Goals for today

- Search for errors in programs with compiler flags
- Define functions and subroutines
- Combine them together with variables in modules
- Understand the scope of objects

Fortran variable names

- Use meaningful names for variables and constants if possible.
- Names may be up to 63 characters long.
- Each name must begin with a letter.
- Letters, numbers and underscores may be used for the characters that follow.

valid

```
time
z123456789
i_am_hungry
```

invalid

```
a_very_very_very_very_very_very_very_very_very_very_very_long_name
3_days
a$
```

Debugging options

- Detect uninitialized variables
  - gfortran -finit-real=snan deep_thought.f90
  - ifort -init=snan,arrays deep_thought.f90
- Catch floating point errors
  - gfortran -ffpe-trap=invalid,zero,underflow, overflow,denormal,inexact deep_thought.f90
  - ifort -fpe0 deep_thought.f90
- Check array bounds
  - gfortran -fbounds-check deep_thought.f90
  - ifort -check bounds deep_thought.f90
- Show warnings
  - gfortran -Wall deep_thought.f90
  - ifort -warn deep_thought.f90

```fortran
PROGRAM deep_thought
  IMPLICIT NONE
  INTEGER :: i, j, a, b
  REAL :: nothing, everything
  INTEGER, ALLOCATABLE :: solution(:), mystery(:)

  ALLOCATE(mystery(16))
  ALLOCATE(solution(16))

  everything = nothing+1

  solution = 1
  mystery = 0

  DO i = 1, 22
    mystery(i) = i-15
  END DO

  a = solution(1)
  b = solution(2)

  PRINT*, 'The answer is: ', a*b
END PROGRAM deep_thought
```

Procedures

If programs become more complex or individual program sections occur more than once, it makes sense to use procedures.

Fortran knows two types of procedures:

- **Functions** return a single value (directly), and can be called within an expression.
- **Subroutines** return any number of values, and are called with CALL.

“Dummy arguments” are the arguments in the definition of the procedure. “Actual arguments” are the arguments in the calling list, where the procedure is called.

The number and type of dummy and actual arguments must be identical.

Function

```
FUNCTION name (argument list)
  declaration part (including type of name)

  execution part

  name = ...
  RETURN
END FUNCTION name
```

Subroutine

```
SUBROUTINE name (argument list)
  declaration part

  execution part

  RETURN
END SUBROUTINE name
```

Factorial as a function

```
FUNCTION factorial(n)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n
  INTEGER :: i, a, factorial

  a = 1
  DO i = 2, n
    a = a*i
  END DO

  factorial = a

  RETURN
END FUNCTION factorial
```

```
PRINT*, num, '! =', factorial(num)
```

... and subroutine

```
SUBROUTINE factorial(n, a)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n
  INTEGER, INTENT(OUT) :: a
  INTEGER :: i

  a = 1
  DO i = 2, n
    a = a*i
  END DO

  RETURN
END SUBROUTINE factorial
```

dummy arguments

actual arguments

```
CALL factorial(num, fact)
PRINT*, num, '! =', fact
```

Names in the main and procedure can be different. What matters is the order of the arguments.

The INTENT attribute

Input/output arguments of a procedure can be declared with an INTENT attribute. The following INTENT attributes are possible:

`INTENT(IN)`	argument is used to pass input data from the main program to the procedure
`INTENT(OUT)`	argument is used to pass results from the procedure to the main program
`INTENT(INOUT)`	argument is used to pass input data from the main program to the procedure as well as results from the procedure to the main program

→ Always declare the INTENT of all arguments. This helps the compiler to detect errors.

Example

```fortran
SUBROUTINE bad_intent(input,output)
  IMPLICIT NONE
  REAL, INTENT(IN) :: input
  REAL, INTENT(OUT) :: output

  output = 2.*input
  input = -1. ! Doesn't work

  RETURN
END SUBROUTINE bad_intent
```

```fortran
SUBROUTINE good_intent(input,output)
  IMPLICIT NONE
  REAL, INTENT(INOUT) :: input
  REAL, INTENT(OUT) :: output

  output = 2.*input
  input = -1. ! Works

  RETURN
END SUBROUTINE good_intent
```

Gives error when compiling

```
$ gfortran bad_intent.f90
    7 |     input = -1. ! Doesn't work
      |    1
Error: Dummy argument 'input' with INTENT(IN) in variable definition context (assignment) at (1)
```

Arrays as arguments: 3 different possibilities

1. Explicit-shape array

```
SUBROUTINE process1 (data1, data2, nvals)
  INTEGER, INTENT(IN) :: nvals
  REAL, INTENT(IN), DIMENSION(nvals) :: data1 ! Explicit shape
  REAL, INTENT(OUT), DIMENSION(nvals) :: data2 ! Explicit shape
  data2 = 3. * data1
END SUBROUTINE process1
```

2. Assumed-shape array
   - Size does not have to be included as an argument, but the bounds are lost

```
SUBROUTINE process2(data1, data2)
  REAL, INTENT(IN), DIMENSION(:) :: data1 ! Assumed shape
  REAL, INTENT(OUT), DIMENSION(:) :: data2 ! Assumed shape
  data2 = 3. * data1
END SUBROUTINE process2
```

3. Assumed-size array
   - Whole array operations only possible with specification of the bounds
   - Bounds checking not possible
   - Obsolescent (not recommended)

```
SUBROUTINE process3(data1, data2, nvals)
  INTEGER, INTENT(IN) :: nvals
  REAL, INTENT(IN), DIMENSION(*) :: data1 ! Assumed size
  REAL, INTENT(OUT), DIMENSION(*) :: data2 ! Assumed size
  data2(1:nvals) = 3. * data1(1:nvals)
END SUBROUTINE process3
```

Internal and external procedures

- **Internal procedures** are CONTAINed in the main program and can be easily linked by the compiler.

- **External procedures** are located outside the main program (after END PROGRAM or in a separate file).

- External functions have to be declared in the main program, subroutines and internal functions don’t.

Factorial as internal function

```
PROGRAM intern
IMPLICIT NONE
INTEGER :: num=0

DO WHILE (num <= 0)
  PRINT*, 'Please enter a positive number:'
  READ*, num
END DO
PRINT*, num, '! =', factorial(num)

CONTAINS

  FUNCTION factorial(n)
    IMPLICIT NONE
    INTEGER, INTENT(IN) :: n
    INTEGER :: i, a, factorial

    a = 1
    DO i = 2, n
      a = a*i
    END DO
    factorial = a
    RETURN
  END FUNCTION factorial
END PROGRAM intern
```

... and external function

```
PROGRAM extern
IMPLICIT NONE
INTEGER :: num=0
INTEGER :: factorial

DO WHILE (num <= 0)
  PRINT*, 'Please enter a positive number:'
  READ*, num
END DO
PRINT*, num, '! =', factorial(num)
END PROGRAM extern

FUNCTION factorial(n)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n
  INTEGER :: i, a, factorial

  a = 1
  DO i = 2, n
    a = a*i
  END DO
  factorial = a
  RETURN
END FUNCTION factorial
```

Interface blocks for external procedures

For external procedures the compiler does not recognize type incompatibilities

```
PROGRAM bad_call
  IMPLICIT NONE
  REAL :: x = 1.
  CALL bad_argument(x)
END PROGRAM bad_call
```

```
SUBROUTINE bad_argument(i)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: i
  WRITE (*,*) 'i = ', i
END SUBROUTINE bad_argument
```

```
$ gfortran bad_argument.f90 bad_call.f90
$ ./a.out
 i =   1065353216
```

Interface blocks can solve this problem

```
PROGRAM bad_call
  IMPLICIT NONE
  INTERFACE
    SUBROUTINE bad_argument(i)
      IMPLICIT NONE
      INTEGER, INTENT(IN) :: i
    END SUBROUTINE bad_argument
  END INTERFACE
  REAL :: x = 1.
  CALL bad_argument(x)
END PROGRAM bad_call
```

```
$ gfortran bad_argument.f90 bad_call.f90
bad_call.f90:12:23:

   12 |    CALL bad_argument(x)
      |                     1
Error: Type mismatch in argument 'i' at (1); passed REAL(4)
to INTEGER(4)
```

Even better: modules (Fortran ≥90)

- Modules are collections of variables and/or procedures defined outside the main program.
- In the main program they can be imported with USE.
- Modules are the best way to share variables between programs, as well as to define functions and subroutines that are used in multiple places.

```
MODULE name
  variable
  definitions
CONTAINS
  functions
  subroutines
END MODULE name
```

For new programs use modules instead of interface blocks if possible.
Interface blocks can be useful as an interface to procedures written in older versions of Fortran or other programming languages (e.g. C++).

Module examples

```
PROGRAM module_demo
  USE functions
  USE important_things
  IMPLICIT NONE
  INTEGER :: num=0
  REAL    :: distance

  DO WHILE (num <= 0)
    PRINT*, 'Please enter a positive number:'
    READ*, num
  END DO
  PRINT*, num, '! =', factorial(num)

  distance = 2*pi*earth_radius*days_per_year
  PRINT*, 'We travel', distance, &
    'meters per year'

END PROGRAM module_demo
```

Compile everything together

```
$ gfortran important_things.f90 functions.f90 module_demo.f90
$ ./a.out
 Please enter a positive number:
5
           5 ! =         120
 We travel   1.46210222E+10 meters per year
```

Constants only

```
MODULE important_things

  IMPLICIT NONE

  REAL, PARAMETER :: pi = 3.14159265, &
        earth_radius = 6.371e6,       &
        days_per_year = 365.25

END MODULE important_things
```

Functions only

```
MODULE functions

CONTAINS

  FUNCTION factorial(n)
    IMPLICIT NONE
    INTEGER, INTENT(IN) :: n
    INTEGER :: i, a, factorial

    a = 1
    DO i = 2, n
      a = a*i
    END DO
    factorial = a
    RETURN
  END FUNCTION factorial

END MODULE functions
```

Scope of objects

The area in which an object (variable, constant, procedure) is visible is the object's **scope.**

- A global object is visible to all contained procedures.
- An object defined in the scope of another object is a different object, even if the names are identical.
- When objects are imported from a module, they are global objects in the main program.

```
PROGRAM Main
REAL :: X=1., Y=2.
  X is global
  Y is global

CONTAINS

  FUNCTION F1()
  REAL :: Y=3.,Z=4.
    X is visible
    Y is now local
    Z is local

    F1 = ....
  END FUNCTION

END PROGRAM Main
```

```
PROGRAM Main
USE Mod
X = 1.
  Y is visible
  F2 is visible

CONTAINS

  FUNCTION F1()
    X is visible
    Y is visible
   F2 is visible

    F1 = ....
  END FUNCTION

END PROGRAM Main
```

```
MODULE Mod
  REAL :: Y = 2.
CONTAINS

  FUNCTION F2()
    Y is visible
    F2 = ....
  END FUNCTION

END MODULE Mod
```

https://www.inscc.utah.edu/~krueger/6150/Scope_Fortran_90.pdf

Scope of objects

- Always declare variables that are used by a function/subroutine.
- Instead of global variables better use arguments and INTENT attribute.

```
PROGRAM scopetest1
  IMPLICIT NONE
  INTEGER :: x=100, y=30
  PRINT*, func(), x, y
CONTAINS
  INTEGER FUNCTION func()
    IMPLICIT NONE
    y=55                          ←  y is global
    func=x*y
    RETURN
  END FUNCTION func
END PROGRAM scopetest1
```

```
gfortran scopetest1.f90
./a.out
        5500         100          55
```

```
PROGRAM scopetest2
  IMPLICIT NONE
  INTEGER :: x=100, y=30
  PRINT*, func(), x, y
CONTAINS
  INTEGER FUNCTION func()
    IMPLICIT NONE
    INTEGER :: y=55               ←  y is local
    func=x*y
    RETURN
  END FUNCTION func
END PROGRAM scopetest2
```

```
gfortran scopetest2.f90
./a.out
        5500         100          30
```

Other useful module things

- Private and public variables/constants/procedures

```
PUBLIC  :: function1, var1
PRIVATE :: function2, var2
```

← default

← cannot be imported

- Import only certain objects:

```
USE module1, ONLY: var1, var2, function1
```

- Rename imported objects:

```
USE module1, ONLY: local_var1 => module_var1
```

Example

```
PROGRAM use_demo
USE trigonometric_functions, ONLY: sin_d => sin_degree
IMPLICIT NONE
REAL :: angle=90.

PRINT*, sin_d(angle)

END PROGRAM use_demo
```

```
$ gfortran trigonometric_functions.f90 usedemo.f90
$ ./a.out
   1.00000000
```

https://pages.mtu.edu/%7eshene/COURSES/cs201/NOTES/chap06/mod-private.html

```
MODULE  trigonometric_functions
   IMPLICIT NONE
   REAL, PARAMETER :: pi = 3.1415926, degree180 = 180.0
   REAL, PARAMETER :: r2d = degree180/pi
   REAL, PARAMETER :: d2r = pi/degree180
   PRIVATE         :: degree180, r2d, d2r
   PRIVATE         :: radian_to_degree, degree_to_radian
   PUBLIC          :: sin_degree, cos_degree

CONTAINS
   REAL FUNCTION radian_to_degree(radian)
      IMPLICIT  NONE
      REAL, INTENT(IN) :: radian
      radian_to_degree = radian * r2d
   END FUNCTION radian_to_degree

   REAL FUNCTION degree_to_radian(degree)
      IMPLICIT  NONE
      REAL, INTENT(IN) :: degree
      degree_to_radian = degree * d2r
   END FUNCTION degree_to_radian

   REAL FUNCTION sin_degree(x)
      IMPLICIT  NONE
      REAL, INTENT(IN) :: x
      sin_degree = SIN(degree_to_radian(x))
   END FUNCTION sin_degree

   REAL FUNCTION cos_degree(x)
      IMPLICIT  NONE
      REAL, INTENT(IN) :: x
      cos_degree = COS(degree_to_radian(x))
   END FUNCTION cos_degree
END MODULE trigonometric_functions
```

Summary

- Debugging options can help us find errors in the code during development.

- Procedures (functions and subroutines) are useful for instructions that occur more than once.

- Modules are collections of variables and/or procedures defined outside of the main program.

- The scope of an object describes the area in which it is visible.