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

March 5, 2012
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
they enable us to repeat operations on different data
and to avoid 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
  \[\Rightarrow\] **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 concepts are 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.

- parallel computing: MPI, OpenMP, CUDA, ...
- ... and the flaws → 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
  - Explicit 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
- Multiple statements in one line separated by a semicolon (;)

Style example

```fortran
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**

```fortran
program square

! This program calculates ...

implicit none
real :: x, x2

x = 5.
x2 = x * x
if (x == 13.) print *, ’Lucky’
end
```

**Bad**

```fortran
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
endo
d

Bad

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

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

- **while-Loop**
  ```fortran
  i = 0
do
  if (i > 20) exit
  i = i + 1
endo
  ```

- **repeat-Loop**
  ```fortran
  i = 0
do
  i = i + 1
  if (i > 20) exit
endo
  ```

• Use the `exit` statement to “jump” out of a loop
Loops: **exit** and **cycle**

- **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

**Exit anywhere**

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

**Skip a loop iteration**

```fortran
do i=1, 100
    x = real(i)
    y = sin(x)
    if (i > 20) cycle
    z = cos(x)
enddo
```
Nested loops: **exit** and **cycle**

**Exit Outer Loop**

```fortran
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**

```fortran
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

```fortran
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
```

---

Modern Programming Languages: Fortran90/95/2003/2008
Modern Fortran Style
Variables of different kind values

```fortran
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

\[
\begin{align*}
\text{real}*8 &: \ x8 & \text{! Real with 8 bytes (double precision)} \\
\text{real}(\text{kind}=8) &: \ y8 & \text{! same, but not completely safe} \\
\text{real}*4 &: \ x4 & \text{! Real with 4 bytes (single precision)} \\
\text{integer}*4 &: \ i4 & \text{! Integer single precision} \\
\text{integer}*8 &: \ i8 & \text{! Integer double precision} \\
\end{align*}
\]

\[
\begin{align*}
x8 &= 3.1415_8 & \text{! Literal constant in double precision} \\
i8 &= 6_8 & \text{! same for an integer}
\end{align*}
\]

- \text{real}*8, \text{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

```fortran
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.

```fortran
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

```fortran
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

```fortran
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

```fortran
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 ⇒ 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

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

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

```fortran
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

```fortran
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

```fortran
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

- Subprograms after the contains statement

```fortran
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 subroutine
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
end program go_mad
```
Example: Public, Private Subroutine

```fortran
module mad_science
private
public :: set_mad
contains

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

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

- 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

```fortran
module mad_science
private
public :: swap
real, dimension(100) :: scratch
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)
end subroutine swap
end module mad_science
```

- 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

```fortran
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))

end subroutine alloc

end module mad_science
```

- **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 be changed 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 $\implies$ Explicit

- Implicit interface: matching positions

```fortran
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 $\Rightarrow$ Explicit

- Explicit interface which does not solely rely on positional information

```fortran
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
  - Assumed-shape arrays, optional parameters, etc.
Passing an array

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

```fortran
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 explicitly 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

```fortran
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
```

- 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)

```fortran
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

```fortran
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
end subroutine calc

```

```fortran
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

```fortran
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
end subroutine calc
```

```fortran
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 bare 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 your 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, unambiguous 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 organize your data (3 slides)
  ⇒ 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 Translation

• Array syntax
• \texttt{where} construct
• \texttt{forall} construct
• Case study: Stencil Update
• User defined Operators
• Elemental Functions
• Inquiry Functions
• Odds and Ends
Simple Array Syntax

- 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

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

a = b
b = d
a(1:10) = b(1:10)
a(2:3) = b(4:5)
a(1:10) = c(1:10,2)
a = x
b = x
a(1:3) = b(1:5:2) ! a(1) = b(1)
    ! a(2) = b(3)
    ! a(3) = b(5)
```
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(len=8) :: name
  integer :: ssn
end type person

type(person) :: you

you = [ 17., 'John Doe', 123456789 ]
Arrays as Indices

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

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

print *, a(i)
```

prints 3. 7.
**where statement**

```fortran
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 \[ A_i = (A_{i-1} + A_{i+1})/2. \]

```fortran
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
endo
```

• Traditional scheme requires scalar variables
• Array syntax: Evaluate RHS, then “copy” the result
Example: Stencil Update

\[ A_i = \frac{(A_{i-1} + A_{i+1})}{2}. \]

```fortran
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
endo

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

\[ A_{i,j} = \frac{(A_{i-1,j} + A_{i+1,j} + A_{i,j-1} + A_{i,j+1})}{4}. \]

```fortran
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

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

\[
a(2:n-1, 2:n-1) = 0.25 \times (a(1:n-2, 2:n) + a(3:n, 2:n) + a(2:n, 1:n-2) + a(2:n, 3:n))
\]

- No copy required (done internally)
Stencil Update  \[ A_{i,j} = \frac{(A_{i-1,j} + A_{i+1,j} + A_{i,j-1} + A_{i,j+1})}{4}. \]

\[
a(2:n-1,2:n-1) = 0.25 \times (a(1:n-2,2:n) + a(3:n,2:n) + a(2:n,1:n-2) + a(2:n,3:n))
\]

- No copy required (done internally)

Now with the \texttt{forall} construct

\[
\texttt{forall } (i=2:n-1, j=2:n-1) \&
\texttt{a(i,j) = 0.25 } \times \\
(a(i-1,j) + a(i+1,j) + a(i,j-1) + a(i,j+1))
\]

- 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

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

- 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()`

```fortran
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

```fortran
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**

```fortran
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
```

```fortran
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
```

- Put an **elemental** function in a module
- Use elemental function with **where** and **any**
Inquiry Functions

- **digits(x):** number of significant digits
- **epsilon(x):** smallest $\epsilon$ with $1 + \epsilon \neq 1$
- **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)

```fortran
integer, parameter :: n = 1000
default real, dimension(n*n), target :: data
default real, dimension(:,), pointer :: ptr, diag
default real, dimension(:,), allocatable, &
  pointer :: ptr_alloc

... ptr => data diag => data(1: :1001) ! start, end, stride
allocate(ptr_alloc(100))
```

- Pointer association: "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 useful to create "linked lists" (not covered here)
Fortran pointers (Aliases)

```fortran
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 column/row of the 2-dim array `data`
- Memory is not contiguous for `row`
- When you pass `row` to a subroutine, a copy-in/copy-out may be necessary
- What is `=>` used for? Referencing and de-referencing is automatic, so a special symbol is needed for pointing
Fortran pointers (Aliases)

```fortran
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
...
p1 => array1 ! use 2 pointers to point
p2 => array2 ! to data
...
ptmp => p1 ! Move the Pointers
p1 => p2 ! Very cheap!
p2 => ptmp
```

• Later, 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 Preprocessor

• 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

```fortran
#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

```fortran
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:
```
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)

```fortran
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
Fortran started in 1954; the first “line” in the diagram.

• 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-contiguous 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 non-contiguous 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 your algorithm requires and the implementation 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 (4), but a very few times a bit confusing. Covers Fortran 2008

  Complete syntax and Reference

- **Guide to Fortran 2003 Programming** by Walter S. Brainerd
  Good to learn the new features, clever examples

  **Some Guidance is definitely needed**

- The same task may be accomplished in several ways
- What to use When?
How long does it take to Debug Code?

• Defect density $r$ per 1000 lines
• Number of lines in code: $\lambda$
• Assume a chronological listing of code, where redundant occurrences are removed
• A bug appears “on average” at line number $\lambda/2$ (in the middle)
• $\rightarrow$ Bug must be located in preceding lines $\lambda/2 - 1$, again “on average” at line number $\frac{1}{2}(\lambda/2 - 1)$ (again, in the middle)
• Code contains $r \times \lambda$ bugs
• $\rightarrow (r\lambda) \frac{1}{2}(\lambda/2 - 1) \approx \lambda^2$ lines have to be inspected
• Defect rate $\lambda$ is independent of Modularity (8-24 per 1000 lines, according to studies)
• “Something” better than Modularity is needed
From Modules to Objects: **Global Variables**

In debugging, most time is spent **locating the bug**, not fixing it

- Let’s assume that your code iterates to find a solution and that the problem occurs in the 2\textsuperscript{nd} iteration
- Almost all your code has been executed at least once, probably except the routines for the final output

If your code employs:

**Global Variables**

- the bug can be anywhere in your $\lambda$ lines of code
- in any routine you may have overwritten a global variable
- the debug-time will be proportional to $\lambda^2$ for all bugs
From Modules to Objects: Modules

• Let’s assume that you have created several modules, and that you have bundled subroutines needed for a specific task in these modules.

• Example:
  – Module “read_data” contains routines that read input into array \( inp \)
  – Module “calc_1” takes \( inp \) and calculates some temporary data \( d1 \)
  – Module “calc_2” takes \( inp \) and \( d1 \) and calculates the result \( r \)

If Bugs appear in Module “calc_2”

• Bugs may be located in the same module and the search time is then proportional to \( \lambda_{calc_2}^2 \)

• but the bugs may come from changes you made in the other 2 modules

• best case: search time proportional to \( \lambda_{read_data}^2 + \lambda_{calc_1}^2 + \lambda_{calc_2}^2 \)

• worst case w/ side effects: search time again proportional to \( \lambda^2 \)
From Modules to Objects:

• Modules (Modularity) create a barrier that is somewhat porous
• Let’s make the barrier (almost) impenetrable:
  → Objects and Object-Oriented programming
• Content in one Object is completely shielded from content in another Object

  Bugs appear in Object “calc_2”
  → your search time is proportional to $\lambda_{\text{calc}_2}^2$

What are Objects and what do they contain?
Object-Oriented Programming: Concept

- Object-oriented code will still contain data (arrays, structures) and subprograms (subroutines, functions), but we need new names for these if they reside in an object.

Objects add functions/subroutines to a Derived Type

- The data in a Derived Type is called a Field
- The subprograms are called Methods
- Methods operate on the Fields
- Only the Methods are visible from the outside
- The data (fields) are not visible
- An Object is an instance of a Class (or a Derived Type); Examples:
  
  ```fortran
  real :: a
  ! a is an instantiation of real
  type(class_a) :: object_a
  ! object_a is an instantiation of the class_a
  ```
Object-Oriented Programming: Nomenclature

- All languages use their own “words”
- There is usually no 1-to-1 translation between Fortran and C++, but the table provides a guideline

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C++</th>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derived type / Class</td>
<td>Class</td>
<td>Abstract data type</td>
</tr>
<tr>
<td>Component</td>
<td>Data member</td>
<td>Attribute or Field</td>
</tr>
<tr>
<td>Class</td>
<td>Dynamic Polymorphism</td>
<td></td>
</tr>
<tr>
<td>Type-bound procedure</td>
<td>Virtual Member function</td>
<td>Method</td>
</tr>
<tr>
<td>Parent type</td>
<td>Base class</td>
<td>Parent class</td>
</tr>
<tr>
<td>Extended type</td>
<td>Subclass</td>
<td>Child class</td>
</tr>
<tr>
<td>Module</td>
<td>Namespace</td>
<td>Package</td>
</tr>
</tbody>
</table>

from "Scientific Software Design" by Rouson, Xia, Xu, Table 2.1
Object-Oriented Programming: Advantages

- Data in a class is shielded / hidden
- Methods are exposed
- Code becomes reusable: Never replicate code!
- Code becomes extensible: Inheritance
- Debug (search) time “scales” with number of lines in code in object

The three (four) pillars of Object-Oriented Programming (OOP)

1. Data encapsulation & data hiding
2. Polymorphism
3. Inheritance
4. Operator overloading

Polymorphism comes in 2 flavors; I’ll only touch on one, which is sometimes called “static” or “procedural” polymorphism
Summary of OO Concept

- Objects contain (have the properties):
  - Data — Instance Variables or Fields
  - Subr./Fct. — Instance Methods
  - Polymorphism/Inheritance — Reusability and Extensibility

- Data is only accessible through the methods
- OO-speak: Call of a Subr. (instance method) ≡ Sending a Message
- A **Class** is a blueprint for a **Object**

\[
\begin{align*}
type(data) & : \text{structure}_\text{containing}_\text{variables} \\
type(data\_plus\_fct) & : \text{object}_\text{containing}_\text{variables}\text{\_and\_functions} \\
\text{class}(data\_plus\_fct) & : with\ \text{dynamic\ polymorphism}
\end{align*}
\]

- 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 Fortran 2003

Module my_mod
private
public :: person

type :: person
class(person) :: this
character(len=*) :: name
integer :: iage
contains
  procedure :: set
  procedure :: print
end type person

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

procedure :: print
  class(person) :: this
  write (0,*), this%name, &
  this%iage
end subroutine

end module
How to use the Class defined in my_mod

program op
use my_mod

type(person) :: you

type(person), dimension(5) :: we

! Allowed
call you%set(’J. Doe’, 25)
call you%print

! Not allowed
call set(...)  
call print(...)  
write (0,*) ’name is ’, you%name
we(1) = you
end

• Declare object as a type without dynamic polymorphism: No performance implications
• Access to the data only through approved public methods

Note:
you%set called with 2 arguments, but Subroutine has 3 arguments
How is a method called?

! Definition (with 3 arguments)
subroutine set(this, &
               name, iage)
class(person) :: this
character(len=*) :: name
integer :: iage
this%name = name
this%iage = iage
write (0,*) 'set'
end subroutine

! Call side (with 2 arguments)
call you%set(’J. Doe’, 25)

• you%set is called with 2 arguments, but Subroutine has 3 arguments
• The you on the call side, becomes the first argument (this) on the definition side
Encapsulation and Code Replication

- All data in your objects is hidden
- Only the methods are exposed
- The definition of all data and all methods are in one module
- Code is never replicated
- It always means the same when you code: call you%print
- If you change the meaning of you%print, the meaning changes everywhere
- There will be only one method in your code that prints the fields of a particular object, i.e. you%print
Inheritance: Avoid Code Replication

-module mod_line
  private
  public :: line

  type :: line
  real, private :: x1, y1, x2, y2
  contains
    procedure :: print_coord
  end type

  contains

  subroutine print_coord(this)
    class(line) :: this
    write (0,*) 'Start : ', x1, y1
    write (0,*) 'End   : ', x2, y2
  end subroutine

end module

- Module contains the class line
- Data:
  start and end points
- Method:
  Print start and end points
Inheritance: Extend an existing class

Class **color_line** extends class **line**
- **color_line** inherits the data `x1, y1, x2, y2`
- and the method **print_coord**
- Additions: `rgb` and **print_color**

```fortran
module mod_line
  private
  public :: line, color_line

  type :: line
  real, private :: x1, y1, &
  x2, y2

  contains
    procedure :: print_coord
  end type

  type, extends(line) :: &
  color_line
  integer, dimension(3), &
  private :: rgb

  contains
    procedure :: print_color
  end type
```

```fortran
```

- Class **color_line** extends class **line**
- **color_line** inherits the data `x1, y1, x2, y2`
- and the method **print_coord**
- Additions: `rgb` and **print_color**
Inheritance: Extend an existing class

- `l1%print_coord` and `l2%print_coord` invoke the same subroutine
- Object `l2` contains the same data and methods as `l1`
- `l2%print_color` is an additional method unique to object `l2`

```fortran
program main
use mod_line

! Inherited Method
! call l2%print_coord

! Invalid
! call l1%print_color
```

```fortran
  type(line) :: l1
  type(color_line) :: l2

  call l1%print_coord
  call l2%print_color

  ! Inherited Method
  call l2%print_coord

  ! Invalid
  call l1%print_color
```

- `l1%print_coord` and `l2%print_coord` invoke the same subroutine
- Object `l2` contains the same data and methods as `l1`
- `l2%print_color` is an additional method unique to object `l2`