ETH zürich

Computing with Fortran

Engineering Tool V
Spring 2015

Dr. Andreas Haselbacher

Institute of Energy Technology

PREC
Professorship of Renewable Energy Carriers

Overview

- Course overview
- Structure of simple Fortran program
- Compiling and linking
- Intrinsic data types
- Operations
- Intrinsic procedures
- Control flow
- Input/output

Course Overview

- Course is about computing, not programming with Fortran
- Programming is means to an end, not the end itself
- Therefore, need to be effective and efficient programmer

Course Overview

- Computers are fast but stupid: Once told what to do, they can do it quickly many times
- Fundamentally, programming is nothing but explaining actions to computer
- Consequences:
  - You need to understand action before you can explain it
  - Programming forces you to understand what you do
  - If computer does not do what you want, it's your fault

Course Overview

- Course focus on Fortran 90 (called Fortran for simplicity)
- Changes in later versions (mostly) not important for us
  - Fortran 95: Minor revision of Fortran 90
  - Fortran 2003: Major additions to Fortran 95
  - Fortran 2008: Minor revision of Fortran 2003
- `gfortran` compiler:
  - Fortran 95: Completely supported
  - Fortran 2003: Partially supported
  - Fortran 2008: Partially supported

Course Overview: Scope & Structure

- Cannot cover all of Fortran in 12 hours
- Course will only cover what I view as most important parts
- Cannot cover debuggers, unfortunately

- Planned structure:

	Day 1	Day 2	Day 3
Lecture 1	Lecture/ programming	Lecture/ programming	Programming
Lecture 2	Lecture/ programming	Lecture/ programming	Programming
Lecture 3	Lecture/ programming	Lecture/ programming	Programming
Lecture 4	Programming	Programming	Programming

Course Overview: Goals

- Students know important Fortran commands and concepts
- Students write well-structured Fortran programs
- Students use compiler effectively

- Students have fun programming!

Course Overview: Administrative Matters

- Course grade: Pass/fail
- Determination of grade based on program submitted after end of lectures on day 3

Course Overview: Recommended Books

- Fortran:
  - S. J. Chapman, *Fortran 95/2003 for Scientists and Engineers*, 3rd ed., McGraw-Hill, 2008
  - T. M. R. Ellis, I. R. Philips, and T. M. Lahey, *Fortran 90 Programming*, Addison-Wesley, 1994
  - M. Metcalf, J. Reid, and M. Cohen, *Fortran 95/2003 Explained*, 3rd ed., 2004
- Programming in general:
  - B. W. Kernighan and R. Pike, *The Practice of Programming*, Addison-Wesley, 1999
  - S. McConnell, *Code Complete*, 2nd ed., Microsoft Press, 2004

Overview

- Course overview
- Structure of simple Fortran program
- Compiling and linking
- Intrinsic data types
- Operations
- Intrinsic procedures
- Control flow
- Input/output

Structure of Simple Fortran Program

```
PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i,j

i = 1
j = 2*i

WRITE(*,*) i,j

END PROGRAM simple_program
```

Structure of Simple Fortran Program

```
PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i,j
```

← Declaration section

```
i = 1
j = 2*i

WRITE(*,*) i,j
```

← Executable section

```
END PROGRAM simple_program
```

← Termination section

Structure of Simple Fortran Program

```
PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i,j

i = 1
j = 2*i

WRITE(*,*) i,j

END PROGRAM simple_program
```

Program name can consist of up to 31 letters, digits, and underscores, that is, A-Z, a-z, 0-9, _

Program name must start with letter

Fortran is case-insensitive! So

```
simple_program
SIMPLE_PROGRAM
sIMplE_pRoGRaM
```

are all equivalent

Structure of Simple Fortran Program

```
PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i,j

i = 1
j = 2*i

WRITE(*,*) i,j

END PROGRAM simple_program
```

Don’t worry about this for now, we’ll deal with it later

Structure of Simple Fortran Program

```
PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i,j

i = 1
j = 2*i

WRITE(*,*) i,j

END PROGRAM simple_program
```

Fortran commands, variables, and constants are also case-insensitive!

Could just as well have written

```
integer :: I,J
InTEgEr :: I,j
```

Recommend following convention:

- Fortran commands: uppercase
- Program names: mixed case
- Variable names: mixed case

Variable names must not coincide with Fortran commands and procedures

Structure of Simple Fortran Program

```
PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i,j

i = 1
j = 2*i

WRITE(*,*) i,j

END PROGRAM simple_program
```

Lines may be up to 132 characters long

Recommendation: Restrict to smaller number (say, 80) which can be printed on A4 paper with normal font size

If line is longer than your maximum length, use continuation character `&` and continue on next line

The maximum number of continuation lines is 39

Structure of Simple Fortran Program

```
PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i,j

i = 1
j = 2*i

WRITE(*,*) i,j

END PROGRAM simple_program
```

Fortran ignores empty lines

Recommendation: Use them to make your programs more readable

Structure of Simple Fortran Program

```
PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i,j

i = 1
j = 2*i

WRITE(*,*) i,j

END PROGRAM simple_program
```

```
program simple_program
implicit none
integer :: i,j
i = 1
j = 2*i
write(*,*) i,j
end program simple_program
```

Structure of Simple Fortran Program

```
PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i,j

i = 1
j = 2*i

WRITE(*,*) i,j

END PROGRAM simple_program
```

In a simple program like this one, comments are not really necessary

But in more complicated ones, comments are very important

In Fortran, comments start with !

Structure of Simple Fortran Program

```
PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i,j

! Define i
i = 1

j = 2*i ! Compute j

WRITE(*,*) i,j

END PROGRAM simple_program
```

In a simple program like this one, comments are not really necessary

But in more complicated ones, comments are very important

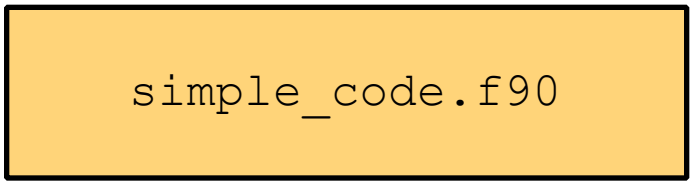
In Fortran, comments start with !

Comments can be placed anywhere, even on same line as statements

Overview

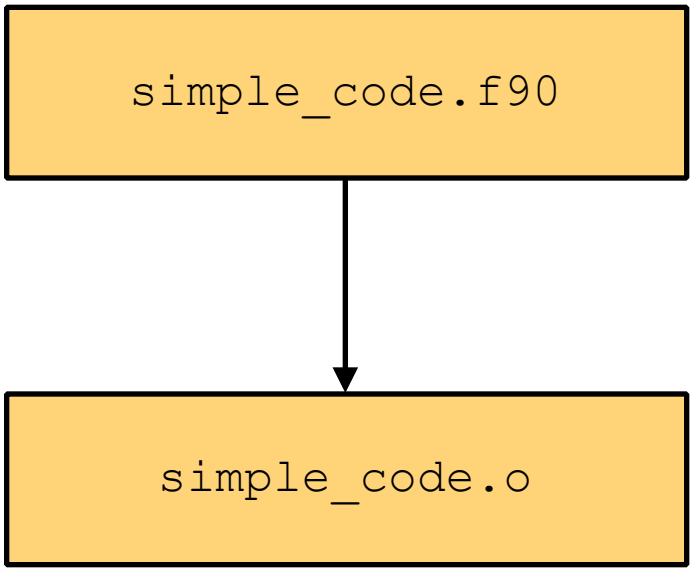
- Course overview
- Structure of simple Fortran program
- Compiling and linking
- Intrinsic data types
- Operations
- Intrinsic procedures
- Control flow
- Input/output

Compiling & Linking



Source code/source file

Compiling & Linking

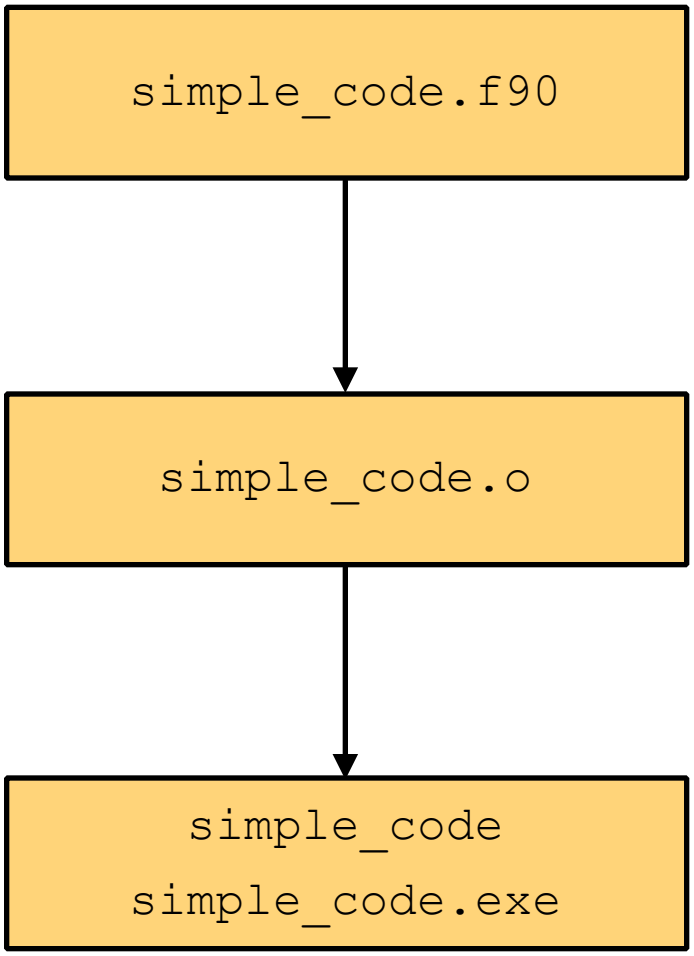


Source code/source file

Compiling

Object code/object file

Compiling & Linking



Source code/source file

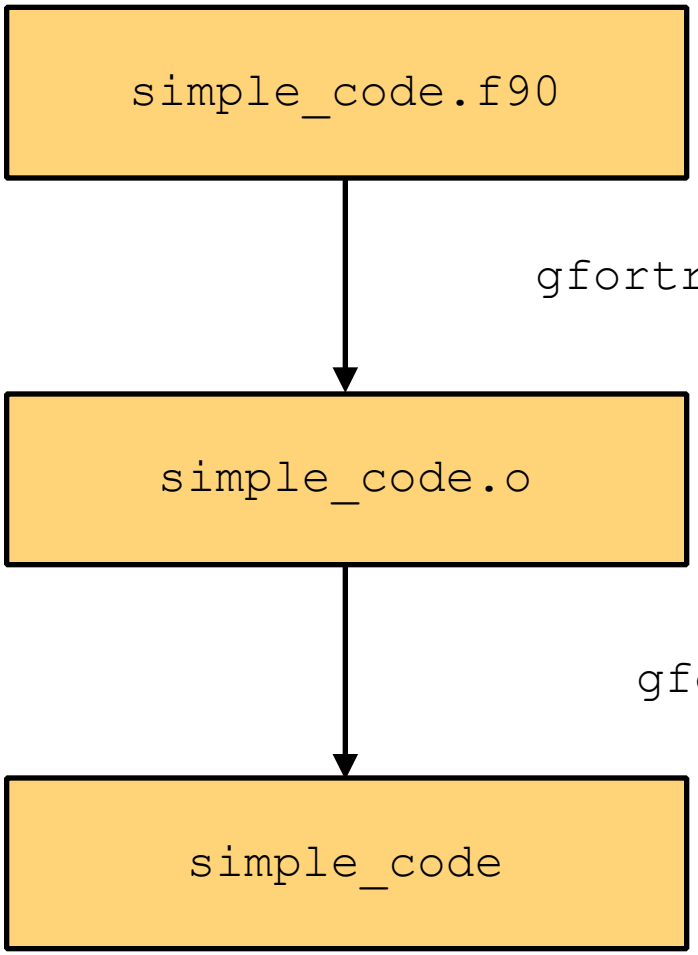
Compiling

Object code/object file

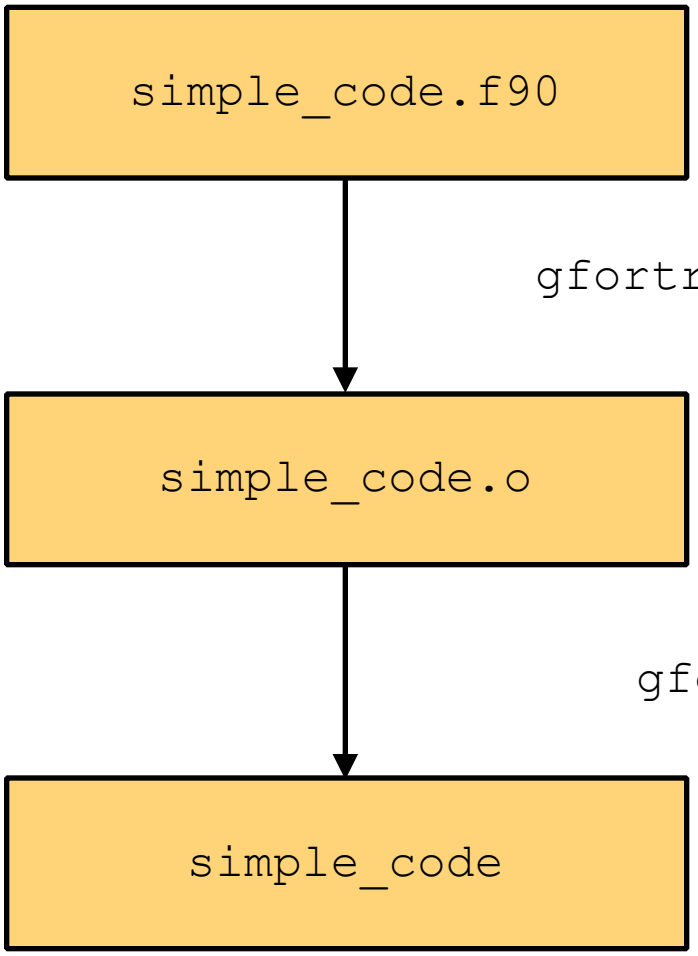
Linking

Executable/executable file

Compiling & Linking



Compiling & Linking



Compiler option indicating "compile only"

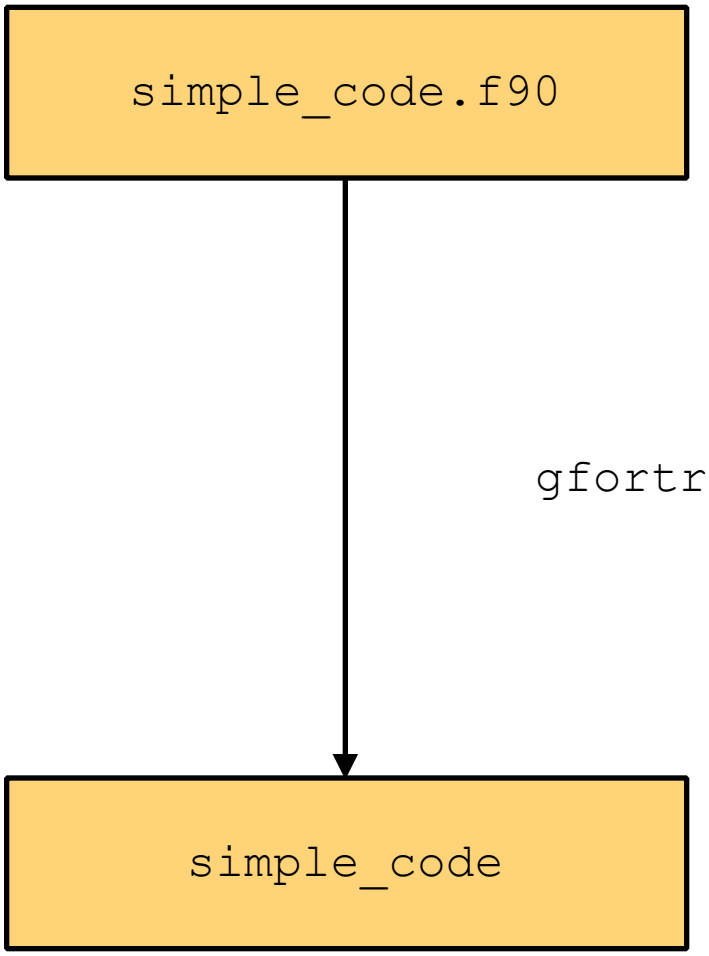
```
gfortran -c simple_code.f90 -o simple_code.o
```

Compiler option indicating name of output

```
gfortran simple_code.o -o simple_code
```

Many additional compiler options are available
We'll talk about some of them later

Compiling & Linking

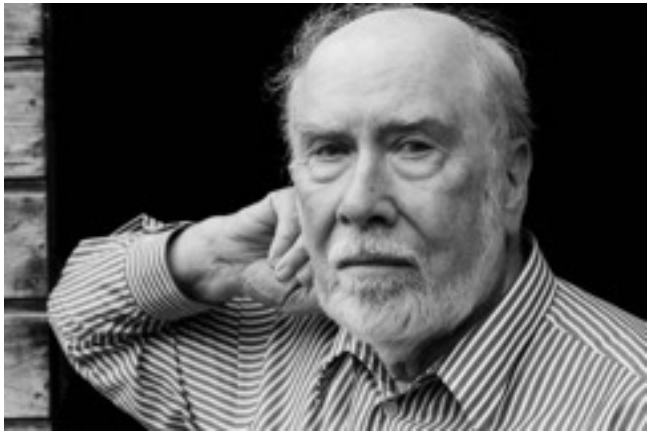



```
gfortran simple_code.f90 -o simple_code
```

Algorithms + Data Structures = Programs

Niklaus Wirth (*1934)

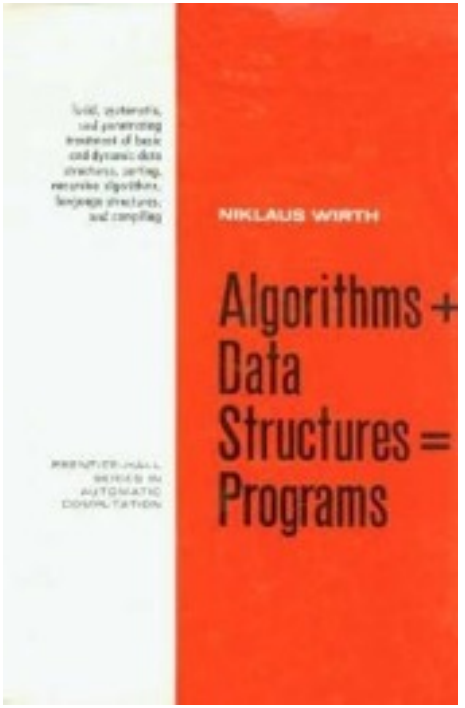
- Swiss computer scientist
- Degree from ETH in 1959
- Professor at ETH from 1968-1999



http://www.heidelberg-laureate-forum.org/blog/laureate/niklaus-e-wirth/

Quote perfectly encapsulates two central components of a program:

- Data (and how it is stored)
- Algorithms/methods/operations that manipulate data



Overview

- Course overview
- Structure of simple Fortran program
- Compiling and linking
- Intrinsic data types
- Operations
- Intrinsic procedures
- Control flow
- Input/output

Intrinsic Data Types

- Fortran provides six intrinsic data types:
  - `INTEGER`
  - `REAL`
  - `DOUBLE PRECISION`
  - `COMPLEX`
  - `LOGICAL`
  - `CHARACTER`
- Each can be used to declare variables or constants
- Can also define your own data types

Intrinsic Data Types: `INTEGER`

- Integer variables declared with:

```
INTEGER :: i,j,k

i = 1
j = 2
k = i + j
```

Recommendation:
Use uppercase names for `PARAMETER`s

Use underscores to separate words

- Integer constants declared with:

```
INTEGER, PARAMETER :: NEDGES_MAX = 1000
```

- Constants (of any data type) often called named constants
- Constants (of any data type) cannot be changed after the declaration

Intrinsic Data Types: `INTEGER`

- Q: Why use named constants?

Intrinsic Data Types: `INTEGER`

- Computers represent data as strings of bits using binary number system
- Each bit assumes values 0 or 1
- Number of bits in string therefore determines range of values that can be represented
- Assuming 8-bit representation:

```
00000010 =           1*2^1 + 0*2^0 =  2
00100011 = 1*2^5 + 1*2^1 + 1*2^0 = 35
```

- Minimum and maximum number that could be represented:

```
00000000 =   0
11111111 = 255
```

- With n bits, can represent numbers ranging from 0 to 2^n-1

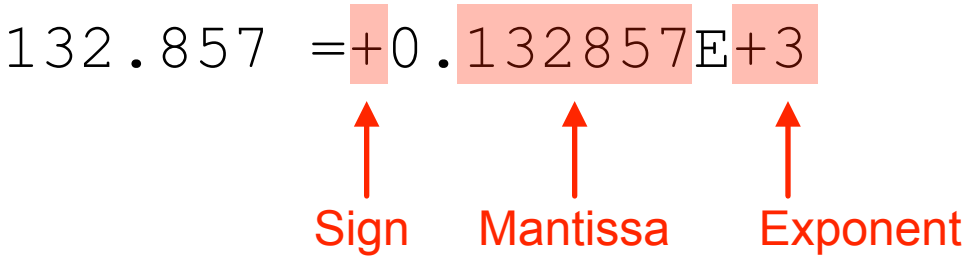
Intrinsic Data Types: `INTEGER`

Program example

- Q: How to represent negative integers?
- A: Reserve one bit to represent sign
- Thus can represent numbers ranging from -2^{n-1} to $2^{n-1}-1$
- Fortran uses 32 bits for `INTEGER`s, thus can represent numbers ranging from -2147483648 to 2147483647
- Very important: Range of representable numbers is limited
- Exceeding the range leads to integer overflow or underflow

Intrinsic Data Types: `REAL`

- Computers represent floating-point numbers:



- Range of real numbers that can be represented depends on numbers of bits used for mantissa and exponent
- Fortran uses 32 bits for `REAL`s:
  - 1 bit for sign
  - 23 bits for mantissa
  - 8 for exponent
- Most floating-point numbers are rounded, so not exact
- Largest number: `3.40282347E+38`
- Machine precision: `1.19209290E-07`

Intrinsic Data Types: `DOUBLE PRECISION`

Program example

- If `REAL` range too small or machine precision too large, need `DOUBLE PRECISION` representation
- Fortran uses 64 bits for `DOUBLE PRECISION`:
  - 1 bit for sign
  - 52 bits for mantissa
  - 11 for exponent
- Largest number: `1.79769313486231 57E+308`
- Machine precision: `2.22044604925031 31E-016`

Intrinsic Data Types: `COMPLEX`

- Convenient way to represent complex numbers:

```
COMPLEX :: c,d

c = (1.0,-2.0)
d = CMPLX(-3,4)
```

Represents 1-2i

Represents -3+4i

- Fortran provides intrinsic procedures for complex numbers, such as magnitudes, complex conjugates, etc.

Intrinsic Data Types: `LOGICAL`

- Logical variables declared with:

```
LOGICAL :: testPassed,resultNegative
```

- Logical variables and constants can only take two values:

```
testPassed     = .TRUE.
resultNegative = .FALSE.
```

Periods important!

- Logical constants can be declared with:

```
LOGICAL, PARAMETER :: flagTrue = .TRUE.
```

- Use of logical constants limited

Intrinsic Data Types: `CHARACTER`

- Single characters are declared with

```
CHARACTER :: c
```

```
c = 'a'
c = "b"
c = '&'
```

Can initialise characters with single or double quotes

- Single characters are not, of course, very useful

Intrinsic Data Types: `CHARACTER`

- To represent words or phrases, use strings of characters
- Example of declaration of string:

```
CHARACTER(LEN=11) :: c
```

```
c = "hello world"
```

← Ok to initialise to string with less than 11 characters, but not more

- Careful with apostrophes:

```
c = 'what's up?'
```

← Gives compiler error (unbalanced quotes)

```
c = "what's up?"
c = 'what''s up?'
```

← Both ok

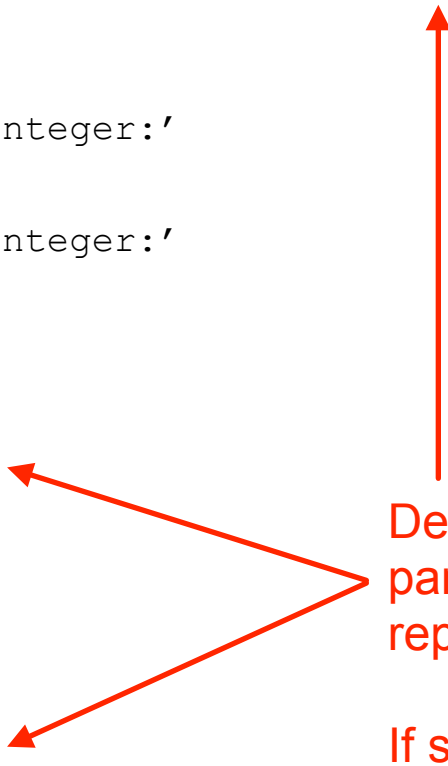
- Can also use shorter declaration

```
CHARACTER(11) :: c
```

Intrinsic Data Types: CHARACTER

- Declaration of character string parameters:

```
CHARACTER(6), PARAMETER :: ERROR_MESSAGE = 'ERROR!'
INTEGER :: i,j

WRITE(*,*) 'Enter positive integer:'
READ(*,*) i
WRITE(*,*) 'Enter negative integer:'
READ(*,*) j

IF ( i < 0 ) THEN
  WRITE(*,*) ERROR_MESSAGE
  STOP
END IF

IF ( j > 0 ) THEN
  WRITE(*,*) ERROR_MESSAGE
  STOP
END IF
```



Definition of character string parameter useful if used repeatedly

If string needs to be changed, only one change is necessary

Intrinsic Data Types: `IMPLICIT NONE`

- In older versions of Fortran, implicit declaration possible:
  - Variables starting with `i`, `j`, `k`, `l`, `m`, `n` were integers by default
  - All other variables were real variables by default
  - Backward compatibility means can still be used
- Now considered dangerous because it masks typing mistakes:

```
SUBROUTINE MakeVectorUnitVector(b)

REAL, INTENT(INOUT) :: b(2)
REAL :: length

length = SQRT(b(1)*b(1)+b(2)*b(2))
b(1) = b(1)/lngth
b(2) = b(2)/lngth

END SUBROUTINE MakeVectorUnitVector
```

This mistake could be caught be compiler options, more later

- Hence recommend that always use `IMPLICIT NONE`

Intrinsic Data Types

- Single variables are not very useful
- Often collection of variables in arrays convenient/needed
- Examples:
  - Solution of linear system of equations
  - Finite-difference/volume/element solutions of PDEs
  - …

Intrinsic Data Types: `INTEGER` Arrays

- Declaration and initialization of 1d `INTEGER` arrays:

```
INTEGER :: a(4)
INTEGER, DIMENSION(4) :: b
```

Equivalent declarations of one-dimensional array with 4 elements

```
a(1) = 1
a(2) = -4
a(3) = 37
a(4) = -121
```

```
a = (/1,-4,37,-121/)
```

Equivalent initializations of array `a`

```
b(1:4) = 0
```

Intrinsic Data Types: `INTEGER` Arrays

- By default, Fortran array elements are indexed 1, 2, …
- Can be changed by suitable declaration and initialization:

```
INTEGER :: a(-1:2)
INTEGER, DIMENSION(0:3) :: b
```

```
a(-1) = 1
a( 0) = -4
a( 1) = 37
a( 2) = -121
```

```
a = (/1,-4,37,-121/)
```

Equivalent initializations of array `a`

```
b(:) = 0
```

This initialization to a constant value works for any numbering

Intrinsic Data Types: `INTEGER` Arrays

- Declaration and initialization of 2d `INTEGER` arrays is similar:

```
INTEGER :: a(2,3)
INTEGER, DIMENSION(4,2) :: b

a(1,1) = 1
a(2,1) = -4
a(1,2) = 37
a(2,2) = -121
a(1,3) = 456
a(2,3) = -17

b(:,:) = 0
```

Intrinsic Data Types: `INTEGER` Arrays

- Fortran stores multi-dimensional arrays in column-major order:

```
INTEGER :: a(2,3)

a(1,1) = 1
a(2,1) = -4
a(1,2) = 37
a(2,2) = -121
a(1,3) = 456
a(2,3) = -17
```

1	37	456
-4	-121	-17

Address	Value
0	a(1,1)= 1
1	a(2,1)= -4
2	a(1,2)= 37
3	a(2,2)=-121
4	a(1,3)= 456
5	a(2,3)= -17

- In general: leftmost subscript varies fastest
- This has important performance implications
- Maximum number of array dimensions: 7

Intrinsic Data Types: Other Arrays

- Arrays for other intrinsic data types defined similarly
- Careful with arrays of character strings:

```
CHARACTER(20) :: a(80)
```

An array of 80 strings, each of which is 20 characters long

Arrays: Stepping Out-of-Bounds

- Stepping out of bounds of an array is common mistake:

```
PROGRAM bounds1d

INTEGER :: a(4)

a = (/1,-4,37,-121/)

WRITE(*,*) a(0)

END PROGRAM bounds1d
```

Compile with

```
gfortran bounds1d.f90 -o bounds1d
```

Compiler produces executable, but also issues warning:

```
bounds1d.f90:10.15:

  WRITE(*,*) a(0)
               1
Warning: Array reference at (1) is
out of bounds (0 < 1) in dimension 1
```

Arrays: Stepping Out-of-Bounds

- What about following program?

```
PROGRAM bounds1d

IMPLICIT NONE

INTEGER :: i
INTEGER :: a(4)

a = (/1,-4,37,-121/)

WRITE(*,*) 'Enter i:'
READ(*,*) i

WRITE(*,*) a(i)

END PROGRAM bounds1d
```

Compiler cannot issue warning, as index `i` is unknown at compile time

Arrays: Stepping Out-of-Bounds

- What happens when you run this program?

i	a(i)
-2	0
-1	0
0	0
1	1
2	-4
3	37
4	-121
5	1475824016
6	32767

Undefined behaviour if `i < 1` or `i > 4`

"Undefined" means that behaviour may be different on different machines, with different compilers, and different each time you run program (even on same machine and with same compiler)

Arrays: Stepping Out-of-Bounds

- Compiler can help you avoid this problem
- By compiling with

```
gfortran -fbounds-check bounds1d.f90 -o bounds1d
```

get following behavior:

```
Enter i:
0
At line 14 of file bounds1d.f90
Fortran runtime error: Index '0' of dimension 1 of array 'a'
below lower bound of 1
```

- Bottom line: Compiler can help you detect and find mistakes
- But you need to tell compiler to do so with appropriate options

Recommended Compiler Options

- Recommendation: while developing program, turn on all compiler options that help detect and find mistakes
- For example:

Option	Effect
`-g`	Produce information for debugger
`-Wall -Wextra`	Issue (lots of) warnings
`-fbounds-check`	Check stepping out of bounds of arrays
`-fbacktrace`	Produce list of procedures called when program crashes
`-ffpe-trap=zero,overflow,underflow,invalid`	Trap division by zero, overflow, underflow, invalid operations

- Full list of warning options:

```
gfortran --help=warnings
```

- These options will slow down executable, ok during development and testing

Recommended Compiler Options

- Once program working correctly, use compiler options to speed up execution:

  ```
  gfortran -On code.f90 -o code
  ```

  where *n* is number from 0 (no optimization) to 3 (most aggressive optimization)
- Check that optimization does not influence results

- Bottom line:
  - Compiler great help in code development; make use of it
  - Familiarize yourself with options of your compiler(s)

Overview

- Course overview
- Structure of simple Fortran program
- Compiling and linking
- Intrinsic data types
- Operations
- Intrinsic procedures
- Control flow
- Input/output

Arithmetic Operations

- Binary operations:

=	Assignment	`a = b` `i = i + 1`
+	Addition	`a + b`
-	Subtraction	`a - b`
/	Division	`a/b`
*	Multiplication	`a*b`
**	Exponentiation	`a**b`

- Unary operations:

+	?	`a = +b`
-	Negation	`a = -b`

- Precedence (top to bottom):

**	left to right
* /	left to right
+ -	left to right

- Use parentheses to influence precedence:

```
a = (a+b)**n/(c*(a-b/(e+f)))
```

Arithmetic Operations: Integer/Mixed Arithmetic

- Careful with Fortran's integer arithmetic, i.e., arithmetic involving only integers
- Fortran's integer arithmetic always produces integer result by truncating fractional part
- Examples:

```
3/4 = 0
4/3 = 1
5/3 = 1
6/3 = 2
```

- Looks like major inconvenience, but can be useful at times

Arithmetic Operations: Integer/Mixed Arithmetic

- To divide two integers, change at least one to real number to get real result:

```
3/4   = 0
3.0/4 = 0.75
3/4.0 = 0.75
```

or

```
i/REAL(j)     i/DBLE(j)
REAL(i)/j     DBLE(i)/j
```

- In mixed (or mixed-mode) arithmetic, Fortran converts integers to real numbers:

```
1 + 3/4   = 1
1 + 3.0/4 = 1.75
1.0 + 3/4 = 1.0
```

- Recommendation: Avoid mixed arithmetic by converting integers yourself using `REAL` and `DBLE` intrinsic procedures

Relational Operations

Program example

- Binary operations:

==	equal to	`a == b`
/=	not equal to	`a /= b`
>	greater than	`a > b`
>=	greater than or equal to	`a >= b`
<	smaller than	`a < b`
<=	smaller than or equal to	`a <= b`

- Careful when comparing real numbers!

- Precedence: Left to right

Logical Operations

- Can only be used with logical variables and constants
- Binary operations:

`.AND.`	logical AND
`.OR.`	logical OR
`.EQV.`	logical equivalence
`.NEQV.`	logical nonequivalence

- Unary operation:

`.NOT.`	logical NOT

Logical Operations

- Truth table:

A	B	A .AND. B	A .OR. B	A .EQV. B	A .NEQV. B
.FALSE.	.FALSE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.
.FALSE.	.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.TRUE.	.TRUE.	.TRUE.	.TRUE.	.TRUE.	.FALSE.

- Precedence (top to bottom):

.NOT.	left to right
.AND.	left to right
.OR.	left to right
.EQV. .NEQV.	left to right

.NEQV. is equivalent to "exclusive or"

Logical Operations: Example of a Mistake

- What I wanted the program to be:

```
IF ( (flag1 .EQV. .FALSE.) .AND. (flag2 .EQV. .FALSE.) ) THEN
  Fortran statements
END IF
```

- What I actually typed:

```
IF ( flag1 .EQV. .FALSE. .AND. flag2 .EQV. .FALSE. ) THEN
  Fortran statements
END IF
```

- Precedence rules mean this is equivalent to:

```
IF ( (flag1 .EQV. (.FALSE. .AND. flag2)) .EQV. .FALSE. ) THEN
  Fortran statements
END IF
```

flag1	flag2	Desired expression	Actual expression
.FALSE.	.FALSE.	.TRUE.	.FALSE.
.FALSE.	.TRUE.	.FALSE.	.FALSE.
.TRUE.	.FALSE.	.FALSE.	.TRUE.
.TRUE.	.TRUE.	.FALSE.	.TRUE.

Overview

- Course overview
- Structure of simple Fortran program
- Compiling and linking
- Intrinsic data types
- Operations
- Intrinsic procedures
- Control flow
- Input/output

Intrinsic procedures

- Fortran provides 108 intrinsic procedures to operate on data
- These include trigonometric procedures, for example
- We will later learn how to write our own procedures
- Following covers only most important intrinsic procedures
- For complete list, see:
  - Chapman (2008), Appendix B
  - Ellis et al. (1994), Appendix A
  - Metcalf et al. (2004), Appendix A

Intrinsic procedures: Integer data

`ABS(x)`	Absolute value of `x`	
`DBLE(x)`	Converts `x` to double-precision floating-point value	
`MAX(a,b)`	Picks the larger of `a` and `b`	
`MIN(a,b)`	Picks the smaller of `a` and `b`	
`MOD(a,b)`	Remainder (modulo) function	
`REAL(x)`	Converts `x` to single-precision floating-point value	

Intrinsic procedures: Floating-point data

`ABS(x)`	Absolute value of `x`	
`ACOS(x)`	Inverse cosine of `x`	result in radians
`ACOSH(x)`	Inverse hyperbolic cosine of `x`	
`ASIN(x)`	Inverse sine of `x`	result in radians
`ASINH(x)`	Inverse hyperbolic sine of `x`	
`ATAN(x)`	Inverse tangent of `x`	result in radians, -π/2 to π/2
`ATAN2(y,x)`	Inverse tangent of `y/x`	result in radians, -π to π
`ATANH(x)`	Inverse hyperbolic tangent of `x`	
`CMPLX(a,b)`	Convert `a,b` to complex number a+bi	
`COS(x)`	Cosine of `x`	`x` in radians
`COSH(x)`	Hyperbolic cosine of `x`	
`EXP(x)`	Exponential of `x`	
`INT(x)`	Truncates `x` to integer	

Intrinsic procedures: Floating-point data

`LOG(x)`	Natural logarithm of `x`	`x>0`
`LOG10(x)`	Base-10 logarithm of `x`	`x>0`
`MAX(a,b)`	Picks the larger of `a` and `b`	
`MIN(a,b)`	Picks the smaller of `a` and `b`	
`MOD(a,b)`	Remainder (modulo) function	
`NINT(x)`	Nearest integer to `x`	
`SIGN(a,b)`	Value of `a` with sign of `b`	
`SIN(x)`	Sine of `x`	`x` in radians
`SINH(x)`	Hyperbolic sine of `x`	
`SQRT(x)`	Square root of `x`	$x \geq 0$
`TAN(x)`	Tangent of `x`	`x` in radians
`TANH(x)`	Hyperbolic tangent of `x`	

Intrinsic procedures: Complex data

`ABS(x)`	Magnitude of `x`	
`AIMAG(x)`	Imaginary part of `x`	
`CONJG(x)`	Complex conjugate of `x`	
`REAL(x)`	Real part of `x`	

Intrinsic procedures: Character string data

`LEN(x)`	Length of `x`	
`LEN_TRIM(x)`	Length of `x`, not counting trailing blank characters	
`TRIM(x)`	Remove trailing blank characters	

Intrinsic procedures: Arrays

`DOT_PRODUCT(a,b)`	Dot product of vectors `a` and `b`	`a` and `b` must be conforming
`LBOUND(a,i)`	Lower bound of `i`th dimension of `a`	
`MATMUL(a,b)`	Matrix product of `a` and `b`	`a` and `b` must be conforming
`MAXLOC(a,m)`	Location of first element of maximum value in `a` given mask `m`	See any of the books for more information
`MAXVAL(a,m)`	Maximum value of elements in `a` given mask `m`	See any of the books for more information
`MINLOC(a,m)`	Location of first element of minimum value in `a` given mask `m`	See any of the books for more information
`MINVAL(a,m)`	Minimum value of elements in `a` given mask `m`	See any of the books for more information
`SIZE(a,i)`	Size of a along `i`th dimension	

Intrinsic procedures: Arrays

`SUM(a,i,m)`	Sum of elements in `a` along dimension `i` given mask `m`	See any of the books for more information
`TRANSPOSE(a)`	Transpose of `a`	
`UBOUND(a,i)`	Upper bound of `i`th dimension of `a`	

Overview

- Course overview
- Structure of simple Fortran program
- Compiling and linking
- Intrinsic data types
- Operations
- Intrinsic procedures
- Control flow
- Input/output

Control Flow

- Often necessary to:
  - Take action in response to data
  - Repeat actions multiple times
- Both amount to “controlling the flow” in the program

Control Flow: `IF` Statements

- General case ("block `IF`"):

```
IF ( logical_expression ) THEN
  Fortran statements
ELSE IF ( logical_expression ) THEN
  Fortran statements
ELSE IF ( logical_expression ) THEN
  Fortran statements
ELSE
  Fortran statements
END IF
```

There can be any number of `ELSE IF` statements (including none)

Important: First true expression is taken

There can be at most one `ELSE` statement

It must appear after the `ELSE IF` statements (if there are any)

- Special case ("logical IF"):

```
IF ( logical_expression ) Fortran statement
```

Control Flow: `IF` Statements

- IF statements can be nested:

```
IF ( logical_expression ) THEN
  IF ( logical_expression ) THEN
    Fortran statements
  ELSE IF
    IF ( logical_expression ) THEN
      Fortran statements
    ELSE
      Fortran statements
    END IF
  END IF
ELSE IF ( logical_expression ) THEN
  Fortran statements
ELSE
  Fortran statements
END IF
```

Control Flow: `IF` Statements

- IF statements can be nested:

```
outerIF: IF ( logical_expression ) THEN
  middleIF: IF ( logical_expression ) THEN
    Fortran statements
  ELSE IF
    innerIF: IF ( logical_expression ) THEN
      Fortran statements
    ELSE
      Fortran statements
    END IF innerIF
  END IF middleIF
ELSE IF ( logical_expression ) THEN
  Fortran statements
ELSE
  Fortran statements
END IF outerIF
```

Fortran 95 allows you to name `IF` statements

Benefits:

1. Easier for people to understand nested `IF`s
2. If you accidentally delete an IF or END IF, compiler can treat is as an error

Control Flow: `SELECT CASE` Statement

```
SELECT CASE ( case_expression )
  CASE ( case_selector_1 )
    Fortran statements
  CASE ( case_selector_2 )
    Fortran statements
  CASE ( case_selector_3 )
    Fortran statements
  CASE DEFAULT
    Fortran statements
END SELECT
```

There must be at least one `CASE` statement

If there is more than one `CASE` statement, the `case_selector`s must be mutually exclusive

The case selectors must be constants (integer, character, or logical)

The statements under `CASE DEFAULT` are executed if none of the other `CASE` statements are executed

The `CASE DEFAULT` statement is not required

It is recommended that you include it to catch programming mistakes

Control Flow: `SELECT CASE` Statement

Example:

```
SELECT CASE( TRIM(line) )
  CASE ('# BC_FARF')
    CALL RFLU_ReadBcFarfSection(pRegion)
  CASE ('# BC_INFLOW_TOTANG')
    CALL RFLU_ReadBcInflowTotAngSection(pRegion)
  CASE ('# BC_INFLOW_VELTEMP')
    CALL RFLU_ReadBcInflowVelTempSection(pRegion)
  CASE ('# BC_INJECT')
    CALL RFLU_ReadBcInjectSection(pRegion)
    …
  CASE DEFAULT
    CALL ErrorStop(global,ERR_REACHED_DEFAULT)
END SELECT ! TRIM(line)
```

Control Flow: `SELECT CASE` Statement

Example:

```
SELECT CASE ( flag )
  CASE ( 1,2 )
    DO iPatch = 1,pGrid%nPatches
      pPatch => pRegion%patches(iPatch)
      …
    END DO ! iPatch
  CASE ( 3 )
    DO iPatch = 1,pGrid%nPatches
      pPatch => pRegion%patches(iPatch)
      …
    END DO ! iPatch
  CASE DEFAULT
    CALL ErrorStop(global,ERR_REACHED_DEFAULT)
END SELECT ! flag
```

`case_selector` can consist of more than one value, separated by commas

`case_selector` can also specify range of values:
`CASE ( val_min:val_max )`

Control Flow: `SELECT CASE` or `IF`?

`IF`	`SELECT CASE`
`logical_expression` may involve variables	*`case_selector`* must only involve constants
Order of `ELSE IF`s does matter	Order of `CASE`s does not matter

Defensive Programming

- Recommendation: always include `CASE DEFAULT` statement
- If it should not be reached, write error message if reached
- Example of defensive programming

Defensive Programming

- McConnell (2004, p. 187):

  "The idea is based on defensive driving. In defensive driving, you adopt the mind-set that you're never sure what other drivers are going to do. That way, you make sure that if they do something dangerous you won't be hurt. You take responsibility for protecting yourself even when it might be the other driver's fault. In defensive programming, the main idea is that if a routine is passed bad data, it won't be hurt, even if the bad data is another routine's fault. More generally, it is the recognition that programs will have problems and modifications, and that a smart programming will develop code accordingly."

- My take:
  - Anticipate problems (Murphy's law…)
  - Write code to handle problems (not to fix them)
  - Write error messages to let you know problem occurred

KISS Principle

- KISS = Keep it simple, stupid
- Simple programs are easier to write, debug, and modify
- Often bad idea to use elegant code that is hard to understand
- Not uncommon to have difficulty understanding code that was written only a couple of weeks/months ago
- Bottom line: Write simple programs
- Does not mean that simple programs are inefficient

Control Flow: DO Statements

- DO-loops are used to repeat operations
- First form of DO-loop:

```
DO i = i_beg,i_end,i_inc
   Fortran statements
END DO
```

which is equivalent to:

```
i = i_beg
Fortran statement
i = i + i_inc
Fortran statement
 …
i = i_end
Fortran statement
```

`i` "loop counter", must be integer

`i_beg` must be integer expression

`i_end` must be integer expression

`i_inc` "loop increment" (1 if not specified)

`i, i_beg, i_end` must not be changed in loop

- Known as "iterative" or "counting" or "count-controlled" loop

Control Flow: DO Statements

- Sometimes, need to repeat statements until condition is met, but number of repetitions is not known
- Second form of DO-loop:

```
DO
   Fortran statements
   IF ( logical expression ) THEN
      Fortran statements
      EXIT
   END IF
   Fortran statements
END DO
```

Causes control to be transferred to first statement after END DO

- Sometimes referred to as “while” loop
- Note that if *logical expression* is never true, get so-called infinite loop…

Defensive Programming

- Avoid infinite loops by introducing loop counters:

```
INTEGER, PARAMETER :: MAX_LOOP_COUNTER = 1000

loopCounter = 0
DO
  loopCounter = loopCounter + 1
  Fortran statements
  IF ( logical expression ) THEN
    Fortran statements
    EXIT
  END IF
  IF ( loopCounter == MAX_LOOP_COUNTER ) THEN
    EXIT
  END IF
  Fortran statements
END DO
```

Maximum value of loop counter must be chosen large enough

Defensive Programming

- Problem: Do not why loop was exited
- Checking `loopCounter` is a possible solution
- Another solution shown on next slide…

Defensive Programming

- Solution:

```
LOGICAL :: flagReachedMaxCounter
DO
  loopCounter = loopCounter + 1
  Fortran statements
  IF ( logical expression ) THEN
    Fortran statements
    flagReachedMaxCounter = .FALSE.
    EXIT
  END IF
  IF ( loopCounter == MAX_LOOP_COUNTER ) THEN
    flagReachedMaxCounter = .TRUE.
    EXIT
  END IF
  Fortran statements
END DO
```

Defensive Programming

- Introduce check after loop:

```
  …
  IF ( loopCounter == MAX_LOOP_COUNTER ) THEN
    flagReachedMaxCounter = .TRUE.
    EXIT
  END IF
  Fortran statements
END DO

IF ( flagReachedMaxCounter .EQV. .TRUE. ) THEN
  WRITE(*,*) 'WARNING! Maximum loop counter reached!'
END IF
```

Control Flow: DO Statements

- Fortran allows DO loops to be named:

```
LOGICAL :: flagReachedMaxCounter

whileLoop: DO
  loopCounter = loopCounter + 1
  Fortran statements
  IF ( logical expression ) THEN
    Fortran statements
    flagReachedMaxCounter = .FALSE.
    EXIT whileLoop
  END IF
  IF ( loopCounter == MAX_LOOP_COUNTER ) THEN
    flagReachedMaxCounter = .TRUE.
    EXIT
  END IF whileLoop
  Fortran statements
END DO whileLoop
```

Control Flow: `DO` Statements

- Already discussed `EXIT` statement, which transfer control to first statement after `END DO`
- There is also the `CYCLE` statement, which transfers control to `END DO` statement (and possibly back to `DO` statement)

Program example: Column-major ordering and performance

Overview

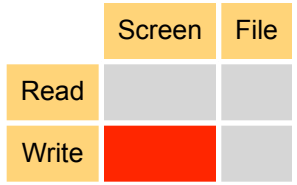
- Course overview
- Structure of simple Fortran program
- Compiling and linking
- Intrinsic data types
- Operations
- Intrinsic procedures
- Control flow
- Input/output

Input/Output

- Programs require input data and generate output data
- Four possibilites:

	Screen	File
Read	Read from screen	Read from file
Write	Write to screen	Write to file

Output: To Screen



- Writing an integer to screen is accomplished by:

```
INTEGER :: i
i = 5
WRITE(*,*) i
```

The significance of `*,*` will be explained soon

- Similar for floating-point variables
- Writing character string accomplished by:

```
INTEGER, PARAMETER :: MAX_CHAR_LEN = 80
CHARACTER(MAX_CHAR_LEN) :: c
c = 'Hello!'
WRITE(*,*) c
```

- This writes whole string, including trailing blank spaces
- This can be avoided with intrinsic procedure `TRIM`:

```
WRITE(*,*) TRIM(c)
```

Output: To Screen

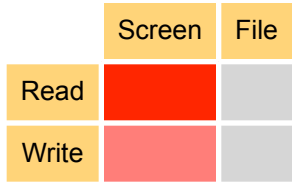
- Can combine writing of data types:

```
INTEGER, PARAMETER :: MAX_CHAR_LEN = 80
INTEGER :: i
CHARACTER(MAX_CHAR_LEN) :: c
i = 5
c = 'Hello!'
WRITE(*,*) TRIM(c),i
```

or

```
WRITE(*,*) 'i=',i
WRITE(*,*) 'c=',TRIM(c)
```

Input: From Screen



- Reading an integer from screen is accomplished by:

```
INTEGER :: i
READ(*,*) i
```

- Similarly, a single-precision floating-point variable is read by:

```
REAL :: r
READ(*,*) r
```

- Finally, a character string is read by:

```
INTEGER, PARAMETER :: MAX_CHAR_LEN = 80
CHARACTER(MAX_CHAR_LEN) :: c
READ(*,*) c
```

Input: From Screen

- Can also read several variables together:

```
INTEGER, PARAMETER :: MAX_CHAR_LEN = 80
INTEGER :: i
CHARACTER(MAX_CHAR_LEN) :: c
WRITE(*,*) 'Enter integer and string:'
READ(*,*) i,c
WRITE(*,*) i,TRIM(c)
```

Can enter one value at a time, separated by <return>

- Get run-time error if data that does not match data type:

```
Enter integer and string:
17.0 3
At line 11 of file readmixed.f90 (unit = 5, file = 'stdin')
Fortran runtime error: Bad integer for item 1 in list input
```

Defensive Programming

- Better way is to check for invalid input using `IOSTAT` specifier:

```
INTEGER :: ef
READ(*,*,IOSTAT=ef) i,c
IF ( ef /= 0 ) THEN
  WRITE(*,*) 'ERROR - Invalid input!'
  STOP
END IF ! ef
```

giving

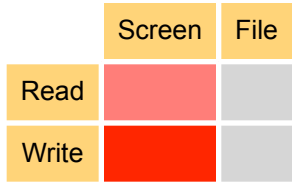
```
Enter integer and string:
17.0 3
ERROR - Invalid input!
```

`IOSTAT` returns an integer status variable

Non-zero status variable indicates `READ` statement was unsuccessful

Values and meaning of non-zero variable compiler-dependent

Output: To Screen with Formatting



- Often desire to influence how output is formatted
- For example, want to write certain number of decimal places,

```
pi = 3.141593
```

instead of

```
pi = 3.14159265358979323846264338327 9
```

or use engineering notation,

```
p = 1.01325E+5
```

instead of

```
p = 101325.0
```

- This can be achieved using edit descriptors
- They are used in place of second `*` in `WRITE` statement:

```
WRITE(*,'(edit descriptors)') something
```

Note apostrophes and parentheses!

Output: To Screen with Formatting

- Following only covers most important edit descriptors
- For more information, see:
  - Chapman (2008): 5.3 and 14.1
  - Ellis et al. (1994): 8.2 and 15.2
  - Metcalf et al. (2004): 9

Output: To Screen with Formatting

- For character variables and constants, use `A` edit descriptor:

`Aw`	*w* = overall width

- Example code:

```
PROGRAM editdescriptors

IMPLICIT NONE

INTEGER, PARAMETER :: MAX_CHAR_LEN = 80
CHARACTER(MAX_CHAR_LEN) :: c

c = 'We are testing edit descriptors!'

WRITE(*,*) LEN_TRIM(c)

WRITE(*,'(A10)') c
WRITE(*,'(A32)') c
WRITE(*,'(A33)') c

END PROGRAM editdescriptors
```

- Output

```
          32
We are tes
We are testing edit descriptors!
We are testing edit descriptors!
```

Output truncated if width too small!

Output: To Screen with Formatting

- For real and double-precision variables and constants:

`E`*w*`.`*d*	*w* = overall width *d* = number of decimal places	±0.nnnnnnnE±ee 123<- *d* ->4567 *w* ≥ *d* + 7 For example: E10.3, E13.6, E16.9
`F`*w*`.`*d*	*w* = overall width *d* = number of decimal places	±mmm.nnnnnnn <- *d* ->

Output: To Screen with Formatting

- Example code:

```
PROGRAM editdescriptors

IMPLICIT NONE

REAL :: r

r = -1234567890.987654321

WRITE(*,'(A,1X,E13.6)') 'E13.6 gives:',r
WRITE(*,'(A,1X,E10.3)') 'E10.3 gives:',r
WRITE(*,'(A,1X,E9.3)')  'E9.3  gives:',r
WRITE(*,'(A,1X,E8.3)')  'E8.3  gives:',r
WRITE(*,'(A,1X,E7.3)')  'E7.3  gives:',r

END PROGRAM editdescriptors
```

Combine edit descriptors separated by commas, Must match!

Use *n*X edit descriptor to insert *n* empty spaces

- Output:

```
E13.6 gives: -0.123457E+10
E10.3 gives: -0.123E+10
E9.3  gives: -.123E+10
E8.3  gives: ********
E7.3  gives: *******
```

If overall width too small, lose 0 and get *

Output: To Screen with Formatting

- Example code:

```
PROGRAM editdescriptors

IMPLICIT NONE

REAL :: r

r = -12345.0

WRITE(*,'(A,1X,F8.1)') 'F8.1 gives:',r
WRITE(*,'(A,1X,F9.2)') 'F9.2 gives:',r
WRITE(*,'(A,1X,F7.1)') 'F7.1 gives:',r

END PROGRAM editdescriptors
```

- Output:

```
F8.1 gives: -12345.0
F9.2 gives: -12345.00
F7.1 gives: *******
```

If overall width too small, get *

Output: To Screen with Formatting

- For integer variables and constants, use `I` edit descriptor:

`I`*w*	*w* = overall width

- For logical variables and constants, use L edit descriptor:

`L`*w*	*w* = overall width

Output: To Screen with Formatting

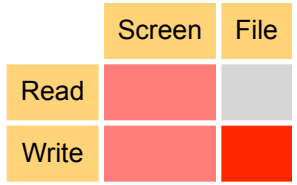
- Any edit descriptor can be prefaced to indicate repetition
- For example, to write out 2 integers of width 4:

```
WRITE(*,'(2I4)')
```

- More complicated example:

```
WRITE(*,'(A,3X,A,2(1X,E23.16),2(1X,I9))') s1,s2,r1,r2,i1,i2
```

Output: To File



- To write to file, replace first `*` in `WRITE` statement:
  `WRITE(fileUnit,'(edit descriptors)') something`
  where `fileUnit` is an integer
- Following will only cover most important aspects of writing to (and reading from) files
- For more information, see:
  - Chapman (2008): 5.5
  - Ellis et al. (1994): 9
  - Metcalf et al. (2004): 10
- File must (should) be opened before can write to it
- Once opened, can write data to file in same way as writing to screen

Input/Output: File Handling

- Files are opened with

  `OPEN(list of specifiers)`

- Specifiers include:

`UNIT=`*`fileUnit`*	*`fileUnit`* is a positive integer	• `fileUnit` values `0`, `5`, and `6` are reserved • The maximum value of `fileUnit` is compiler-dependent (often 99)
`FILE=`*`fileName`*	*`fileName`* is a string	
`STATUS=`*`fileStatus`*	*`fileStatus`* is a string	Permissible values of `fileStatus`: • `'NEW'` = File must not exist, causes error if does exist • `'OLD'` = File must exist, causes error if does not exist • `'REPLACE'` = Create file regardless of whether it exists • `'UNKNOWN'` = File may exist, behaviour is compiler-dependent
`IOSTAT=`*`errorFlag`*	`errorFlag` is an integer	Non-zero values of errorFlag indicate that an error occurred on opening the file. The values and meaning are compiler-dependent.

Input/Output: File Handling

- The `INQUIRE` statement can be used to find out more about particular `fileUnit`
- Here only illustrate use to determine whether file exists:

```
INQUIRE(UNIT=fileUnit,EXIST=existFlag, &
        IOSTAT=errorFlag)
```

- If the file exists, `existFlag` is set to `.TRUE.`, otherwise to `.FALSE.`
- There are many additional specifiers, see, for example, Chapman (2008, 14.3.3)
- Can be used as follows...

Defensive Programming

Opening file in Fortran:

```
INQUIRE(UNIT=fileUnit,EXIST=existFlag,IOSTAT=errorFlag)
IF ( errorFlag /= 0 ) THEN
  WRITE(*,*) 'ERROR - INQUIRE returned non-zero IOSTAT!'
  STOP
ELSE
  IF ( existFlag .EQV. .TRUE. ) THEN
    OPEN(UNIT=fileUnit,FILE=TRIM(fileName),STATUS='REPLACE',IOSTAT=errorFlag)
  ELSE
    OPEN(UNIT=fileUnit,FILE=TRIM(fileName),STATUS='NEW',IOSTAT=errorFlag)
  END IF ! existFlag

  IF ( errorFlag /= 0 ) THEN
    WRITE(*,*) 'ERROR - Could not open file!'
    STOP
  END IF ! errorFlag
END IF ! errorFlag
```

May not always be appropriate option...

Input/Output: File Handling

- Closing files is considerably easier:

  `CLOSE(list of specifiers)`

- Specifiers include:

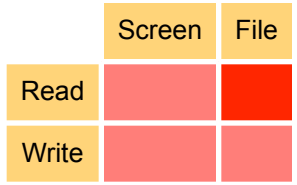
`UNIT=`*`fileUnit`*	*`fileUnit`* is a positive integer	
`STATUS=`*`fileStatus`*	*`fileStatus`* is a string	Permissible values of `fileStatus`: • `'KEEP'` = File will not be deleted after closing (default) • `'DELETE'` = File will be deleted after closing
`IOSTAT=`*`errorFlag`*	`errorFlag` is an integer	Non-zero values of errorFlag indicate that an error occurred on opening the file. The values and meaning are compiler-dependent.

Defensive Programming

Closing file in Fortran:

```
CLOSE(FILE=fileUnit,IOSTAT=errorFlag)
IF ( errorFlag /= 0 ) THEN
  WRITE(*,*) 'ERROR - Could not close file!'
  STOP
END IF ! errorFlag
```

Input: From File



- File must (should) be opened before can read from it
- Once opened, can read data from file in same way as reading from screen

Input/Output: File Handling

- Dirty little secret about Fortran file output:
  - If file with unit `fileUnit` not opened before writing to it, Fortran will write to file called `fort.`*`fileUnit`*
  - This can be very useful while developing or debugging
  - This behavior is compiler-dependent, so not portable and should not be used in general

ETH zürich

Computing with Fortran

Engineering Tool V
Spring 2015

Dr. Andreas Haselbacher

Institute of Energy Technology

PREC
Professorship of Renewable Energy Carriers

Overview

- Structuring a program
- Functions
- Makefiles
- Subroutines
- Functions & subroutines: discussion
- Modules
- Arrays
  - Dynamic memory allocation
  - Arrays as arguments
- Additional topics

Structuring a Program

- Can write complex programs with material already covered
- However, these programs will consist of one single routine
- They will be hard to understand, test, change, share…
- … which is everything we do not want programs to be

Structuring a Program

- Good approach of tackling complex problems:
  - Subdivide into smaller and simpler parts (analysis)
  - Work on smaller and simpler parts
  - Put smaller and simpler parts back together (synthesis)
- Fortran provides several ways of using this approach:
  - Procedures: Functions and subroutines
  - Modules: Data and procedures
- Functions and subroutines are important part of programming method called structured or modular programming

Overview

- Structuring a program
- Functions
- Makefiles
- Subroutines
- Functions & subroutines: discussion
- Modules
- Arrays
  - Dynamic memory allocation
  - Arrays as arguments
- Additional topics

Functions

- Example: Fortran provides intrinsic function to compute square root, but not cube root
- Can address this problem by writing own function:

```
REAL FUNCTION cbrt(x)

IMPLICIT NONE

REAL, INTENT(IN) :: x
```

← Declaration section

```
cbrt = EXP(LOG(x)/3.0)
```

← Executable section

```
END FUNCTION cbrt
```

← Termination section

Functions

- Example: Fortran provides intrinsic function to compute square root, but not cube root
- Can address this problem by writing own function:

```
REAL FUNCTION cbrt(x)
```
← Name of function and dummy argument
Note: Can have more than one dummy argument, name must not coincide with Fortran commands, intrinsic procedures

```
IMPLICIT NONE
```

```
REAL, INTENT(IN) :: x
```
← Declaration of dummy argument(s)

```
cbrt = EXP(LOG(x)/3.0)
```
← Compute result, must be assigned to variable with same name as function

```
END FUNCTION cbrt
```
← End of function

- Will address `INTENT(IN)` attribute shortly…

Functions

- This function could now be used as follows:

```
PROGRAM main

IMPLICIT NONE

REAL :: x,y,r

x = 3.0
y = 3.0
r = cbrt(x)

WRITE(*,*) r,cbrt(y)

END PROGRAM main
```

Used in assignment statement (or in other operations)

Used in `WRITE` statement

Note: In the above, `x` and `y` are actual arguments

- Two problems:
  - Small problem: what should be improved in `cbrt`?
  - Bigger problem: this program will not compile because `main` does not “know” what `cbrt` is/does

Functions

- Two basic ways of "telling" `main` about `cbrt`:

`cbrt` is internal procedure	`cbrt` is external procedure
Put `cbrt` into same file as `main`, use `CONTAINS` statement, and compile as single file	Put `cbrt` and `main` into separate files, compile separately, then link

Functions: Internal Procedure

- Example:

```
PROGRAM main
IMPLICIT NONE
REAL :: x,r
x = 3.0
r = cbrt(x)
WRITE(*,*) r

CONTAINS

REAL FUNCTION cbrt(x)
IMPLICIT NONE
REAL, INTENT(IN) :: x
cbrt = EXP(LOG(x)/3.0)
END FUNCTION cbrt

END PROGRAM main
```

Functions: Internal Procedure

- Example:

```
PROGRAM main
IMPLICIT NONE
REAL :: x,r
x = 3.0
r = cbrt(x)
WRITE(*,*) r

CONTAINS

REAL FUNCTION cbrt(x)
IMPLICIT NONE
REAL, INTENT(IN) :: x
cbrt = EXP(LOG(x)/3.0)
END FUNCTION cbrt

END PROGRAM main
```

Main program

Functions: Internal Procedure

- Example:

```
PROGRAM main
IMPLICIT NONE
REAL :: x,r
x = 3.0
r = cbrt(x)
WRITE(*,*) r

CONTAINS

REAL FUNCTION cbrt(x)
IMPLICIT NONE
REAL, INTENT(IN) :: x
cbrt = EXP(LOG(x)/3.0)
END FUNCTION cbrt

END PROGRAM main
```

CONTAINS statement

Functions: Internal Procedure

- Example:

```
PROGRAM main
IMPLICIT NONE
REAL :: x,r
x = 3.0
r = cbrt(x)
WRITE(*,*) r

CONTAINS

REAL FUNCTION cbrt(x)
IMPLICIT NONE
REAL, INTENT(IN) :: x
cbrt = EXP(LOG(x)/3.0)
END FUNCTION cbrt

END PROGRAM main
```

Function

Note: Function appears after `CONTAINS` statement but before `END PROGRAM` statement

Functions: `INTENT(IN)` Attribute

- Background: Fortran uses “call by reference”, so addresses of arguments rather than argument values are passed
- Could have written `cbrt` as:

```
REAL FUNCTION cbrt(x)

IMPLICIT NONE

REAL :: x

x = LOG(x)/3.0

cbrt = EXP(x)

END FUNCTION cbrt
```

No `INTENT(IN)` attribute

Statement changes `x`, but not `cbrt`

This function is said to have a side-effect:
It does not just return a result (main effect), but also modifies the dummy argument

Functions: `INTENT(IN)` Attribute

- Effectively, functions with side effects have two (or more) outputs
- Side effects are permitted in Fortran, but considered poor programming practice

- Good programming practice: Write functions without side effects, that is, functions that do not modify dummy arguments
- Recommendation: always use `INTENT(IN)` attribute for dummy arguments in functions

- `INTENT(IN)` attribute is nothing but a sign to compiler to check if your actions and intentions are consistent
- If variable with `INTENT(IN)` modified by mistake, compiler issues error

Functions: `INTENT(IN)` Attribute

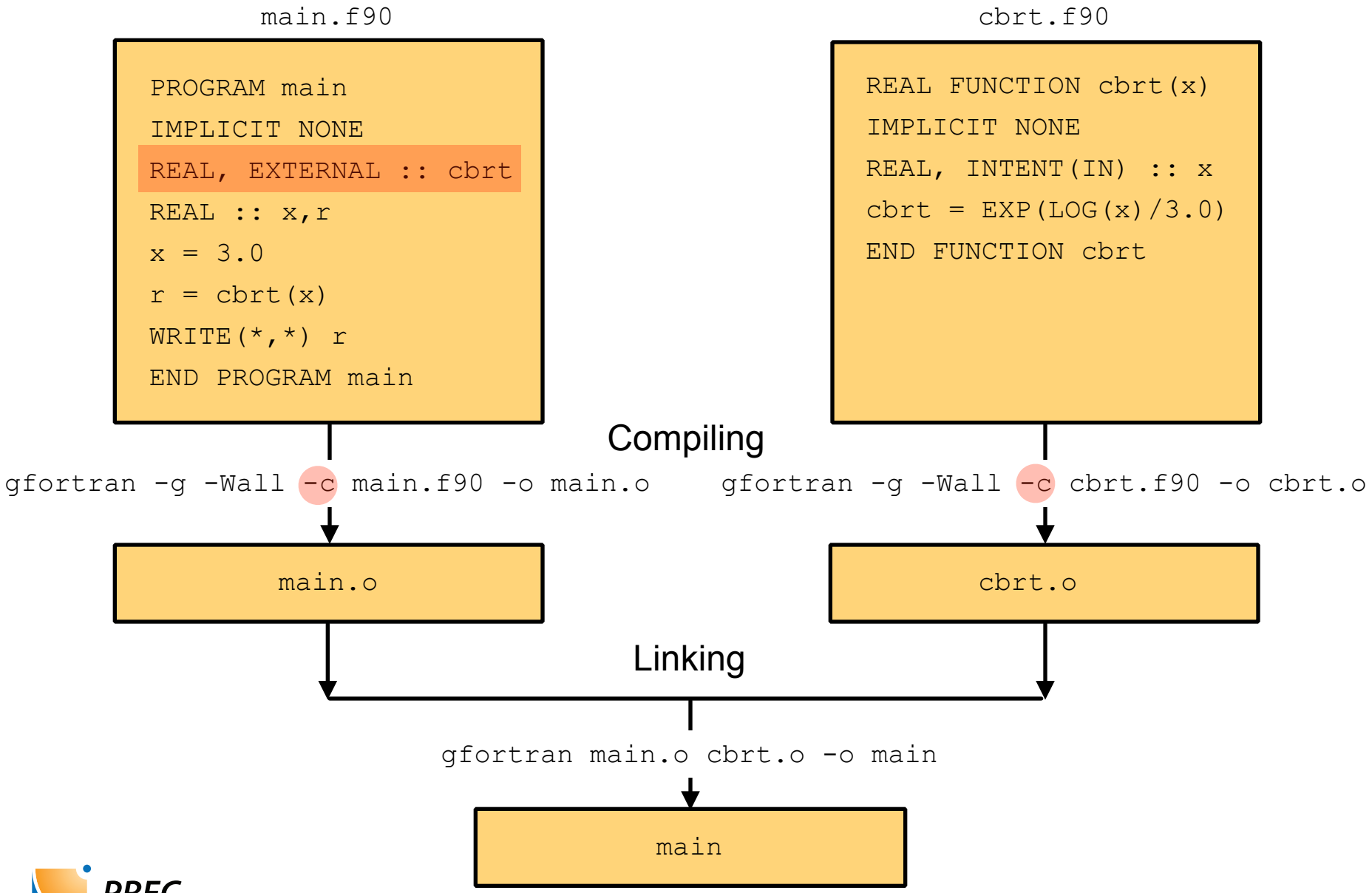
- Example:

```
REAL FUNCTION cbrt(x)

IMPLICIT NONE

REAL, INTENT(IN) :: x

x = LOG(x)/3.0

cbrt = EXP(x)

END FUNCTION cbrt
```

Statement changes `x` and hence violates `INTENT(IN)` attribute

Therefore, compiler will issue error

Functions: External Procedure



Functions: External Procedure

- Functions in files other than that containing main program must be declared with the `EXTERNAL` attribute

- Advantage: Putting functions into separate files has advantage of keeping size of main program manageable
- Disadvantage: Compiler cannot check whether function called correctly (illustrated later)
- Later, will learn how to solve this problem

Functions: Alternative Form

- Example:

```
REAL FUNCTION cbrt(x)

IMPLICIT NONE

REAL, INTENT(IN) :: x

cbrt = EXP(LOG(x)/3.0)

END FUNCTION cbrt
```

Functions: Alternative Form

- Example:

```
FUNCTION cbrt(x)

IMPLICIT NONE

REAL :: cbrt
REAL, INTENT(IN) :: x

cbrt = EXP(LOG(x)/3.0)

END FUNCTION cbrt
```

Can omit data type from `FUNCTION` statement…

… provided it is declared

Note: This form is necessary with functions that return user-defined data types

Functions: Summary

- Functions are Fortran procedures that return a single result
- Non-intrinsic functions are sometimes called user-defined functions

Overview

- Structuring a program
- Functions
- Makefiles
- Subroutines
- Functions & subroutines: discussion
- Modules
- Arrays
  - Dynamic memory allocation
  - Arrays as arguments
- Additional topics

Makefiles

- Manually compiling and linking of codes consisting of multiple files is tedious
- Makefiles can be used to automate compilation and linking
- Use/adapt the simple Makefile provided

Makefiles

```
### Macros ###########################################################
SHELL= /bin/sh
EXEC= mycode
FC= gfortran
FFLAGS= -g -Wall -Wextra -fbounds-check
#FFLAGS= -O3
MODEXT= mod
RIFFRAFF= *.bak *.bck *.ckp *.dSYM
RM= rm -rf
```

Define macros for later use

```
### Source and object section ########################################
SRCF90= ModSample.f90 main.f90
```

List source files (modules first)

```
### Pattern rules ####################################################
%.o: %.f90
        $(FC) $(FFLAGS) -c $< -o $@

OBJF90= $(SRCF90:.f90=.o)
```

Explain to `make` how to produce object files from source files

```
### Target section ###################################################
$(EXEC): $(OBJF90)
        $(FC) $(OBJF90) $(LDFLAGS) -o $@

.PHONY: clean
clean:
        $(RM) $(OBJF90) *.$(MODEXT) $(EXEC) $(RIFFRAFF)
```

Explain to `make` what to do

```
### Dependencies section #############################################
main.o: ModSample.o
```

Explain to `make` how your files depend on each other

Makefiles

```
### Macros ###########################################################################
SHELL= /bin/sh
EXEC= mycode
FC= gfortran
FFLAGS= -g -Wall -Wextra -fbounds-check
#FFLAGS= -O3
MODEXT= mod
RIFFRAFF= *.bak *.bck *.ckp *.dSYM
RM= rm -rf

### Source and object section ########################################################
SRCF90= ModSample.f90 main.f90

### Pattern rules ####################################################################
%.o: %.f90
        $(FC) $(FFLAGS) -c $< -o $@

OBJF90= $(SRCF90:.f90=.o)

### Target section ###################################################################
$(EXEC): $(OBJF90)
        $(FC) $(OBJF90) $(LDFLAGS) -o $@

.PHONY: clean
clean:
        $(RM) $(OBJF90) *.$(MODEXT) $(EXEC) $(RIFFRAFF)

### Dependencies section #############################################################
main.o: ModSample.o
```

Specify executable name

Specify compiler

Specify compiler options

If you add or remove source files from your code, edit this list accordingly. If you need more lines, terminate a line with \ and continue with a tab on the next line

If you add or remove source files from your code, edit this list accordingly

Overview

- Structuring a program
- Functions
- Makefiles
- Subroutines
- Functions & subroutines: discussion
- Modules
- Arrays
  - Dynamic memory allocation
  - Arrays as arguments
- Additional topics

Subroutines

- Dummy argument(s) to functions should not be modified
- Hence functions are limited to returning single result
- To avoid this restriction, use subroutines
- Example: Computing roots of quadratic function

```
PROGRAM main

IMPLICIT NONE

REAL :: a,b,c
COMPLEX :: r1,r2

WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c

CALL SolveQuadratic(a,b,c,r1,r2)

WRITE(*,*) 'Solutions:',r1,r2

END PROGRAM main
```

Get coefficients

Call subroutine to compute roots

Write roots

Subroutines

- As with `cbrt` function discussed earlier, can treat `SolveQuadratic` as internal or external procedure
- First, take closer look at `SolveQuadratic`

Subroutines

```
SUBROUTINE SolveQuadratic(a,b,c,r1,r2)

IMPLICIT NONE

REAL, INTENT(IN) :: a,b,c
COMPLEX, INTENT(OUT) :: r1,r2

REAL :: d,q
```

← Declaration section

```
d = b**2 - 4.0*a*c

IF ( d < 0.0 ) THEN
  q  = SQRT(-d)
  r1 = CMPLX(-0.5*b/a, 0.5*q/a)
  r2 = CMPLX(-0.5*b/a,-0.5*q/a)
ELSE
  q  = -0.5*(b + SIGN(1.0,b)*SQRT(d))
  r1 = CMPLX(q/a,0.0)
  r2 = CMPLX(c/q,0.0)
END IF ! d
```

← Executable section

```
END SUBROUTINE SolveQuadratic
```

← Termination section

Subroutines

```
SUBROUTINE SolveQuadratic(a,b,c,r1,r2)

IMPLICIT NONE

REAL, INTENT(IN) :: a,b,c
COMPLEX, INTENT(OUT) :: r1,r2

REAL :: d,q

d = b**2 - 4.0*a*c

IF ( d < 0.0 ) THEN
  q  = SQRT(-d)
  r1 = CMPLX(-0.5*b/a, 0.5*q/a)
  r2 = CMPLX(-0.5*b/a,-0.5*q/a)
ELSE
  q  = -0.5*(b + SIGN(1.0,b)*SQRT(d))
  r1 = CMPLX(q/a,0.0)
  r2 = CMPLX(c/q,0.0)
END IF ! d

END SUBROUTINE SolveQuadratic
```

Name of subroutine and dummy arguments

Declaration of dummy arguments

Declaration of local variables

Recommendation:
Separate declarations into
- input arguments
- output arguments
- local variables

End of subroutine

Subroutines: `INTENT(OUT)` Attribute

```
SUBROUTINE SolveQuadratic(a,b,c,r1,r2)

IMPLICIT NONE

REAL, INTENT(IN) :: a,b,c
COMPLEX, INTENT(OUT) :: r1,r2

REAL :: d,q

d = b**2 - 4.0*a*c

IF ( d < 0.0 ) THEN
  q  = SQRT(-d)
  r1 = CMPLX(-0.5*b/a, 0.5*q/a)
  r2 = CMPLX(-0.5*b/a,-0.5*q/a)
ELSE
  q  = -0.5*(b + SIGN(1.0,b)*SQRT(d))
  r1 = CMPLX(q/a,0.0)
  r2 = CMPLX(c/q,0.0)
END IF ! d

END SUBROUTINE SolveQuadratic
```

`INTENT(OUT)` indicates that these dummy arguments become undefined on entry

It does not mean that compiler should check that they will be set!

If these dummy arguments are not set in the subroutine, they will be undefined on exit also!

Subroutines

- Now, let's take a look at how we can "tell" the main program about `SolveQuadratic`
- Discussed two options earlier:
  - Internal procedure: Use `CONTAINS` statement
  - External procedure: Separate files

Subroutines: Internal Procedure

```
PROGRAM main

IMPLICIT NONE

REAL :: a,b,c
COMPLEX :: r1,r2

WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c

CALL SolveQuadratic(a,b,c,r1,r2)

WRITE(*,*) 'Solutions:',r1,r2

CONTAINS

SUBROUTINE SolveQuadratic(a,b,c,r1,r2)

…

END SUBROUTINE SolveQuadratic

END PROGRAM main
```

← Main program

Subroutines: Internal Procedure

```
PROGRAM main

IMPLICIT NONE

REAL :: a,b,c
COMPLEX :: r1,r2

WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c

CALL SolveQuadratic(a,b,c,r1,r2)

WRITE(*,*) 'Solutions:',r1,r2

CONTAINS

SUBROUTINE SolveQuadratic(a,b,c,r1,r2)

…

END SUBROUTINE SolveQuadratic

END PROGRAM main
```

CONTAINS statement

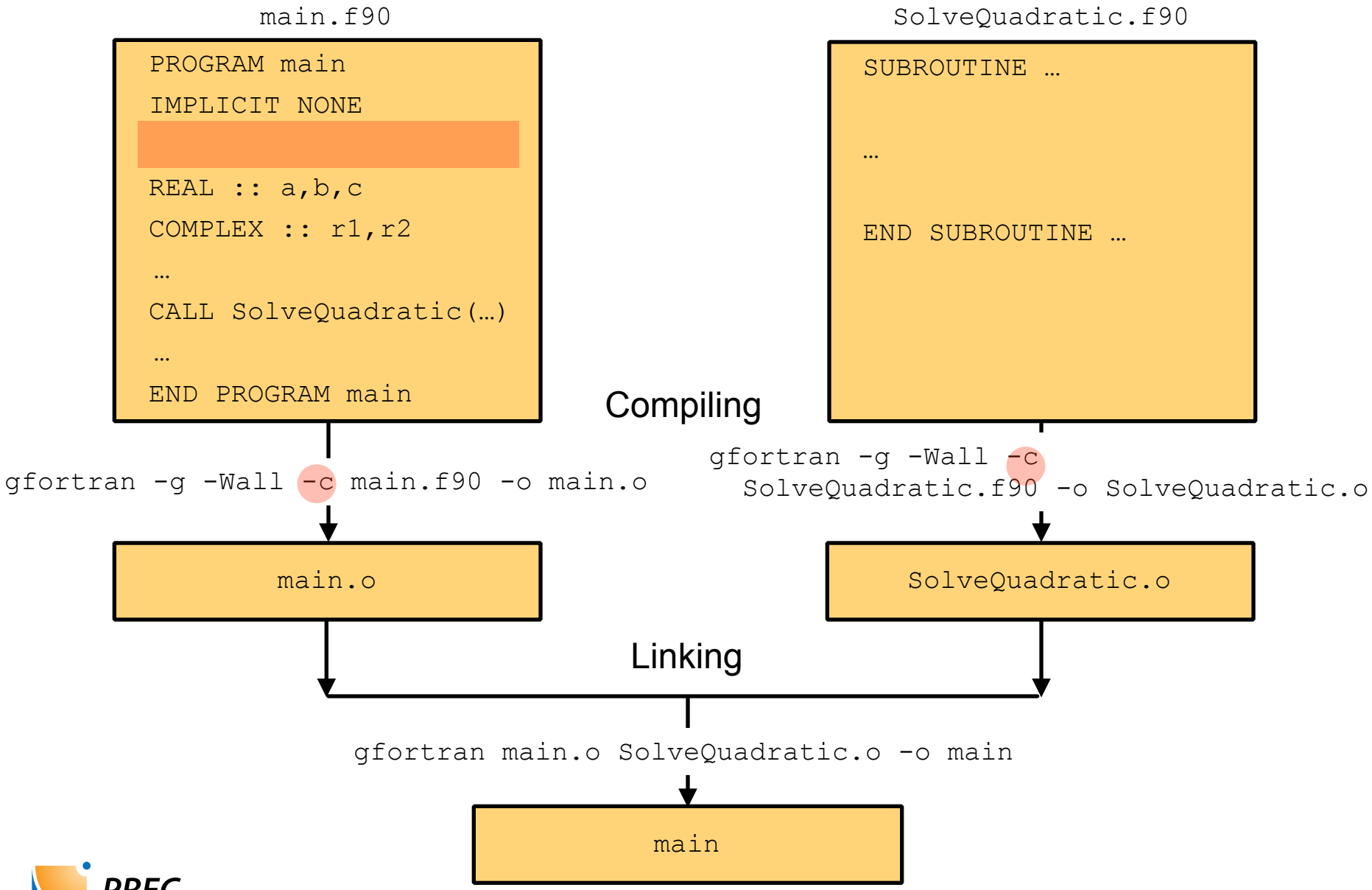
Subroutines: Internal Procedure

```
PROGRAM main

IMPLICIT NONE

REAL :: a,b,c
COMPLEX :: r1,r2

WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c

CALL SolveQuadratic(a,b,c,r1,r2)

WRITE(*,*) 'Solutions:',r1,r2

CONTAINS

SUBROUTINE SolveQuadratic(a,b,c,r1,r2)

…

END SUBROUTINE SolveQuadratic

END PROGRAM main
```

← Subroutine

Note: Subroutine appears before `END PROGRAM` statement

Subroutines: External Procedure



Overview

- Structuring a program
- Functions
- Makefiles
- Subroutines
- Functions & subroutines: discussion
- Modules
- Arrays
  - Dynamic memory allocation
  - Arrays as arguments
- Additional topics

Functions & Subroutines: Discussion

- Functions and subroutines are indispensable for writing modular programs: Use them extensively
- Guidelines for writing one or the other:

Write a `FUNCTION` when…	… you have one result/output
Write a `SUBROUTINE` when…	… you have more than one result/output

- Advantages of writing modular programs:
  - Program easier to understand
  - Program easier to test/debug
  - Parts of program can be reused/shared
  - …
- With well-designed and documented interface, procedure internals are not important

Functions & Subroutines: Discussion

- Personal recommendations:
  - Keep function names short (like `SIN`, `SQRT`) to keep expressions short
  - Choose subroutine names that clearly indicate objective:

```
SolveQuadratic
SolveCubic
ComputeFluxes
ComputeSolutionError
IterateJacobi
IterateGaussSeidel
```

  - Above all, use a consistent naming scheme!

Functions & Subroutines: Discussion

- Keep in mind the following about `INTENT` attributes:

`INTENT(IN)`	• Argument is defined on entry to procedure • Argument will not be changed in procedure • Compiler will issue error if argument is changed
`INTENT(OUT)`	• Argument becomes undefined on entry to the procedure • Argument will be undefined on exit unless it is set in the procedure • It is up to you to set the value of the argument! • The compiler may not warn you if the argument is not defined!
`INTENT(INOUT)`	• Argument is defined on entry to procedure • Argument can be changed in procedure

Functions & Subroutines: Discussion

- What are advantages/disadvantages of internal and external procedures?

	Advantages	Disadvantages
Internal procedure (with `CONTAINS` statement)	• Simplicity: no need to compile and link multiple files	• Single-file program may become long
External procedure (separate files)	• Separate files easier to keep short and easy to understand	• Need to compile and link multiple files

- There is another, very important advantage to internal procedures…
- Consider example on following slide

Functions & Subroutines: Discussion

```
PROGRAM main
IMPLICIT NONE
REAL :: a,b,c
COMPLEX :: r1,r2
WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c
CALL SolveQuadratic(a,b,c,r1,r2)
WRITE(*,*) 'Solutions:',r1,r2
CONTAINS
SUBROUTINE SolveQuadratic(a,b,c,r1,r2)
IMPLICIT NONE
REAL, INTENT(IN) :: a,b,c
COMPLEX, INTENT(OUT) :: r1,r2
REAL :: d,q
d = b**2 - 4.0*a*c
IF ( d < 0.0 ) THEN
  q  = SQRT(-d)
  r1 = CMPLX(-0.5*b/a, 0.5*q/a)
  r2 = CMPLX(-0.5*b/a,-0.5*q/a)
ELSE
  q  = -0.5*(b + SIGN(1.0,b)*SQRT(d))
  r1 = CMPLX(q/a,0.0)
  r2 = CMPLX(c/q,0.0)
END IF ! d
END SUBROUTINE SolveQuadratic
END PROGRAM main
```

- Consider first case of internal procedure as shown on left
- Compiles and runs successfully

Functions & Subroutines: Discussion

```
PROGRAM main
IMPLICIT NONE
REAL :: a,b,c
COMPLEX :: r1,r2
WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c
CALL SolveQuadratic(a,b,c,r1,r2)
WRITE(*,*) 'Solutions:',r1,r2
CONTAINS
SUBROUTINE SolveQuadratic(a,b,c,r1,r2)
IMPLICIT NONE
REAL, INTENT(IN) :: a,b,c
COMPLEX, INTENT(OUT) :: r1,r2
REAL :: d,q
d = b**2 - 4.0*a*c
IF ( d < 0.0 ) THEN
  q  = SQRT(-d)
  r1 = CMPLX(-0.5*b/a, 0.5*q/a)
  r2 = CMPLX(-0.5*b/a,-0.5*q/a)
ELSE
  q  = -0.5*(b + SIGN(1.0,b)*SQRT(d))
  r1 = CMPLX(q/a,0.0)
  r2 = CMPLX(c/q,0.0)
END IF ! d
END SUBROUTINE SolveQuadratic
END PROGRAM main
```

- Now assume we made mistake and accidentally forgot to pass third argument
- Very common mistake!

Functions & Subroutines: Discussion

```
PROGRAM main
IMPLICIT NONE
REAL :: a,b,c
COMPLEX :: r1,r2
WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c
CALL SolveQuadratic(a,b,r1,r2)
WRITE(*,*) 'Solutions:',r1,r2
CONTAINS
SUBROUTINE SolveQuadratic(a,b,c,r1,r2)
IMPLICIT NONE
REAL, INTENT(IN) :: a,b,c
COMPLEX, INTENT(OUT) :: r1,r2
REAL :: d,q
d = b**2 - 4.0*a*c
IF ( d < 0.0 ) THEN
  q  = SQRT(-d)
  r1 = CMPLX(-0.5*b/a, 0.5*q/a)
  r2 = CMPLX(-0.5*b/a,-0.5*q/a)
ELSE
  q  = -0.5*(b + SIGN(1.0,b)*SQRT(d))
  r1 = CMPLX(q/a,0.0)
  r2 = CMPLX(c/q,0.0)
END IF ! d
END SUBROUTINE SolveQuadratic
END PROGRAM main
```

- This is clearly wrong because argument lists do not match

Functions & Subroutines: Discussion

```
PROGRAM main
IMPLICIT NONE
REAL :: a,b,c
COMPLEX :: r1,r2
WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c
CALL SolveQuadratic(a,b,r1,r2)
WRITE(*,*) 'Solutions:',r1,r2
CONTAINS
SUBROUTINE SolveQuadratic(a,b,c,r1,r2)
IMPLICIT NONE
REAL, INTENT(IN) :: a,b,c
COMPLEX, INTENT(OUT) :: r1,r2
REAL :: d,q
d = b**2 - 4.0*a*c
IF ( d < 0.0 ) THEN
  q  = SQRT(-d)
  r1 = CMPLX(-0.5*b/a, 0.5*q/a)
  r2 = CMPLX(-0.5*b/a,-0.5*q/a)
ELSE
  q  = -0.5*(b + SIGN(1.0,b)*SQRT(d))
  r1 = CMPLX(q/a,0.0)
  r2 = CMPLX(c/q,0.0)
END IF ! d
END SUBROUTINE SolveQuadratic
END PROGRAM main
```

- Because procedure is internal, compiler can compare arguments lists
- Inconsistent argument lists lead to compiler error:

```
gfortran -g -Wall main.f90 -o main
main.f90:12.26:

  CALL SolveQuadratic(a,b,r1,r2)
                           1
Error: Type mismatch in argument 'c'
at (1); passed COMPLEX(4) to REAL(4)
```

Functions & Subroutines: Discussion

```
PROGRAM main
IMPLICIT NONE
REAL :: a,b,c
COMPLEX :: r1,r2
WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c
CALL SolveQuadratic(a,b,r1,r2)
WRITE(*,*) 'Solutions:',r1,r2
END PROGRAM main
```

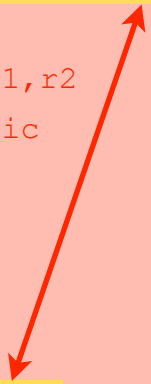
```
SUBROUTINE SolveQuadratic(a,b,c,r1,r2)
IMPLICIT NONE
REAL, INTENT(IN) :: a,b,c
COMPLEX, INTENT(OUT) :: r1,r2
REAL :: d,q
d = b**2 - 4.0*a*c
IF ( d < 0.0 ) THEN
  q  = SQRT(-d)
  r1 = CMPLX(-0.5*b/a, 0.5*q/a)
  r2 = CMPLX(-0.5*b/a,-0.5*q/a)
ELSE
  q  = -0.5*(b + SIGN(1.0,b)*SQRT(d))
  r1 = CMPLX(q/a,0.0)
  r2 = CMPLX(c/q,0.0)
END IF ! d
END SUBROUTINE SolveQuadratic
```

- Now consider case of external procedure
- Compiler cannot compare argument lists
- Code compiles successfully and gives wrong answer!

- Problem is implicit interface of procedure

Functions & Subroutines: Discussion

```
PROGRAM main
IMPLICIT NONE
INTERFACE
  SUBROUTINE SolveQuadratic(a,b,c,r1,r2)
    REAL, INTENT(IN) :: a,b,c
    COMPLEX, INTENT(OUT) :: r1,r2
  END SUBROUTINE SolveQuadratic
END INTERFACE
REAL :: a,b,c
COMPLEX :: r1,r2
WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c
CALL SolveQuadratic(a,b,r1,r2)
WRITE(*,*) 'Solutions:',r1,r2
END PROGRAM main
```



```
SUBROUTINE SolveQuadratic(a,b,c,r1,r2)
IMPLICIT NONE
REAL, INTENT(IN) :: a,b,c
COMPLEX, INTENT(OUT) :: r1,r2
…
…
…
END SUBROUTINE SolveQuadratic
```

- Problem can be solved by providing explicit interface
- Inconsistent argument lists lead to compiler error:

```
gfortran -g -Wall -c main.f90 -o main.o
main.f90:19.26:

  CALL SolveQuadratic(a,b,r1,r2)
                              1
Error: Type mismatch in argument 'c' at
(1); passed COMPLEX(4) to REAL(4)
```

- Will not discuss explicit interfaces in detail here since there exists a better way…

Functions & Subroutines: Discussion

- What are advantages/disadvantages of internal and external procedures?

	Advantages	Disadvantages
Internal procedure (with `CONTAINS` statement)	• Simplicity: no need to compile and link multiple files • Interfaces are always explicit	• Single-file program may become long • Inconvenient for sharing
External procedure (separate files)	• Separate files easier to keep short and easy to understand • Easy to share	• Need to compile and link multiple files • Explicit interfaces must be provided

Overview

- Structuring a program
- Functions
- Makefiles
- Subroutines
- Functions & subroutines: discussion
- Modules
- Arrays
  - Dynamic memory allocation
  - Arrays as arguments
- Additional topics

Modules

- Modules collect variables, data-type definitions, and procedures
- Advantages:
  - Simplify writing of modular programs
  - Module procedures automatically have explicit interfaces
  - …

Modules

```
MODULE ModSolvePoly
```

← Name of module (up to 31 characters)

```
CONTAINS
```

← CONTAINS statement indicates that procedures follow

```
SUBROUTINE SolveQuadratic(a,b,c,r1,r2)
IMPLICIT NONE
REAL, INTENT(IN) :: a,b,c
COMPLEX, INTENT(OUT) :: r1,r2
REAL :: d,q
d = b**2 - 4.0*a*c
IF ( d < 0.0 ) THEN
  q  = SQRT(-d)
  r1 = CMPLX(-0.5*b/a, 0.5*q/a)
  r2 = CMPLX(-0.5*b/a,-0.5*q/a)
ELSE
  q  = -0.5*(b + SIGN(1.0,b)*SQRT(d))
  r1 = CMPLX(q/a,0.0)
  r2 = CMPLX(c/q,0.0)
END IF ! d
END SUBROUTINE SolveQuadratic
```

← Subroutine

```
END MODULE ModSolvePoly
```

← End of module

Modules

- Accessing contents of module is known as use association:

```
PROGRAM main

USE ModSolvePoly

IMPLICIT NONE

REAL :: a,b,c
COMPLEX :: r1,r2

WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c

CALL SolveQuadratic(a,b,c,r1,r2)

WRITE(*,*) 'Solutions:',r1,r2

END PROGRAM main
```


`USE` statements must appear immediately after `PROGRAM`, `FUNCTION`, `SUBROUTINE`, or `MODULE` statement and before `IMPLICIT NONE`

Modules

- Modules are compiled like other Fortran source files:

```
gfortran -g -Wall -c ModSolvePoly.f90 -o ModSolvePoly.o
gfortran -g -Wall -c main.f90 -o main.o
gfortran -g -Wall main.o ModSolvePoly.o -o main
```

- Compilation of `ModSolvePoly.f90` produces

```
ModSolvePoly.o
modsolvepoly.mod
```

- Modules must be compiled before files in which they are used, otherwise get compiler error:

```
main.f90:4.6:

  USE ModSolvePoly
      1
Fatal Error: Can't open module file 'modsolvepoly.mod' for reading at (1): No
such file or directory
```

Modules

- Key point: By placing procedure `SolveQuadratic` in module, it automatically acquires explicit interface
- Therefore, if use incorrect argument list, such as

```
PROGRAM main

USE ModSolvePoly

IMPLICIT NONE

REAL :: a,b,c
COMPLEX :: r1,r2

WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c

CALL SolveQuadratic(a,b,r1,r2)

WRITE(*,*) 'Solutions:',r1,r2

END PROGRAM main
```

get compiler error:

```
main.f90:14.26:

  CALL SolveQuadratic(a,b,r1,r2)
                           1
Error: Type mismatch in argument 'c' at (1); passed COMPLEX(4) to REAL(4)
```

Modules

- Modules give flexibility of splitting program into several files without having to specify explicit interfaces
- `ModSolvePoly.f90` could be made more useful by including additional routines:
  - `SolveCubic(a,b,c,d,r1,r2,r3)`
  - `SolveQuartic(a,b,c,d,e,r1,r2,r3,r4)`
  - Versions with double-precision arithmetic
  - Versions with complex coefficients `a`, `b`, and `c`
  - …

Modules

- So far, only placed procedures in modules
- But modules can also contain variables and constants
- Simple examples:

```
MODULE ModPhysicalConstantsSP

IMPLICIT NONE

REAL, PARAMETER :: PHYS_CONST_SP_G  = 9.81
REAL, PARAMETER :: PHYS_CONST_SP_C0 = 299792458.0

END MODULE ModPhysicalConstantsSP


MODULE ModMathematicalConstantsSP

IMPLICIT NONE

REAL, PARAMETER :: MATH_CONST_SP_PI  = 3.141593
REAL, PARAMETER :: MATH_CONST_SP_PHI = 1.618034

END ModMathematicalConstantsSP
```

- Could be used to get unique values anywhere in code

Modules

- More powerful to combine data and procedures:

```
PROGRAM main

USE ModSample

IMPLICIT NONE

INTEGER :: a

IntegerPublic = 2

a = 10

CALL DummySubroutine(a)

WRITE(*,*) a

END PROGRAM main
```

```
MODULE ModSample

INTEGER :: IntegerPublic

CONTAINS

SUBROUTINE DummySubroutine(a)

IMPLICIT NONE

INTEGER, INTENT(INOUT) :: a

WRITE(*,*) IntegerPublic,a

a = IntegerPublic*a

END SUBROUTINE DummySubroutine

END MODULE ModSample
```

- Output:

```
 2           10
20
```

Modules

- More powerful to combine data and procedures:

```
PROGRAM main

USE ModSample

IMPLICIT NONE

INTEGER :: a

IntegerPublic = 2


a = 10

CALL DummySubroutine(a)

WRITE(*,*) a

END PROGRAM main
```

```
MODULE ModSample

INTEGER :: IntegerPublic

CONTAINS

SUBROUTINE DummySubroutine(a)

IMPLICIT NONE

INTEGER, INTENT(INOUT) :: a

WRITE(*,*) IntegerPublic,a

a = IntegerPublic*a

END SUBROUTINE DummySubroutine

END MODULE ModSample
```

- Output:

```
 2          10
20
```

Modules

- `IntegerPublic` accessible to:
  - All procedures in `ModSample` and
  - All procedures that use-associate `ModSample`
- Therefore, `IntegerPublic` said to be public variable of module `ModSample`
- Consequence: Must be careful with naming variables and constants to avoid conflicts with public variables from modules

Modules

- For example, variable called `IntegerPublic` in `main` would conflict with `IntegerPublic` in `ModSample`...
- … and have led to compiler error:

```
main.f90:8.29:

  INTEGER :: a,IntegerPublic
                           1
main.f90:4.6:

  USE ModSample
      2
Error: Symbol 'integerpublic' at (1) conflicts with symbol
from module 'modsample', use-associated at (2)
```

Modules

- To avoid conflicts:
  - Careful naming of variables and constants in procedures that use-associate variables and constants from modules
  - Restricting access to variables and constants in modules where possible: Make variables public only if necessary
- Fortran enables this with `PUBLIC` and `PRIVATE` attributes for variables and constants declared in modules: data hiding

Modules

- One solution: Public as default

```
PROGRAM main

USE ModSample

IMPLICIT NONE

INTEGER :: a

IntegerPublic = 2

a = 10

CALL DummySubroutine(a)

WRITE(*,*) a

END PROGRAM main
```

```
MODULE ModSample

INTEGER :: IntegerPublic

INTEGER, PRIVATE :: IntegerPrivate

CONTAINS

SUBROUTINE DummySubroutine(a)

IMPLICIT NONE

INTEGER, INTENT(INOUT) :: a

WRITE(*,*) IntegerPublic,a

IntegerPrivate = 20

a = IntegerPublic*IntegerPrivate*a

END SUBROUTINE DummySubroutine

END MODULE ModSample
```

Modules

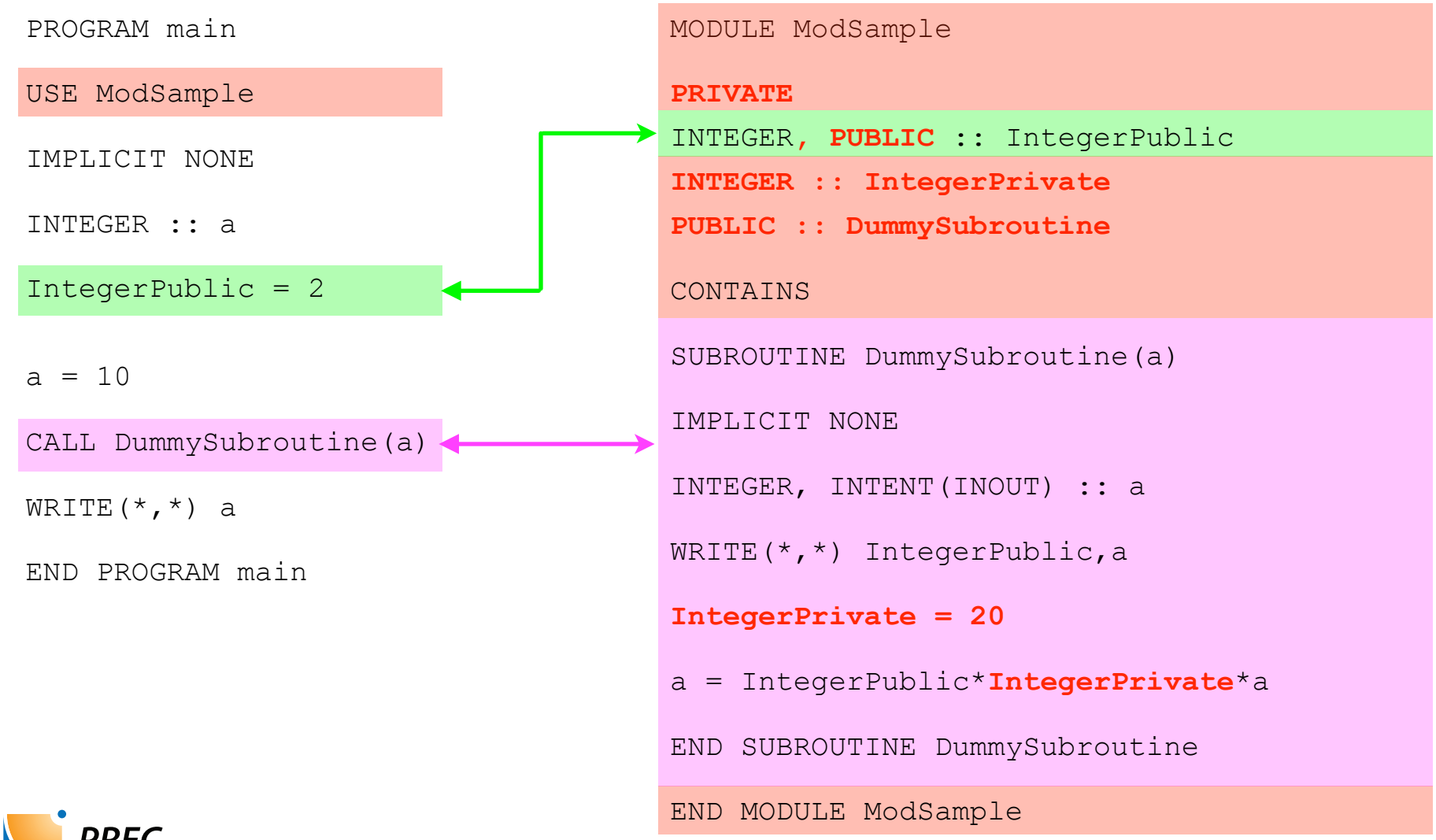
- Better (and recommended) solution: Private as default

```
PROGRAM main

USE ModSample

IMPLICIT NONE

INTEGER :: a

IntegerPublic = 2

a = 10

CALL DummySubroutine(a)

WRITE(*,*) a

END PROGRAM main
```

```
MODULE ModSample

PRIVATE

INTEGER :: IntegerPrivate,IntegerPublic

PUBLIC :: IntegerPublic

PUBLIC :: DummySubroutine

CONTAINS

SUBROUTINE DummySubroutine(a)

IMPLICIT NONE

INTEGER, INTENT(INOUT) :: a

WRITE(*,*) IntegerPublic,a

IntegerPrivate = 20

a = IntegerPublic*IntegerPrivate*a

END SUBROUTINE DummySubroutine

END MODULE ModSample
```

Modules

- Alternative declarations:



```
PROGRAM main

USE ModSample

IMPLICIT NONE

INTEGER :: a

IntegerPublic = 2

a = 10

CALL DummySubroutine(a)

WRITE(*,*) a

END PROGRAM main
```

```
MODULE ModSample

PRIVATE

INTEGER, PUBLIC :: IntegerPublic

INTEGER :: IntegerPrivate

PUBLIC :: DummySubroutine

CONTAINS

SUBROUTINE DummySubroutine(a)

IMPLICIT NONE

INTEGER, INTENT(INOUT) :: a

WRITE(*,*) IntegerPublic,a

IntegerPrivate = 20

a = IntegerPublic*IntegerPrivate*a

END SUBROUTINE DummySubroutine

END MODULE ModSample
```

Modules

- Note that if `DummySubroutine` not `PUBLIC`, that is:

```
PROGRAM main

USE ModSample

IMPLICIT NONE

INTEGER :: a

IntegerPublic = 2

a = 10

CALL DummySubroutine(a)

WRITE(*,*) a

END PROGRAM main
```

```
MODULE ModSample

PRIVATE

INTEGER :: IntegerPrivate,IntegerPublic

PUBLIC :: IntegerPublic

CONTAINS

SUBROUTINE DummySubroutine(a)

IMPLICIT NONE

INTEGER, INTENT(INOUT) :: a

WRITE(*,*) IntegerPublic,a

IntegerPrivate = 20

a = IntegerPublic*IntegerPrivate*a

END SUBROUTINE DummySubroutine

END MODULE ModSample
```

- Compiling this leads to…

Modules

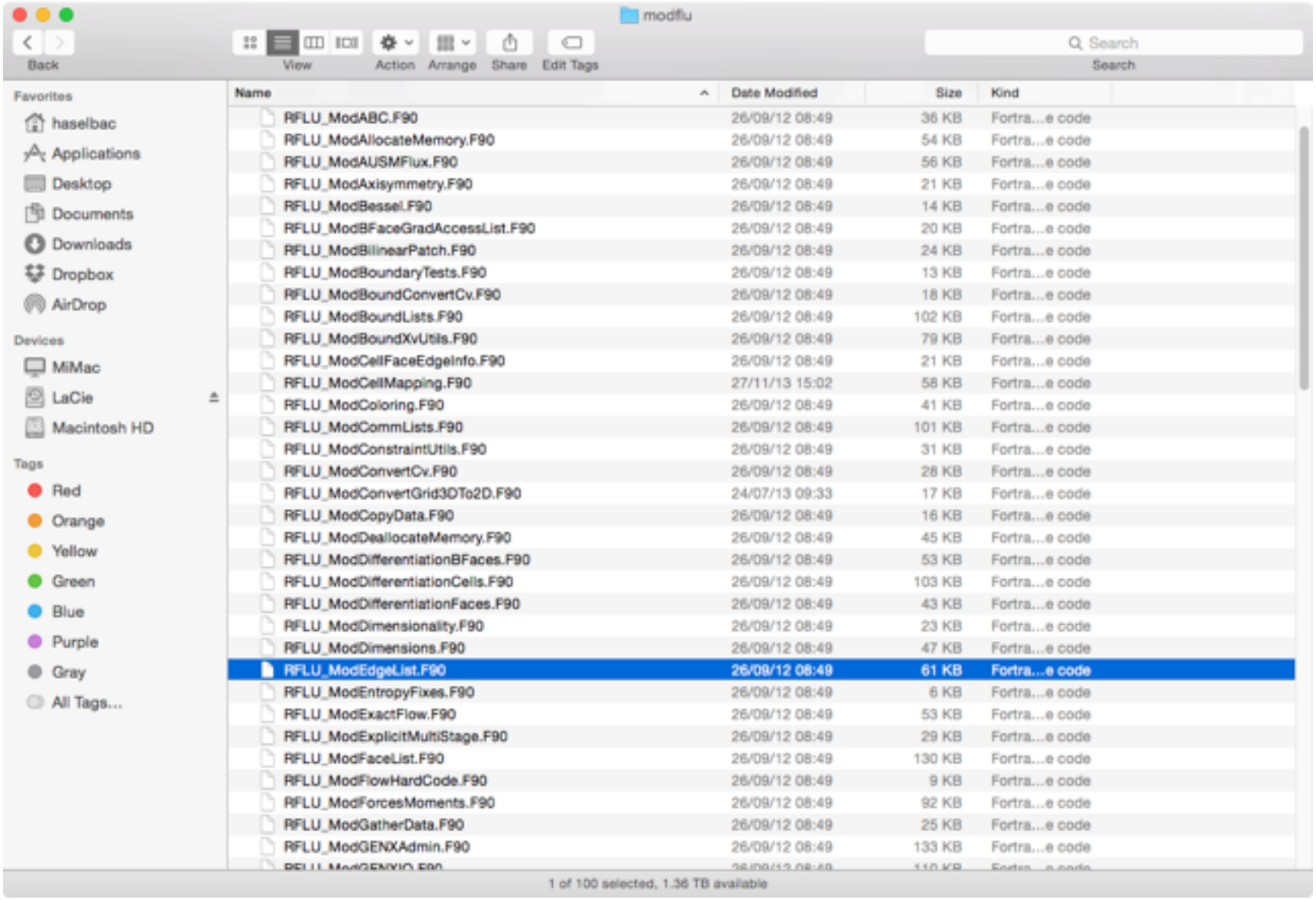
- Compiling this leads to:

```
gfortran -g -Wall  -c ModSample.f90 -o ModSample.o
ModSample.f90:10:0: warning: 'dummysubroutine' defined but
not used [-Wunused-function]
gfortran -g -Wall  -c main.f90 -o main.o
gfortran ModSample.o main.o  -o main
Undefined symbols for architecture x86_64:
  "_dummysubroutine_", referenced from:
      _MAIN__ in main.o
ld: symbol(s) not found for architecture x86_64
```

Modules: Examples

- Module for vectors:
  - Define vector
  - Multiply vector by scalar
  - Scalar product
  - Vector product
  - Angle between vectors
  - Rotation of vector
  - …
- Module for intersections:
  - Two lines
  - Line and circle
  - Two circles
  - …

Modules: Examples



Modules

- Fortran allows definition of new data types: derived data types
- Examples:

```
TYPE :: t_point
  REAL :: x,y
END TYPE t_point

TYPE :: t_line
  REAL :: intercept,slope
END TYPE t_line

TYPE :: t_circle
  REAL :: radius
  REAL :: origin(2)
END TYPE t_circle
```

Recommendation: Use `_t` prefix or postfix to distinguish type definition from variables/constants

- Modules are well suited to contain derived data types and the procedures to manipulate them
- Otherwise, would need type definition in every procedure that needs them

Modules

- Example module:

```
MODULE ModIntersections

IMPLICIT NONE

TYPE :: t_point
  REAL :: x,y
END TYPE t_point

TYPE :: t_line
  REAL :: intercept,slope
END TYPE t_line

TYPE :: t_circle
  REAL :: radius
  REAL :: origin(2)
END TYPE t_circle

CONTAINS

SUBROUTINE IntersectLineCircle(line,circle,point1,point2)
…
END SUBROUTINE IntersectLineCircle

END MODULE ModIntersections
```

```
SUBROUTINE IntersectLineCircle(line,circle,point1,point2)

IMPLICIT NONE

TYPE(t_circle), INTENT(IN) :: circle
TYPE(t_line), INTENT(IN) :: line
TYPE(t_point), INTENT(OUT) :: point1,point2

…

END SUBROUTINE IntersectLineCircle
```

Modules

- The example definitions could be used as follows:

```
PROGRAM typedemo

USE ModIntersections

IMPLICIT NONE

TYPE(t_point) :: ipoint1,ipoint2
TYPE(t_line) :: line
TYPE(t_circle) :: circle

line%intercept =  1.05
line%slope     = -2.31

circle%origin(1) = -0.72
circle%origin(2) =  0.59
circle%radius    =  2.17

CALL IntersectLineCircle(line,circle,ipoint1,ipoint2)

WRITE(*,*) 'Intersection point 1:',ipoint1%x,ipoint1%1
WRITE(*,*) 'Intersection point 1:',ipoint2%x,ipoint2%y

END PROGRAM typedemo
```

Contains derived data type definition and procedures

Declaration with derived data type definition

Set components of derived data types using %

Call subroutine with derived data types as arguments

Overview

- Structuring a program
- Functions
- Makefiles
- Subroutines
- Functions & subroutines: discussion
- Modules
- Arrays
  - Dynamic memory allocation
  - Arrays as arguments
- Additional topics

Arrays: Dynamic Memory Allocation

- Previously discussed arrays with known dimensions
- In practice, array dimensions usually problem-dependent
- Such arrays are declared with

```
REAL, ALLOCATABLE :: a(:,:)
```

or

```
REAL, DIMENSION(:,:), ALLOCATABLE :: a
```

- Once dimensions known, can allocate memory:

```
ALLOCATE(a(m,n),STAT=ef)
IF ( ef /= 0 ) THEN
  WRITE(*,*) 'ERROR!'
  STOP
END IF ! ef
```

`ALLOCATE` returns an integer status variable

Non-zero status variable indicates memory allocation was unsuccessful

Values and meaning of non-zero status variable compiler-dependent

`ef` is short for "error flag"

Arrays: Dynamic Memory Allocation

- If array no longer needed, can be deallocated:

```
DEALLOCATE(a,STAT=ef)
IF ( ef /= 0 ) THEN
  WRITE(*,*) 'ERROR!'
  STOP
END IF ! ef
```

`DEALLOCATE` returns an integer status variable

Non-zero status variable indicates memory deallocation was unsuccessful

Values and meaning of non-zero status variable compiler-dependent

- Before allocation and after deallocation, array elements cannot be accessed!
- Consider following example…

Arrays: Dynamic Memory Allocation

```
PROGRAM test

IMPLICIT NONE

INTEGER :: ef,m,n
REAL, ALLOCATABLE :: a(:,:)

WRITE(*,*) 'Enter m,n:'
READ(*,*) m,n

ALLOCATE(a(m,n),STAT=ef)
IF ( ef /= 0 ) THEN
  WRITE(*,*) 'ERROR!'
  STOP
END IF ! ef

a = 1.0

DEALLOCATE(a,STAT=ef)
IF ( ef /= 0 ) THEN
  WRITE(*,*) 'ERROR!'
  STOP
END IF ! ef

WRITE(*,*) a(1,1)

END PROGRAM test
```

Stepping out-of-bounds leads to run-time error:

```
Enter m,n:
4
2

Program received signal SIGSEGV:
Segmentation fault - invalid memory
reference.

Backtrace for this error:
#0  0x104d492d2
#1  0x104d49a8e
#2  0x7fff875c1f19
#3  0x104e1db4a
#4  0x104e20514
#5  0x104e2104e
#6  0x104d40d87
#7  0x104d40dce
Segmentation fault: 11
```

Compare this with next example…

Arrays: Dynamic Memory Allocation

```
PROGRAM test

IMPLICIT NONE

INTEGER :: ef,m,n
REAL, ALLOCATABLE :: a(:,:)

WRITE(*,*) 'Enter m,n:'
READ(*,*) m,n

ALLOCATE(a(m,n),STAT=ef)
IF ( ef /= 0 ) THEN
  WRITE(*,*) 'ERROR!'
  STOP
END IF ! ef

a = 1.0

WRITE(*,*) a(m+1,n)

DEALLOCATE(a,STAT=ef)
IF ( ef /= 0 ) THEN
  WRITE(*,*) 'ERROR!'
  STOP
END IF ! ef

END PROGRAM test
```

Stepping out-of-bounds leads to run-time error:

```
Enter m,n:
4
2
At line 20 of file test.f90
Fortran runtime error: Index '5' of dimension 1 of array 'a' above upper bound of 4
```

Arrays: Dynamic Memory Allocation

- If unsure whether `a` allocated, use `ALLOCATED(a)` function:
  - Returns logical variable, set to true if `a` allocated, false otherwise
  - Can only be used if `a` declared with `ALLOCATABLE` attribute
- If unsure about dimensions of `a`, use `LBOUND(a,i)` and `UBOUND(a,i)`

Arrays: Dynamic Memory Allocation

- Good programming practice:
  - Always check status variables
  - Write helpful error messages

Arrays: Dynamic Memory Allocation

- How could error handling in adjacent code be improved?

```
PROGRAM allocdealloc

IMPLICIT NONE

INTEGER :: ef,i,j,m,n
REAL, ALLOCATABLE :: a(:,:)

WRITE(*,*) 'Enter m,n:'
READ(*,*) m,n

ALLOCATE(a(m,n),STAT=ef)
IF ( ef /= 0 ) THEN
  WRITE(*,*) 'ERROR!'
  STOP
END IF ! ef

a = 1.0

DEALLOCATE(a,STAT=ef)
IF ( ef /= 0 ) THEN
  WRITE(*,*) 'ERROR!'
  STOP
END IF ! ef

END PROGRAM allocdealloc
```

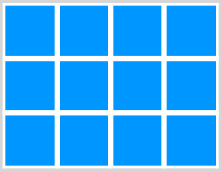
Overview

- Structuring a program
- Functions
- Makefiles
- Subroutines
- Functions & subroutines: discussion
- Modules
- Arrays
  - Dynamic memory allocation
  - Arrays as arguments
- Additional topics

Arrays as Arguments: Background

Array dimension	Number of subscripts; each subscript has lower and upper bound
Array rank	Same as array dimension
Dimension extent	Number of elements in that dimension
Array size	Total number of elements in array
Array shape	Determined by rank and dimension extents

Arrays as Arguments: Background

Array	Shape	Dimension/ Rank	Dimension extents	Size
`a(2,3)`		2	2,3	6
`b(0:1,-1:1)`		2	2,3	6
`c(-1:1,0:3)`		2	3,4	12
`e(5)`		1	5	5
`f(-3:1)`		1	5	5

Arrays as Arguments

- Four types of arrays in Fortran:

	Shape	Use
Explicit-shape array	Explicitly declared	As dummy argument or as local variable
Assumed-shape array	Not given	As dummy argument provided procedure has explicit interface, cannot be used in main program
Automatic array	Explicitly declared, at least one index bound not constant	Not as dummy argument, cannot be used in main program
Deferred-shape array (Allocatable array)	Not given, determined with `ALLOCATE` statement	Not as dummy argument, can be used as local variable

Arrays as Arguments

```
  SUBROUTINE arrayargs(m,n,a,b)

  IMPLICIT NONE

! Arguments
  INTEGER, INTENT(IN) :: m,n
  REAL, INTENT(INOUT) :: a(m,n) ! Explicit-shape array
  REAL, INTENT(INOUT) :: b(:,:) ! Assumed-shape array

! Locals
  INTEGER :: r,s
  REAL :: c(m,n),d(5,n)         ! Automatic arrays
  REAL :: e(5)                  ! Explicit-shape array
  REAL, ALLOCATABLE :: f(:,:,:) ! Deferred-shape array

! Allocate memory
  r = …
  s = …
  ALLOCATE(f(3,r,s),STAT=ef)
  …
  DEALLOCATE(f,STAT=ef)
  …

  END SUBROUTINE arrayargs
```

Arrays as Arguments

```
  SUBROUTINE arrayargs(m,n,a,b)

  IMPLICIT NONE

! Arguments
  INTEGER, INTENT(IN) :: m,n
  REAL, INTENT(INOUT) :: a(m,n) ! Explicit-shape array
  REAL, INTENT(INOUT) :: b(:,:) ! Assumed-shape array

! Locals
  INTEGER :: r,s
  REAL :: c(m,n),d(5,n)         ! Automatic arrays
  REAL :: e(5)                  ! Explicit-shape array
  REAL, ALLOCATABLE :: f(:,:,:) ! Deferred-shape array

! Allocate memory
  r = …
  s = …
  ALLOCATE(f(3,r,s),STAT=ef)
  …
  DEALLOCATE(f,STAT=ef)
  …

  END SUBROUTINE arrayargs
```

Assumed-shape array can only be dummy argument in procedure with explicit interface, so either

1. calling routine includes explicit interface for `arrayargs`, or
2. `arrayargs` is placed in `MODULE` and `USE`d in calling routine

Arrays as Arguments

```
PROGRAM arrayexpshape

IMPLICIT NONE

INTEGER :: a(-3:2,4:5)

a = 1

CALL sub(a)

CONTAINS
```

```
SUBROUTINE sub(b)

INTEGER, INTENT(IN) :: b(:,:)

WRITE(*,*) LBOUND(b,1),UBOUND(b,1)
WRITE(*,*) LBOUND(b,2),UBOUND(b,2)

END SUBROUTINE sub
```

```
END PROGRAM arrayexpshape
```

- Careful when passing array with non-unity lower bounds!
- Output of code on left: Not

```
-3   2
 4   5
```

but

```
 1   6
 1   2
```

- Q: Can you think of a work-around?

Additional Topics

Topic	Brief description	Additional information
Whole-array operations	Perform operations on whole arrays	Chapman (2008): 6.3
Namelist input/output	Simplified input/output of group of variables	Chapman (2008): 14.4 Ellis et al. (1994): 15.7
Binary files	Faster reading/writing, smaller file sizes	Chapman (2008): 14.3
Recursion	Enable procedures to call themselves	Chapman (2008): 13.2 Ellis et al. (1994): 11.5
Object-oriented programming (a little in Fortran 90/95, but mostly in Fortran 2003)	Enhanced modularity, flexibility, abstraction	Chapman (2008): 16
Command-line arguments (Fortran 2003)	Access to command-line arguments	Chapman (2008): 13.11

Additional Topics

Topic	Brief description	Additional information
Parameterized data types	Change representation of data	Chapman (2008): 11 Ellis et al. (1994): 10.2, 14
Optional arguments	Do not pass all arguments to procedures	Chapman (2008): 13.3 Ellis et al. (1994): 11.1
Procedures as arguments	Pass procedure names as arguments to procedures	Chapman (2008): 7.5
Generic procedures	Procedures for different data types	Chapman (2008): 13.5 Ellis et al. (1994): 11.6