private items not visible outside of the module
- private array scratch not accessible from outside of the module
- Keywords private or public can stand alone, or be an attribute

Example: Protected Variables

```
module mad_science
real, parameter :: pi = 3.   &
                   c  = 3.e8 &
                   e  = 2.7
integer, protected :: n
real, dimension(:), private &
       allocatable :: scratch

contains
subroutine alloc()
n = ...  ! n defined in the module
allocate (scratch(n))
```

- protected variables are visible on the outside
- protected variables cannot be modified outside the module
- protected variables may be modified inside of the module
- variable n is set in the module subroutine alloc
- n is visible to all subprograms that use the module
- n cannot by change outside of the module

Example: Rename Components of a Module

```
module mad_science
real, parameter :: pi = 3.
end module

program t
use mad_science, mad_pi => pi
real, parameter :: pi = 3.1415

print *, 'mad_pi = ', mad_pi
print *, '    pi = ', pi
end program
```

- Use module mad_science
- change the name of pi (so that you can declare your own and correct pi)
- mad_pi => pi: Refer to pi from the module as mad_pi
- renaming works with function names, too

```
prints mad_pi = 3
prints     pi = 3.1415
```

Interfaces: Implicit ⟹ Explicit

- Implicit interface: matching positions

```
subroutine s(a, b, c, n, ...)
...
call       s(x, y, z, m, ...)
```

- The subroutine may be compiled separately (separate file) from the other routine(s) or the main program that calls the subroutine
- The position is the only information available

Interfaces: Implicit ⟹ Explicit

- Explicit interface which does not solely rely on positional information

```
module my_module
contains
subroutine s(a, b, c, n, ...)
...
subroutine upper_level
use my_module
call       s(x, y, z, m, ...)
```

- Modules have to be compiled first
- Compilation of a module results in a .mod file
- At compile time (Subr. upper_level), the (content of the) module (my_module) is known through the .mod file (my_module.mod)
- Benefits:
  - Allows consistency check by the compiler
  - Assume-shape arrays, optional parameters, etc.

Passing an array

- Traditional scheme: Shapes of the actual and the dummy array (may) have to agree

```
integer, parameter :: n = 100
real, dimension(n) :: x
call sub(x, n)

subroutine sub(y, m)
integer            :: m
real, dimension(m) :: y
```

- You can, of course, play some games here
- The shape and the size do not have to match, but you have to explicitle declare the shape and size in the subroutine

Passing Assumed-shape arrays

```
module my_module
contains
subroutine sub(x)
real, dimension(:) :: x
print *, size(x)  ! prints 100

subroutine upper_level   ! calls the subroutine ‘‘sub’’
use my_module
real, dimension(100) :: y
call sub(y)
```

- Variable y is declared as an array in subroutine upper_level
- The subroutine (sub), “knows” the shape of the array

Example: Assumed-shape and Automatic Arrays

```
subroutine swap(a, b)
real, dimension(:)        :: a, b
real, dimension(size(a)) :: work  ! Scratch array
     ! work is an automatic array on the Stack
work = a                         ! uses Array syntax
a    = b                             ! Inquire with
b    = work                          ! lbound, ubound
end subroutine swap                  ! shape, size
```

- swap has to be in a module (explicit interface)
- calling routine has to use the module containing the subroutine swap
- No need to communicate the shape of the array
- size(a) returns the size of a, used to determine the size of work
- Automatic array work appears and disappears automatically

Intent: In, Out, InOut

- Formalize if a parameter is
  - Input: intent(in)
  - Output: intent(in)
  - Both: intent(inout)

```
subroutine calc(result, a, b, c, d)
! This routine calculates ...
!   Input: a, b, c
!   Output: result
!   d is scratch data: Input and Output
real, intent(out)   :: result
real, intent(in)    :: a, b, c
real, intent(inout) :: d      ! Default
```

- You would put this information in the comment anyway.
- Improves maintainability
- Compiler will check for misuse

Optional Arguments

- Optional arguments require an explicit interface
- Optional arguments may not be changed, if they are not passed

```
module my_module
subroutine calc(a, b, c, d)
real              :: a, b, c
real, optional    :: d
real              :: start
if (present(d)) then
  start = d
  d     = d_new
else
  start = 0.
endif
```

```
subroutine upper_level
use my_module
call calc(  1.,   2.,   3.,   4.)
call calc(  1.,   2.,   3.)
call calc(a=1., b=2., c=3., d=4.)

call calc(b=2., d=4., a=1., c=3.)
call calc(  1.,   2.,   3., d=4.)
call calc(  1.,   2., d=4., c=3)
```

- Positional arguments first, then keyword arguments

Optional Arguments

- Optional arguments require an explicit interface
- Optional arguments may not be changed, if they are not passed

```
module my_module
subroutine calc(a, b, c, d)
real           :: a, b, c
real, optional :: d
real           :: start
if (present(d)) then
  start = d
  d     = d_new
else
  start = 0.
endif
```

```
subroutine upper_level
use my_module
call calc(  1.,   2.,   3.,   4.)
call calc(  1.,   2.,   3.)
call calc(a=1., b=2., c=3., d=4.)

call calc(b=2., d=4., a=1., c=3.)
call calc(  1.,   2.,   3., d=4.)
call calc(  1.,   2., d=4., c=3)
```

- Positional arguments first, then keyword arguments

This just in from the Complaints Department

- Isn't it really easy to screw up in these advanced languages (Fortran2003 and C++)?
- If modern Fortran is so much like C++,
  Do I have to write Object-Oriented code in Fortran?
- Isn't C++ (supposed to be) quite ugly? Will my Fortran code be ugly, too?
- C++ does this name-mangling. That's hideous! Does Fortran do the same?
- There are so many features, do I need to master all of them to write good code?
- I'm new to Fortran. How much of the old stuff do I need to know?
- What is the bear minimum to get started?

A more complex language can create more confusion!

We all deal with that every day ...

A more complex language can create more confusion!

We all deal with that every day ...

... because as we know, there are known knowns;
there are things we know we know.

We also know there are known unknowns;
that is to say, we know there are some things we do not know.

But there are also unknown unknowns,
the ones we don't know we don't know ...

some politician

Perfectly valid point, but the presentation is lacking

Do I have to write Object-Oriented code?

No, but you have to learn (sooner or later) how to write module-oriented code.

Writing Object-Oriented code for access control is actually pretty nice!

If you problem/algorithm requires, you may add Object-Oriented code exploiting Polymorphism (supported in Fortran2003 & 2008).

Learn later, how to write Object-Oriented code in Fortran without performance penalty; Access control only.

Isn't C++ code (supposed to be) ugly?
Will my Fortran2003 code be ugly, too?

Write clean code

Clean code is not ugly (in any language: C++ and/or modern Fortran)

- Use blanks, blank lines, indentation
- Comment your code
- Use modern constructs
- Use the language in a clear, unambigious manner

C++ does name-mangling
Does Fortran do the same?

It's not a bug, it is a feature!

- It protects against misuse
- The objects (.o files) in your library (.a files) contain "protected" names
- If you do it right, name mangling causes no problems (see also chapter on *Interoperability with C*)

There are so many features.
Do I have to master all of them?

Here is how you get started:

- Do **not** use common blocks or equivalence statements!
  If you find yourself in a situation where you think they are needed, please revisit the modern constructs
- Use Heap arrays: allocate and deallocate (2 slides)
- Use structures to organzie your data (3 slides)
  $\Longrightarrow$ Heap arrays + structures:
  There is Absolutely! no need for common blocks and equivalence statements
- Use Modules: start writing module-oriented code (2 slides)

Here is how you get started: cont'd

Use Modules: start writing module-oriented code

- What to put in a Module:
  1. Constants (parameters)
  2. Derived type declarations
     avoid repeating parameter and derived type definitions. Sometimes physical constants are put in an *include* file. This should be done using a module.
  3. Variables (probably not?)
  4. Functions and Subroutines,
     move on by using the public, private and protected attributes
  5. Write Object-Oriented code without performance penalty
  6. Use Inheritance and Polymorphism with care

What about learning old Fortran (F77 and older)?

- Don't bother, if you don't have to
- Learn how to read code, assume that the code works correctly

Formula Tranlation

- Array syntax
- where construct
- forall construct
- Case study: Stencil Update
- User defined Operators
- Elemental Functions
- Inquiry Functions
- Odds and Ends

Simple Array Syntax

```
real                        :: x
real, dimension(10)         :: a, b
real, dimension(10,10)      :: c, d

a       = b
c       = d
a(1:10) = b(1:10)
a(2:3)  = b(4:5)
a(1:10) = c(1:10,2)
a       = x
c       = x
a(1:3)  = b(1:5:2) ! a(1) = b(1)
                   ! a(2) = b(3)
                   ! a(3) = b(5)
```

- Variables on the left and the right have to be conformable
- Number of Elements have to agree
- Scalars are conformable, too
- Strides can be used, too

Array constructor

```
real, dimension(4) :: x = [ 1., 2., 3. 4. ]
real, dimension(4) :: y, z
y = [ -1., 0., 1., 2. ]                   ! Array constructor
z(1:4) = [ (sqrt(real(i)), i=1, 4) ] ! with implicit
                                          ! loop
```

```
real, dimension(:), &
allocatable           :: x
...
x = [ 1, 2, 3 ]
print *, size(x)
x = [ 4, 5 ]
print *, size(x)
```

prints 3

prints 2

Derived Type constructor

```
type person
  real      :: age
  character :: name
  integer   :: ssn
end type person

type(person) :: you

you = [ 17., 'John Doe', 123456789 ]
```

Arrays as Indices

```
real, dimension(5) :: &
  a = [ 1, 3, 5, 7, 9 ]
real, dimension(2) :: &
  i = [ 2, 4 ]

print *, a(i)
```

`prints 3. 7.`

- Variable i is an array (vector)
- a(i) is [a(i(1)), a(i(2)), ...]

where statement

```
real, dimension(4) :: &
  x = [ -1, 0, 1, 2 ] &
  a = [  5, 6, 7, 8 ]
...
where (x < 0)
  a = -1.
end where

where (x /= 0)
  a = 1. / a
elsewhere
  a = 0.
end where
```

- arrays must have the same shape
- code block executes when condition is true
- code block can contain
  - Array assignments
  - other where constructs
  - forall constructs

where statement

```
real                          :: v
real, dimension(100,100)      :: x
...
call random_number(v) ! scalar
call random_number(x) ! array
where (x < 0.5)
  x = 0.
end where
```

- Distinction between scalar and array vanishes
  call to `random_number()`
- Subroutine random_number accepts scalars and arrays
- see also slides on elemental functions

any statement

```
integer, parameter   :: n = 100
real, dimension(n,n) :: a, b, c1, c2

c1 = my_matmul(a, b) ! home-grown function
c2 = matmul(a, b)    ! built-in function
if (any(abs(c1 - c2) > 1.e-4)) then
  print *, 'There are significant differences'
endif
```

- matmul (also dot_product) is provided by the compiler
- abs(c1 - c2): Array syntax
- any returns one logical

Example: Stencil Update $A_i = (A_{i-1} + A_{i+1})/2.$

```
real, dimension(n) :: v
real               :: t1, t2
...
t2 = v(1)
do i=2, n-1
  t1   = v(i)
  v(i) = 0.5 * (t2 + v(i+1))
  t2   = t1
enddo
```

Example: Stencil Update $A_i = (A_{i-1} + A_{i+1})/2.$

```
real, dimension(n) :: v
real               :: t1, t2
...
t2 = v(1)
do i=2, n-1
  t1   = v(i)
  v(i) = 0.5 * (t2 + v(i+1))
  t2   = t1
enddo

v(2:n-1) = 0.5 * (v(1:n-2) + v(3:n))
```

- Traditional scheme requires scalar variables
- Array syntax: Evaluate RHS, then "copy" the result

Stencil Update

$$A_{i,j} = (A_{i-1,j} + A_{i+1,j} + A_{i,j-1} + A_{i,j+1})/4.$$

```
real, dimension(n,n) :: a, b
do j=2, n-1
  do i=2, n-1
    b(i,j) = 0.25 *
             (a(i-1,j) + a(i+1,j) + a(i,j-1) + a(i,j+1))
  enddo
enddo

do j=2, n-1
  do i=2, n-1
    a(i,j) = b(i,j)
  enddo
enddo
```

- Two copies required: b = f(a); a = b

Stencil Update

$$A_{i,j} = (A_{i-1,j} + A_{i+1,j} + A_{i,j-1} + A_{i,j+1})/4.$$

```
a(2:n-1,2:n-1) = 0.25 *
  (a(1:n-2,2:n) + a(3:n,2:n) + a(2:n,1:n-2) + a(2:n,3:n))
```

- No copy required (done internally)

Stencil Update

$$A_{\mathrm{i,j}} = (A_{\mathrm{i-1,j}} + A_{\mathrm{i+1,j}} + A_{\mathrm{i,j-1}} + A_{\mathrm{i,j+1}})/4.$$

```
a(2:n-1,2:n-1) = 0.25 *
  (a(1:n-2,2:n) + a(3:n,2:n) + a(2:n,1:n-2) + a(2:n,3:n))
```

- No copy required (done internally)

Now with the forall construct

```
forall (i=2:n-1, j=2:n-1) &
a(i,j) = 0.25 *
  (a(i-1,j) + a(i+1,j) + a(i,j-1) + a(i,j+1))
```

- Fortran statement looks exactly like the original formula

Detached Explicit Interfaces

- Enables User-defined Operators and Generic Subprograms
- The interface can be detached from the routine
- Only the interface may reside in the module (like in a C header file)
- Comes in handy, when a large number of people (n>1) work on one project

```
module my_interfaces
interface
  subroutine swap(a, b)
  real, dimension(:)       :: a, b
  real, dimension(size(a)) :: work  ! Scratch array
  end subroutine
end interface
```

- Any subprogram that calls swap has to use the module my_interfaces

Generic Interfaces — Function/Subroutine Overload

Motivation: Write code that allows to swap two variables of type `real` and two variables of type `integer`

- Subroutine 1: swap_real()
- Subroutine 2: swap_integer()

```
module mod_swap
contains
subroutine swap_real(x, y)
real :: x, y, t
t = x; x = y; y = t
end subroutine

subroutine swap_integer(i, j)
real :: i, j, k
k = i; i = j; j = k
end subroutine
end module
```

```
program p_swap
use mod_swap
real    :: a, b
integer :: i1, i2

! Get a, b, i1 and i2 from
!   somewhere
call swap_real(a, b)
call swap_integer(i1, i2)

end program
```

Generic Interfaces — Function/Subroutine Overload

- Add a generic interface (swap) to both routines
- swap with real arguments → swap_real
- swap with integer arguments → swap_integer

```
module mod_swap
public  swap
private swap_real, swap_integer

interface swap
  module procedure &
    swap_real, swap_integer
end interface

contains
```

```
subroutine swap_real(x, y)
real :: x, y, t
t = x; x = y; y = t
end subroutine

subroutine swap_integer(i, j)
real :: i, j, k
k = i; i = j; j = k
end subroutine
end module
```

Generic Interfaces — Function/Subroutine Overload

```
module mod_swap
public  swap
private swap_real, swap_integer

interface swap
  module procedure &
    swap_real, swap_integer
end interface

contains
...
```

```
program p_swap
use mod_swap
call swap(a, b)    ! swap_real
call swap(i1, i2) ! swap_integer
call swap_real(a, b) ! Does NOT
                     ! compile!
end program
```

- Interface swap is public
- Inner workings (swap_real, swap_integer) are private
- User of module mod_swap cannot access/mess-up "inner" routines

Generic Interfaces — Function/Subroutine Overload

- Anything distinguishable works
- `real, integer, real(8), ...`
- Only one argument may differ

```
module mod_swap
public  swap
private swap_real, swap_real8

interface swap
  module procedure &
    swap_real, swap_real8
end interface

contains
```

```
subroutine swap_real(x, y)
real :: x, y, t
t = x; x = y; y = t
end subroutine

subroutine swap_real8(x, y)
real(8) :: x, y, t
t = x; x = y; y = t
end subroutine
end module
```

User-defined Operators

```
module operator
public  :: operator(.lpl.)
private :: log_plus_log
  interface operator(.lpl.)
  module procedure log_plus_log
  end interface
contains
  function log_plus_log(x, y) &
    result(lpl_result)
  real, intent(in)  :: x, y
  real              :: lpl_result
  lpl_result = log(x) + log(y)
  end function
end module
```

```
program op
use operator
print *, 2. .lpl. 3.
end program
```

- prints 1.791759
- .lpl. is the new operator (defined public)
- rest of the definition is private
  - interface
  - function log_plus_log
- .lpl. is defined as
  log(x) + log(y)
- log(2.) + log(3.) = 1.791759

Elemental Functions

```
module e_fct
elemental function sqr(x) &
  result(sqr_result)
real, intent(in) :: x
real             :: sqr_result
sqr_result = x * x
end function
end module
```

```
program example
use e_fct
real                 :: x = 1.5
real, dimension(2) :: a = &
                    [ 2.5, 3.5 ]
print *, 'x = ', sqr(x)
print *, 'a = ', sqr(a)
end program
```

- Write function for scalars
- add elemental
- routine will also accept arrays

- prints a = 2.25
- prints x = 6.25 12.25
- allows to extend array syntax to more operations

where/any in combination with elemental functions

```
module e_fct
elemental function log_sqr(x) &
  result(ls_result)
real, intent(in) :: x
real             :: ls_result
ls_result = log(sqr(x))
end function
end module
```

- Put an elemental function in a module

```
subroutine example(x, y)
use e_fct
real, dimension(100) :: x, y
where (log_sqr(x) < 0.5)
  y = x * x
end where
if (any(log_sqr(x) > 10.)) then
  print *, '... something ...'
endif
end program
```

- Use elemental function with where and any

Inquiry Functions

- **digits(x):** numer of significant digits
- **epsilon(x):** smallest ϵ with $1 + \epsilon <> 1$
- **huge(x):** largest number
- **maxexponent/minexponent:** largest/smallest exponent
- **tiny(x):** smallest positive number (that is not 0.)
- ubound, lbound, size, shape, ...
- input_unit, output_unit, error_unit
- file_storage_size (Good when you use the Intel compiler!)
- character_storage_size, numeric_storage_size
- etc.

Mathematical Functions

- sin, cos, tan, etc.
- New in Fortran 2008: Bessel fct., Error-fct., Gamma-fct., etc.

Fortran pointers (Aliases)

```
integer, parameter              :: n = 1000
real, dimension(n*n), target :: data
real, dimension(:), pointer  :: ptr, diag
real, dimension(:), allocatable, &
                    pointer  :: ptr_alloc
...
ptr  => data
diag => data(1: :1001) ! start, end, stride
allocate(ptr_alloc(100))
```

- Pointer asscociation : “Pointing to”
- Pointer is of the same type as the target
- Target has the target attribute (needed for optimization)
- Pointers can have memory allocated by themselves (ptr_alloc in C)
- Pointers are usefull to create “linked lists” (not covered here)

Fortran pointers (Aliases)

```
integer, parameter               :: n = 5
real, dimension(n,n), target :: data
real, dimension(:), pointer  :: row, col
...
row => data(4,:)     ! 4th row
col => data(:,2)     ! 2nd column
print *, row, col    ! Use pointer like a variable
```

- Pointers col and row are pointing to a colum/row of the 2-dim array data
- Memory is not contigous for row
- When you pass row to a subroutine, a copy-in/copy-out may be necessary
- What is '=>' good for? Referencing and de-referencing is automatic, so a special symbol is needed for pointing

Fortran pointers (Aliases)

```
real, dimension(100), target :: array1, array2, temp
real, dimension(:), pointer  :: p1,     p2,     ptmp
...
temp   = array1      ! Copy the whole array 3 times
array1 = array2      ! Very costly!
array2 = temp
...
ptmp => p1           ! Move the Pointers
p1   => p2           ! Very cheap!
p2   => ptmp
```

- Avoid copying data
- Switch the pointers
- Use the pointers as of they were normal variables

Command Line Arguments

```
command_argument_count()  ! Function: returns
                          !   number of arguments
call get_command argument(number, value, length, status)
                          ! input:  number
                          ! output: value, length, status
                          !         (all optional)
call get_command(command, length, status)
                          ! output: command, length, status
Example:
./a.out option X
character(len=16) :: command
call get_command(command)
print command             !  prints: ./a.out option X
```

Environment Variables

```
call get_environment_variable(name, value)
                              ! Input :  name
                              ! Output: value

character(len=16) :: value
call get_environment_variable('SHELL', value)
print value                   !  prints /bin/bash
```

Fortran Prepocessor

- same as in C (#ifdef, #ifndef, #else, #endif)
- compile with -fpp
- use option -D<variable> to set variable to true
- Example: ifort -Dmacro t.f

```
#ifdef macro
  x = y
#else
  x = z
#endif
```

Interoperability with C (Name Mangling)

- Variables, Functions and Subroutines, etc., that appear in modules have mangled names
- This enables hiding the components from misuse
- No naming convention for the mangled names

```
file t.f
module operator
  real :: x
contains
  subroutine s()
  return
  end subroutine
end
```

```
compile with:
ifort -c t.f
result is t.o

nm t.o prints this:
  (nm is a Unix command)
T _operator_mp_s_
C operator_mp_x_
```

Give Objects (in object file) a specific Name

- Use intrinsic module (iso_c_binding) to use pass strings (not shown here)

```
file t.f
module operator
  real, bind(C) :: x
contains
  subroutine s() &
    bind(C, name='_s')
  return
  end subroutine
end
```

```
compile with:
ifort -c t.f
result is t.o

nm t.o prints this:
T _s
C _x
```

Use C-compatible variable types

- Use variables of a special kind
- c_float, c_double, c_int, c_ptr, etc.
- works with characters, too

```
module operator
  real, bind(C) :: x

  type, bind(C) :: c_comp
    real(c_float)  :: data
    integer(c_int) :: i
    type(c_ptr)    :: ptr
  end type
contains
  subroutine s() &
    bind(C, name='_s')
```

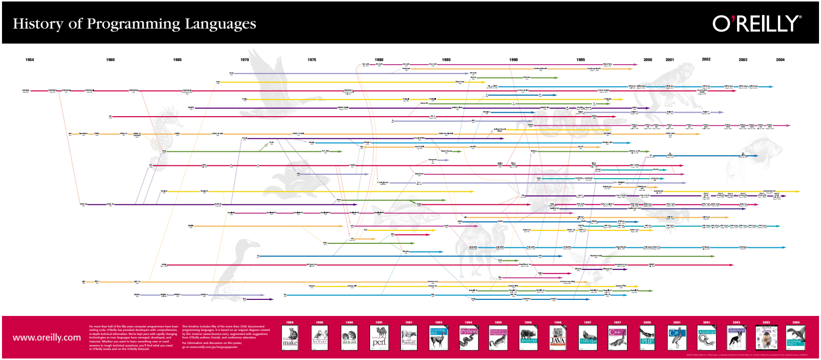
```
Arrays:
‘‘Fortran’’:
real(c_float) :: x(5,6,7)

‘‘C’’:
float y[7][6][5]
```

Not Covered

- Floating-point Exception Handling
- Linked-Lists, Binary Trees
- Recursion
- I/O (Stream Data Access)
- Object-Oriented Programming, but see introduction in the next chapter

History of Fortran



Fortran started in 1954; the first "line" in the diagram.

Fortran 90+: 90, 95, 2003, 2008

- Modern, efficient, and appropriate for Number Crunching and High Performance Computing
- Upgrades every few years: 90, 95, 2003, 2008, ...
- Major upgrade every other release: 90, 2003
- Easy switch: F90 is fully compatible with F77

Where are we now?

- F2003 fully supported by Cray, IBM, PGI and Intel compilers
- F2008 is partially supported

Performance Considerations and Object-Oriented Programming

- (Most of the) Language elements shown in this class do not have (any/severe) performance implications
  - Most of the module-oriented programming handles access
  - Some array syntax may! be done better in explicit loops, if more than one statement can be grouped into one loop
  - Pointers that have *non-contigous* elements in memory may! require a copy in/out, when passed to a subprogram
  - Compiler can warn you (Intel: -check arg_temp_created)
  - Use pointers (references) and]em non-contigous data with care
- Fortran allows for an Object-Oriented Programming style
  - Access control, really a great concept!
  - Type extension, Polymorphic entities
  - Use with care (may be slower),
  - but use these features if you algorithm requires and the implemenation benefits from it

Functions, Modules, Objects

- Use Functions and Subroutines to hide local Data
- Use Modules to hide Data, Functions and Subroutines
- Use Objects to hide Data and expose Methods

Book Recommendations

- **Fortran 95/2003 for Scientists and Engineers** by Chapman
  Very! verbose, with many examples. Guides the programmer nicely towards a good programming style. (International/cheaper edition available)
- **modern fortran explained** by Metcalf, Reid and Cohen
  Good to learn new features; more complete than the Guide (1), but sometimes a bit confusing. Covers Fortran 2008
- **Guide to Fortran 2003 Programming** by Walter S. Brainerd
  Good to learn the new features, clever examples
- **The Fortran 2003 Handbook** by Adams, Brainerd, et al.
  Complete syntax and Reference

Some Guidance is definitely needed

- The same task may be accomplished in several ways
- What to use When?

OO Concept in 1 Slide

- Objects contain (have the properties):

Data	— Instance Variables or Fields
Subr./Fct.	— Instance Methods
Polymorphism/Inheritance	— to allow for a lot of flexibility

- Data is only accessible through the methods
- OO-speak: Call of a Subr. (instance method) $\equiv$ Sending a Message
- A `Class` is a blueprint for a `Object`
  Similar to a `Derived Type` being a blueprint for a structure

```
type(data)            :: structure_containing_variables
class(data_plus_fct) :: object_containing_variables_and_functions
```

- Classes are organized in Hierarchies and can inherit instance variables and methods from higher level classes
- An object can have many forms (polymorphism), depending on context

Example of an Object in Fortran2003

```
module my_mod
type, public :: person
character(len=8), private :: &
                             name
integer, private          :: &
                             iage
contains
  procedure, public :: set
  procedure, public :: out
end type person

private; contains
```

- Interface is Public
- Subroutines are Private

```
subroutine set(p, name, iage)
class(person) :: p
character(len=*) :: name
integer          :: iage
p%name = name
p%iage = iage
write (0,*) 'set'
end subroutine

subroutine out(p)
class(person) :: p
write (0,*) p%name, p%iage
end subroutine

end module
```

How to use the Class defined in my_mod: Non-polymorphic object

```
program op
use my_mod

! Non-polymorphic
type(person), allocatable :: x
type(person), pointer     :: y

allocate(x, y)

call x%set('J. Doe', 25)
call x%out          ! or call y%out
end
```

- Declare object as a type
- Non-polymorphic: No performance penalty
- Access to the data only through approved methods
- Object may be a pointer

Note:
x%set called with 2 arguments,
but Subroutine has 3 arguments

How to use the Class defined in my_mod: Polymorphic object

```
program op
use my_mod

! Polymorphic
class(person), pointer    :: z

allocate(z)

call z%set('J. Doe', 25)
call z%out
end
```

- Declare object as a class
- Polymorphic: full OO functionality
- Object must be a pointer

Note:
z%set called with 2 arguments, but Subroutine has 3 arguments