Introduction to Fortran 90

F. Salvadore I. Baccarelli G. Amati

CINECA Roma - SCAI Department

Roma, 20-23 January 2014
Part I

A Fortran Survey 1

Outline

Introduction

Fortran Basics

More Fortran Basics

Integer Types and Iterating

More on Compiling and Linking

Homeworks
Formula Translator History

- Developed in the 50s among the earliest high level languages (HLL)
- Widely and rapidly adopted in the area of numerical, scientific, engineering and technical applications
- First standard in 1966: Fortran 66
  - The first of all programming language standards
- Second standard in 1978: Fortran 77
- Third standard in 1991: Fortran 90
  - Adds new, modern features such as structured constructs, array syntax and ADT
  - Extended and revised in 1997: Fortran 95
  - Further extended with published Technical Reports
- Fourth standard in 2004: Fortran 2003
  - Major revision, incorporates TRs, adds many new features (OO!), still not fully supported
- Fifth standard in 2010: Fortran 2008
Fortran General Philosophy

- Strongly oriented to *number crunching*
- Efficient language, highly optimized code
  - Basic data types and operators mapping "naturally" to CPUs
  - Translated by a compiler to machine language
  - Language rules allow for aggressive, automatic optimization
  - Facilities to build new data types from the basic ones
  - Flexible flow control structures mapping the most common numerical computing use cases
- Scientific computing specialized syntax
  - A wealth of math data types and functions available as intrinsics of the language
  - Compact, readable array syntax to operate on many values as a whole
Why Fortran is bad

- Current standard embodies four different language versions,...
- ... all of them still alive in legacy codes
- Non-numeric computing in Fortran is a real pain
- There are more C than Fortran programmers
- GUI and DB accesses are best programmed in C
- C99 partly addressed numerical computing needs
Why Fortran is bad

- Current standard embodies four different language versions,...
- ... all of them still alive in legacy codes
- Non-numeric computing in Fortran is a real pain
- There are more C than Fortran programmers
- GUI and DB accesses are best programmed in C
- C99 partly addressed numerical computing needs

Why Fortran is good

- Fortran is highly tuned for numerical computation
- Fortran is older and more “rigid” than C, compilers optimize better
- Much better than C at managing user defined data types
- Object-oriented features are now part of the language
- Provides facilities for interoperability with C and other languages
Our Aims

- Teach you the fundamentals of modern Fortran
- For both reading (old and new) and writing (new) programs
- Showing common idioms
- Illustrating and demonstrating many of the extensions introduced in the more recent standards
- Illustrating best practices
- Blaming bad ones
- Making you aware of the typical traps

You’ll happen to encounter things we didn’t cover, but it will be easy for you to learn more... or to attend a more advanced course!

A course is not a substitute for a reference manual or a good book!

Neither a substitute for personal practice
Our Aims

- Teach you the fundamentals of modern Fortran
- For both reading (old and new) and writing (new) programs
- Showing common idioms
- Illustrating and demonstrating many of the extensions introduced in the more recent standards
- Illustrating best practices
- Blaming bad ones
- Making you aware of the typical traps
- You’ll happen to encounter things we didn’t cover, but it will be easy for you to learn more... or to attend a more advanced course!
- A course is not a substitute for a reference manual or a good book!
Our Aims

- Teach you the fundamentals of modern Fortran
- For both reading (old and new) and writing (new) programs
- Showing common idioms
- Illustrating and demonstrating many of the extensions introduced in the more recent standards
- Illustrating best practices
- Blaming bad ones
- Making you aware of the typical traps
- You’ll happen to encounter things we didn’t cover, but it will be easy for you to learn more… or to attend a more advanced course!
- A course is not a substitute for a reference manual or a good book!
- Neither a substitute for personal practice
Outline

Introduction

Fortran Basics
- My First Fortran Program
- Compiling and Linking Your First Program
- Making Choices
- More Types and Choices
- Wrapping it Up 1

More Fortran Basics

Integer Types and Iterating

More on Compiling and Linking

Homeworks
Outline

Introduction

Fortran Basics

My First Fortran Program

Compiling and Linking Your First Program

Making Choices

More Types and Choices

Wrapping it Up 1

More Fortran Basics

Integer Types and Iterating

More on Compiling and Linking

Homeworks
! roots of a 2nd degree equation with real coefficients

program second_degree_eq
  implicit none
  real :: delta
  real :: x1, x2
  real :: a, b, c

  print *, 'Solving ax^2+bx+c=0, enter a, b, c:'
  read (*,*) a, b, c

  delta = sqrt(b**2 - 4.0*a*c) ! square root of discriminant
  x1 = -b + delta
  x2 = -b - delta
  x1 = x1/(2.0*a)
  x2 = x2/(2.0*a)

  write(*,*) 'Real roots:', x1, x2

end program second_degree_eq
! roots of a 2nd degree equation with real coefficients

program second_degree_eq
    implicit none
    real :: delta
    real :: x1, x2
    real :: a, b, c

    print *,’Solving ax^2+bx+c=0, enter a, b, c:’
    read (*,*) a, b, c

    delta = sqrt(b**2 - 4.0*a*c) ! square root of discriminant
    x1 = -b + delta
    x2 = -b - delta
    x1 = x1/(2.0*a)
    x2 = x2/(2.0*a)

    write(*,*) ’Real roots:’, x1, x2

end program second_degree_eq
Best practice: do comment your code!

- Variable contents
- Algorithms
- Assumptions
- Tricks

Best practice: do not over-comment your code!

- Obvious comments obfuscate code and annoy readers

Text following ! is ignored up to the end of current line
Text following ! is ignored up to the end of current line

Best practice: do comment your code!
  - Variable contents
  - Algorithms
  - Assumptions
  - Tricks

Best practice: do not over-comment your code!

Obvious comments obfuscate code and annoy readers

! square root of discriminant is a bad example
Text following ![is ignored up to the end of current line](#)

**Best practice: do comment your code!**
- Variable contents
- Algorithms
- Assumptions
- Tricks

**Best practice: do not over-comment your code!**
- Obvious comments obfuscate code and annoy readers
- ![square root of discriminant](#) is a bad example
! roots of a 2nd degree equation with real coefficients

program second_degree_eq
  implicit none
  real :: delta
  real :: x1, x2
  real :: a, b, c

  print *,’Solving ax^2+bx+c=0, enter a, b, c:’
  read (*,*) a, b, c

  delta = sqrt(b**2 - 4.0*a*c)
  x1 = -b + delta
  x2 = -b - delta
  x1 = x1/(2.0*a)
  x2 = x2/(2.0*a)

  write(*,*) ’Real roots:’, x1, x2

end program second_degree_eq
Program Units: Main Program

- Fortran code is organized in program units
  - Main program
  - Procedures (subroutines and functions)
  - Modules
  - More on this later...
Program Units: Main Program

- Fortran code is organized in program units
  - Main program
  - Procedures (subroutines and functions)
  - Modules
  - More on this later...

- The main program (one, and only one!) can’t be dispensed with
  - It’s called automatically to execute the program
  - An optional `program program-name` can appear at the beginning
  - An `end` statement must terminate it, optionally followed by `program` or `program program-name`
Program Units: Main Program

- Fortran code is organized in program units
  - Main program
  - Procedures (subroutines and functions)
  - Modules
  - More on this later...

- The main program (one, and only one!) can’t be dispensed with
  - It’s called automatically to execute the program
  - An optional `program program-name` can appear at the beginning
  - An `end` statement must terminate it, optionally followed by `program` or `program program-name`

- Best practice: always mark unit beginning and ending with its type and name
  - Makes your readers (including you) happier
My First Scientific Program in Fortran

! roots of a 2nd degree equation with real coefficients

program second_degree_eq
  implicit none
  real :: delta
  real :: x1, x2
  real :: a, b, c

  print *,’Solving ax^2+bx+c=0, enter a, b, c:’
  read (*,*) a, b, c

  delta = sqrt(b**2 - 4.0*a*c)
  x1 = -b + delta
  x2 = -b - delta
  x1 = x1/(2.0*a)
  x2 = x2/(2.0*a)

  write(*,*) ’Real roots:’, x1, x2

end program second_degree_eq
Variables

- `real :: x1, x2` declares two variables
  - Named memory locations where values can be stored
  - Declared by specifying a data type, an optional attribute list, and a comma-separated list of names
  - On most CPUs (notably x86), `real` means that `x1` and `x2` host IEEE single precision (i.e. 32 bits) floating point values
Variables

- `real :: x1, x2` declares two variables
  - Named memory locations where values can be stored
  - Declared by specifying a data type, an optional attribute list, and a comma-separated list of names
  - On most CPUs (notably x86), `real` means that `x1` and `x2` host IEEE single precision (i.e. 32 bits) floating point values

- A legal *name* must be used for a variable:
  - Permitted characters: `a-z, A-Z, 0-9, _`
  - The first one cannot be a digit
    (e.g. `x1` is a valid name, `1x` is not)
  - At most 31 characters are permitted (63 in Fortran 2003)
  - A good advice: do not exceed 31 characters in a name
- `real :: x1, x2` declares two variables
  - Named memory locations where values can be stored
  - Declared by specifying a data type, an optional attribute list, and a comma-separated list of names
  - On most CPUs (notably x86), `real` means that `x1` and `x2` host IEEE single precision (i.e. 32 bits) floating point values

- A legal *name* must be used for a variable:
  - Permitted characters: `a-z, A-Z, 0-9, _`
  - The first one cannot be a digit
    (e.g. `x1` is a valid name, `1x` is not)
  - At most 31 characters are permitted (63 in Fortran 2003)
  - A good advice: do not exceed 31 characters in a name

- Beware: Fortran is CaSe inSenSITIVE!
By default, Fortran assumes that variables not appearing in any declaration statement are implicitly declared as follows:

- Variables whose name starts with `A` - `H` and `O` - `Z` are reals.
- Variables whose name starts with `I`, `J`, `K`, `L`, `M`, `N` are integers.

Best practice: it is strongly recommended to turn off implicit declarations with `implicit none`, at the beginning of each program unit. Improves readability and clarity: each variable has its type declared. Mistyped names can be caught by the compiler as undeclared variables.
Implicit Declarations

- By default, Fortran assumes that variables not appearing in any declaration statement are implicitly declared as follows:
  - Variables whose name starts with A - H and O - Z are reals
  - Variables whose name starts with I, J, K, L, M, N are integers

Best practice: it is strongly recommended to turn off implicit declarations with `implicit none`, at the beginning of each program unit. Improves readability and clarity: each variable has its type declared. Mistyped names can be caught by the compiler as undeclared variables.
Implicit Declarations

- By default, Fortran assumes that variables not appearing in any declaration statement are implicitly declared as follows:
  - Variables whose name starts with `A - H` and `O - Z` are reals
  - Variables whose name starts with `I, J, K, L, M, N` are integers
- Best practice: it is strongly recommended to turn off implicit declarations with `implicit none`, at the beginning of each program unit.
  - Improves readability and clarity: each variable has its type declared
  - Mistyped names can be caught by the compiler as undeclared variables
Implicit Declarations

- By default, Fortran assumes that variables not appearing in any declaration statement are implicitly declared as follows:
  - Variables whose name starts with A - H and O - Z are reals
  - Variables whose name starts with I, J, K, L, M, N are integers

- Best practice: it is strongly recommended to turn off implicit declarations with `implicit none`, at the beginning of each program unit
  - Improves readability and clarity: each variable has its type declared
  - Mistyped names can be caught by the compiler as undeclared variables
program second_degree_eq
    implicit none
    real :: delta
    real :: x1, x2
    real :: a, b, c

    print *, 'Solving ax^2+bx+c=0, enter a, b, c:'
    read (*,*) a, b, c

    delta = sqrt(b**2 - 4.0*a*c)
    x1 = -b + delta
    x2 = -b - delta
    x1 = x1/(2.0*a)
    x2 = x2/(2.0*a)

    write(*,*) 'Real roots:', x1, x2

end program second_degree_eq
A Few First Words on I/O

- The bare minimum: textual input output from/to the user terminal
  - `read(*,*)` and `read *`, `read`
  - `write(*,*)` and `print *`, `write`

- Enough for now, disregard details
A Few First Words on I/O

- The bare minimum: textual input output from/to the user terminal
  - `read(*,*)` and `read *`, `read`
  - `write(*,*)` and `print *`, `write`
- These very common idioms perform formatted, list directed I/O
  - *Formatted* means that translation from/to user readable text to/from internal binary formats is performed
  - *List directed* means that external and internal formats are chosen according to the type of each variable or value on the list

Enough for now, disregard details
A Few First Words on I/O

- The bare minimum: textual input output from/to the user terminal
  - `read(*,*)` and `read *`, `read`
  - `write(*,*)` and `print *`, `write`
- These very common idioms perform formatted, list directed I/O
  - *Formatted* means that translation from/to user readable text to/from internal binary formats is performed
  - *List directed* means that external and internal formats are chosen according to the type of each variable or value on the list
- `read(*,*)` and `read *`, `read` are equivalent
A Few First Words on I/O

- The bare minimum: textual input output from/to the user terminal
  - `read(*,*)` and `read *`, `read`
  - `write(*,*)` and `print *`, `write`
- These very common idioms perform formatted, list directed I/O
  - *Formatted* means that translation from/to user readable text to/from internal binary formats is performed
  - *List directed* means that external and internal formats are chosen according to the type of each variable or value on the list
- `read(*,*)` and `read *`, are equivalent
- `write(*,*)` and `print *`, are equivalent

Enough for now, disregard details
A Few First Words on I/O

- The bare minimum: textual input output from/to the user terminal
  - `read(*,*)` and `read *`, `read`
  - `write(*,*)` and `print *`, `write`

- These very common idioms perform formatted, list directed I/O
  - *Formatted* means that translation from/to user readable text to/from internal binary formats is performed
  - *List directed* means that external and internal formats are chosen according to the type of each variable or value on the list

- `read(*,*)` and `read *`, are equivalent
- `write(*,*)` and `print *`, are equivalent
- Enough for now, disregard details
! roots of a 2nd degree equation with real coefficients

program second_degree_eq
    implicit none
    real :: delta
    real :: x1, x2
    real :: a, b, c

    print *,’Solving ax^2+bx+c=0, enter a, b, c:’
    read (*,*) a, b, c

    delta = sqrt(b**2 - 4.0*a*c)
    x1 = -b + delta
    x2 = -b - delta
    x1 = x1/(2.0*a)
    x2 = x2/(2.0*a)

    write(*,*) ’Real roots:’, x1, x2

end program second_degree_eq
Most of program work takes place in statements and expressions.

Operators compute values from terms: +, -, *, / behave like in "human" arithmetic. So do unary -, (, and ). ** is the exponentiation operator. sqrt() is an intrinsic function returning the square root of its argument. 

x1 = x1 + delta is a statement assigning the value of expression x1 + delta to variable x1. By the way, expressions can be passed as argument to functions, as to sqrt(): their value will be computed and passed to the function.
Statements, Expressions and Operators

- Most of program work takes place in statements and expressions
- Operators compute values from terms
  - +, −, * (multiplication), and / behave like in “human” arithmetic
  - So do unary −, (, and )
  - ** is the exponentiation operator

sqrt() is an intrinsic function returning the square root of its argument

x1 = x1 + delta is a statement assigning the value of expression x1 + delta to variable x1

By the way, expressions can be passed as argument to functions, as to sqrt(): their value will be computed and passed to the function
Most of program work takes place in statements and expressions

Operators compute values from terms
- +, −, ∗ (multiplication), and / behave like in “human” arithmetic
- So do unary −, (, and )
- ** is the exponentiation operator

sqrt() is an intrinsic function returning the square root of its argument
Most of program work takes place in statements and expressions. Operators compute values from terms, including:

- $+$, $-$, $\times$ (multiplication), and $/$ behave like in “human” arithmetic
- So do unary $-$, $($, and $)$
- $\star\star$ is the exponentiation operator

$sqrt()$ is an intrinsic function returning the square root of its argument.

$x1 = x1 + delta$ is a statement assigning the value of expression $x1 + delta$ to variable $x1$. By the way, expressions can be passed as arguments to functions, as to $sqrt()$: their value will be computed and passed to the function.
Most of program work takes place in statements and expressions.

Operators compute values from terms:

- +, −, ∗ (multiplication), and / behave like in “human” arithmetic
- So do unary −, (, and )
- ** is the exponentiation operator

`sqrt()` is an intrinsic function returning the square root of its argument.

`x1 = x1 + delta` is a statement assigning the value of expression `x1 + delta` to variable `x1`.

By the way, expressions can be passed as argument to functions, as to `sqrt()`: their value will be computed and passed to the function.
What a Compiler Is

- Fortran lets you write programs in a high-level, human-readable language
- Computer CPUs do not directly understand this language
- You need to translate your code into machine-level instructions for your CPU architecture
- Compilers take care of that translation and generate machine code that can be actually executed by a CPU
What a Compiler Does

- Compilers are sophisticated tools, made up of many components
- When compiler is invoked to generate executable code, three main steps are performed:
  1. parsing of source files, various kinds of analysis and transformations, optimization and *assembly* files creation
  2. machine-code generation and object file creation
      - an object file is an organized collection of all symbols (variables, functions...) used or referenced in the code
  3. linking and executable creation
- Options are provided to execute each step separately, take a look at the manual of your favourite compiler, there’s a lot to learn!
Compile your first Fortran program!

- GNU compiler collection includes `gfortran` compiler, supporting Fortran 95 and several features of the 2003 standard (GNU 4.8)

  Compile with:
  ```
  user@cineca$> gfortran second_degree_eq.f90
  ```

  An executable file named `a.out` (or `a.exe` under Windows) will be generated.

  Run the program under GNU/Linux with:
  ```
  user@cineca$> ./a.out
  ```

  or under Windows with:
  ```
  C:\Documents and Settings\user> a.exe
  ```
Compile your first Fortran program!

- GNU compiler collection includes `gfortran` compiler, supporting Fortran 95 and several features of the 2003 standard (GNU 4.8)
- Many more available on the market (Intel, PGI, Pathscale, IBM XL Fortran, Sun Studio Fortran, Lahey, NAG, etc)

Compile with:
```
user@cineca$> gfortran second_degree_eq.f90
```
An executable file named `a.out` (a.exe under Windows) will be generated.

Run the program under GNU/Linux with:
```
user@cineca$> ./a.out
```
or under Windows with:
```
C:\Documents and Settings\user> a.exe
```
Compile your first Fortran program!

- GNU compiler collection includes **gfortran** compiler, supporting Fortran 95 and several features of the 2003 standard (GNU 4.8)
- Many more available on the market (Intel, PGI, Pathscale, IBM XL Fortran, Sun Studio Fortran, Lahey, NAG, etc)
- Let’s use **gfortran** to compile and run our examples and exercises

Compile with:

    user@cineca$> gfortran second_degree_eq.f90

An executable file named **a.out** will be generated

Run the program under GNU/Linux with:

    user@cineca$> ./a.out

or under Windows with:

    C:\Documents and Settings\user> a.exe
Compile your first Fortran program!

- GNU compiler collection includes **gfortran** compiler, supporting Fortran 95 and several features of the 2003 standard (GNU 4.8)

- Many more available on the market (Intel, PGI, Pathscale, IBM XL Fortran, Sun Studio Fortran, Lahey, NAG, etc)

- Let’s use **gfortran** to compile and run our examples and exercises
  - Compile with:
    ```bash
    user@cineca$> gfortran second_degree_eq.f90
    ```
    An executable file named **a.out** (**a.exe** under Windows) will be generated
Compile your first Fortran program!

- GNU compiler collection includes **gfortran** compiler, supporting Fortran 95 and several features of the 2003 standard (GNU 4.8)
- Many more available on the market (Intel, PGI, Pathscale, IBM XL Fortran, Sun Studio Fortran, Lahey, NAG, etc)
- Let’s use **gfortran** to compile and run our examples and exercises
  - Compile with:
    ```
    user@cineca$> gfortran second_degree_eq.f90
    ```
    An executable file named **a.out** (**a.exe** under Windows) will be generated
  - Run the program under GNU/Linux with:
    ```
    user@cineca$> ./a.out
    ```
    or under Windows with:
    ```
    C:\Documents and Settings\user> a.exe
    ```
Do You Like IDEs? Geany
! roots of a 2nd degree equation with real coefficients

program second_degree_eq
    implicit none
    real :: delta
    real :: x1, x2
    real :: a, b, c

    print *, 'Solving ax^2+bx+c=0, enter a, b, c:'
    read (*,*) a, b, c

    delta = sqrt(b**2 - 4.0*a*c)
    x1 = -b + delta
    x2 = -b - delta
    x1 = x1/(2.0*a)
    x2 = x2/(2.0*a)

    write(*,*) 'Real roots:', x1, x2

end program second_degree_eq
Outline

Introduction

Fortran Basics
- My First Fortran Program
- Compiling and Linking Your First Program
- Making Choices
- More Types and Choices
- Wrapping it Up 1

More Fortran Basics

Integer Types and Iterating

More on Compiling and Linking

Homeworks
Fixing a Defect

- User wants to solve $x^2 + 1 = 0$

- Enters: 1, 0, 1

- Gets: Real roots: NaN, NaN

- Discriminant is negative, its square root is Not A Number, NaN

- Let's avoid this, by changing from:

  $\delta = \sqrt{b^2 - 4.0a \cdot c}$

  to:

  $\delta = b^2 - 4.0a \cdot c$

  if ($\delta < 0.0$) then

  stop

  end if

  $\delta = \sqrt{\delta}$

- Try it now!

- Did you check that normal cases still work? Good.
Fixing a Defect

- User wants to solve \( x^2 + 1 = 0 \)
  - Enters: 1, 0, 1

Let's avoid this, by changing from:

\[
\delta = \sqrt{b^2 - 4.0a \cdot c}
\]

...to:

\[
\delta = b^2 - 4.0a \cdot c
\]

if (\( \delta < 0.0 \)) then

\[
\text{stop}
\]

end if

\[
\delta = \sqrt{\delta}
\]

Try it now!

Did you check that normal cases still work? Good.
Fixing a Defect

- User wants to solve $x^2 + 1 = 0$
  - Enters: 1, 0, 1
  - Gets: Real roots: NaN, NaN

Let's avoid this, by changing from:

$$\delta = \sqrt{b^2 - 4.0 \times a \times c}$$

To:

$$\delta = b^2 - 4.0 \times a \times c$$

if ($\delta < 0.0$) then
  stop
end if
$$\delta = \sqrt{\delta}$$

Try it now!

Did you check that normal cases still work? Good.
Fixing a Defect

- User wants to solve \( x^2 + 1 = 0 \)
  - Enters: 1, 0, 1
  - Gets: Real roots: NaN, NaN

- Discriminant is negative, its square root is Not A Number, NaN

- Let's avoid this, by changing from:
  \[
  \text{delta} = \sqrt{b^2 - 4.0 \times a \times c}
  \]
  to:
  \[
  \text{delta} = b^2 - 4.0 \times a \times c
  \]
  if (delta < 0.0) then
  stop
  end if
  delta = sqrt(delta)

- Try it now!

- Did you check that normal cases still work? Good.
Fixing a Defect

- User wants to solve $x^2 + 1 = 0$
  - Enters: $1, 0, 1$
  - Gets: Real roots: NaN, NaN

- Discriminant is negative, its square root is Not A Number, NaN

- Let's avoid this, by changing from:
  
  \[
  \text{delta} = \sqrt{b^2 - 4.0*a*c}
  \]

  to:
  
  \[
  \text{delta} = b^2 - 4.0*a*c \\
  \text{if} (\text{delta} < 0.0) \text{ then} \\
  \quad \text{stop} \\
  \text{end if} \\
  \text{delta} = \sqrt{\text{delta}}
  \]

- Try it now!
Fixing a Defect

- User wants to solve $x^2 + 1 = 0$
  - Enters: 1, 0, 1
  - Gets: Real roots: NaN, NaN

- Discriminant is negative, its square root is Not A Number, NaN

- Let’s avoid this, by changing from:
  
  $$\text{delta} = \sqrt{b^2 - 4.0*a*c}$$

  to:

  $$\text{delta} = b^2 - 4.0*a*c$$

  if ($\text{delta} < 0.0$) then
    stop
  end if
  \text{delta} = \sqrt{\text{delta}}

- Try it now!
- Did you check that normal cases still work? Good.
Conditional Statement

- `if (logical-condition) then
  block of statements
end if`

  Executes `block of statements` only if `logical-condition` is true

  Comparison operators: `==` (equal), `!=` (not equal), `>`, `<`, `>=`, `<=`

  When `block` is made up by a single statement, you can use one-liner `if (logical-condition) statement` instead

Try it now!

Did you check that normal cases still work? Good.
Conditional Statement

- **if** (logical-condition) **then**
  *block of statements*
end if

- Executes *block of statements* only if *logical-condition* is true
- Comparison operators: == (equal), /= (not equal), >, <, >=, <=
- When *block* is made up by a single statement, you can use one-liner **if** (logical-condition) **statement** instead

- But let’s be more polite by changing from:

```
if (delta < 0.0) then
  stop
endif
```

  to:

```
if (delta < 0.0) stop 'No real roots!'
```

- Try it now!
Conditional Statement

- if (logical-condition) then
  block of statements
end if

  - Executes block of statements only if logical-condition is true
  - Comparison operators: == (equal), /= (not equal), >, <, >=, <=
  - When block is made up by a single statement, you can use one-liner if (logical-condition) statement instead

- But let’s be more polite by changing from:
  
  ```
  if (delta < 0.0) then
    stop
  endif
  ```

- to:
  
  ```
  if (delta < 0.0) stop ‘No real roots!’
  ```

- Try it now!

- Did you check that normal cases still work? Good.
Some folks prefer this:

```fortran
if (delta < 0.0) stop 'No real roots!'
```

and it’s OK
Some folks prefer this:

```fortran
if (delta < 0.0) stop 'No real roots!'
```

and it’s OK

Other folks prefer this:

```fortran
if (delta < 0.0) then
   stop 'No real roots!'
end if
```

and it’s OK
Some folks prefer this:

```fortran
if (delta < 0.0) stop 'No real roots!'
```

and it’s OK

Other folks prefer this:

```fortran
if (delta < 0.0) then
  stop 'No real roots!'
end if
```

and it’s OK

Sloppy guys write:

```fortran
if (delta < 0.0) then
  stop 'No real roots!'
end if
```

but this is not that good...
Good Style

- Some folks prefer this:
  ```fortran
  if (delta < 0.0) stop 'No real roots!'
  ```
  and it’s OK

- Other folks prefer this:
  ```fortran
  if (delta < 0.0) then
    stop 'No real roots!'  
  end if
  ```
  and it’s OK

- Sloppy guys write:
  ```fortran
  if (delta < 0.0) then
    stop 'No real roots!'  
  end if
  ```
  but this is not that good...

- In general, Fortran disregards white space, but proper indentation visualizes program control flow
Outline

Introduction

Fortran Basics
My First Fortran Program
Compiling and Linking Your First Program
Making Choices
More Types and Choices
Wrapping it Up 1

More Fortran Basics

Integer Types and Iterating

More on Compiling and Linking

Homeworks
Let’s Refactor Our Program (and Test it!)

! roots of a 2nd degree equation with real coefficients

program second_degree_eq
    implicit none
    real :: delta
    real :: rp
    real :: a, b, c

    print *,’Solving ax^2+bx+c=0, enter a, b, c: ’
    read(*,*) a, b, c

    delta = b*b - 4.0*a*c
    if (delta < 0.0) stop ’No real roots!’
    delta = sqrt(delta)/(2.0*a)

    rp = -b/(2.0*a)

    print *,’Real roots: ’, rp+delta, rp-delta

end program second_degree_eq
And Now Make it More Complex!

! roots of a 2nd degree equation with real coefficients

program second_degree_eq
  implicit none
  real :: delta, rp, a, b, c
  logical :: rroots

  print *, 'Solving ax^2+bx+c=0, enter a, b, c: '
  read(*,*) a, b, c
  delta = b*b - 4.0*a*c
  rroots = .true.
  if (delta < 0.0) then
    delta = -delta
    rroots = .false.
  end if
  delta = sqrt(delta)/(2.0*a)
  rp = -b/(2.0*a)
  if (rroots) then
    print *, 'Real roots: ', rp+delta, rp-delta
  else
    print *, 'Complex roots: ', rp, '+', delta, 'i ', &
             rp, '-', delta, 'i'
  end if
end program second_degree_eq
More Types and Choices

- **logical** type represents logical values
  - Can be `.true.` or `.false.`
More Types and Choices

- **logical** type represents logical values
  - Can be `.true.` or `.false.`

- **else** has to appear inside an `if () then/end if` pair, and
  the following statements up to `end if` are executed when the
  logical condition is false

- Allows for choosing between alternative paths
More Types and Choices

- **logical** type represents logical values
  - Can be `.true.` or `.false.`

- **else** has to appear inside an `if () then/end if` pair, and the following statements up to `end if` are executed when the logical condition is false

- Allows for choosing between alternative paths

- Again, use proper indentation
And Now Make it More Complex!

! roots of a 2nd degree equation with real coefficients

program second_degree_eq
    implicit none
    real :: delta, rp, a, b, c
    logical :: rroots

    print *, 'Solving ax^2+bx+c=0, enter a, b, c: '
    read(*,*) a, b, c
    delta = b*b - 4.0*a*c
    rroots = .true.
    if (delta < 0.0) then
        delta = -delta
        rroots = .false.
    end if
    delta = sqrt(delta)/(2.0*a)
    rp = -b/(2.0*a)
    if (rroots) then
        print *, 'Real roots: ', rp+delta, rp-delta
    else
        print *,'Complex roots: ', rp, '+', delta, 'i ', &
             rp, '-', delta, 'i'
    end if
end program second_degree_eq
More Types and Choices

- **logical** type represents logical values
  - Can be `.true.` or `.false.`

- **else** has to appear inside an `if () then/end if` pair, and the following statements up to `end if` are executed when the logical condition is false

- Allows for choosing between alternative paths

- Again, use proper indentation

- And Fortran statements cannot exceed one line, unless it ends with an `&`
program second_degree_eq
  implicit none
  real :: delta, rp, a, b, c
  logical :: rroots

  print *, 'Solving ax^2+bx+c=0, enter a, b, c: '  
  read(*,*) a, b, c
  delta = b*b - 4.0*a*c  
  rroots = .true.
  if (delta < 0.0) then
    delta = -delta
    rroots = .false.
  end if
  delta = sqrt(delta)/(2.0*a)
  rp = -b/(2.0*a)
  if (rroots) then
    print *, 'Real roots: ', rp+delta, rp-delta
  else
    print *,'Complex roots: ', rp, '+', delta, 'i ', &
             rp, '-', delta, 'i'
  end if
end program second_degree_eq
Let's Make it as Complex as Possible!

! roots of a 2nd degree equation with real coefficients

program second_degree_eq
  implicit none
  complex :: delta
  complex :: z1, z2
  real :: a, b, c

  print *, 'Solving ax^2+bx+c=0, enter a, b, c: '
  read(*,*) a, b, c

  delta = b*b - 4.0*a*c
  delta = sqrt(delta)

  z1 = (-b+delta)/(2.0*a)
  z2 = (-b-delta)/(2.0*a)

  print *, 'Roots: ', z1, z2
end program second_degree_eq
Fortran has **complex** type:

- hosting two real values, real and imaginary parts
Complex Numbers

- Fortran has `complex` type:
  - hosting two real values, real and imaginary parts

- Most math functions like `sqrt()` work for complex type too!
  - Returning correct results, instead of NaNs
Fortran has **complex** type:
- hosting two real values, real and imaginary parts

Most math functions like `sqrt()` work for complex type too!
- Returning correct results, instead of NaNs

And so do **read, write, and print**
Complex Numbers

- Fortran has **complex** type:
  - hosting two real values, real and imaginary parts
- Most math functions like `sqrt()` work for complex type too!
  - Returning correct results, instead of NaNs
- And so do **read, write, and print**
- `(1.5, 2.3)` is *Fortranese* for `1.5 + 2.3i`
Try it Now!

! roots of a 2nd degree equation with real coefficients

program second_degree_eq
  implicit none
  complex :: delta
  complex :: z1, z2
  real :: a, b, c

  print *, 'Solving ax^2+bx+c=0, enter a, b, c: '
  read(*,*) a, b, c

  delta = b*b - 4.0*a*c
  delta = sqrt(delta)

  z1 = (-b+delta)/(2.0*a)
  z2 = (-b-delta)/(2.0*a)

  print *, 'Roots: ', z1, z2
end program second_degree_eq
Making it More Robust

- What if user inputs zeroes for $a$ or $a$ and $b$?

```plaintext
if (a == 0.0) then
  if (b == 0.0) then
    if (c == 0.0) then
      write(0,*) 'A trivial identity!
    else
      write(0,*) 'Plainly absurd!
    end if
  else
    write(0,*) 'Too simple problem!
  end if
end if
```
Making it More Robust

What if user inputs zeroes for $a$ or $a$ and $b$?

Let’s prevent these cases, inserting right after input:

```fortran
if (a == 0.0) then
   if (b == 0.0) then
      if (c == 0.0) then
         write(0,*) 'A trivial identity!'
      else
         write(0,*) 'Plainly absurd!'
      end if
   else
      write(0,*) 'Too simple problem!'
   end if
else
   write(0,*) 'Too simple problem!'
end if
stop
end if
```
Making it More Robust

- What if user inputs zeroes for \( a \) or \( a \) and \( b \)?
- Let’s prevent these cases, inserting right after input:
  
  ```
  if (a == 0.0) then
    if (b == 0.0) then
      if (c == 0.0) then
        write(0,*) 'A trivial identity!'
      else
        write(0,*) 'Plainly absurd!'
      end if
    else
      write(0,*) 'Too simple problem!'
    end if
  else
    write(0,*) 'Too simple problem!'
  end if
  ```

- Can you see the program logic?

- Try it now!

- Did you check that normal cases still work? Good.
Making it More Robust

- What if user inputs zeroes for $a$ or $a$ and $b$?
- Let’s prevent these cases, inserting right after input:

```python
if (a == 0.0) then
    if (b == 0.0) then
        if (c == 0.0) then
            write(0,*) 'A trivial identity!'
        else
            write(0,*) 'Plainly absurd!'
        end if
    else
        write(0,*) 'Too simple problem!'
    end if
else
    write(0,*) 'Too simple problem!'
end if
```

- Can you see the program logic?
- Try it now!
What if user inputs zeroes for $a$ or $a$ and $b$?

Let’s prevent these cases, inserting right after input:

```fortran
if (a == 0.0) then
  if (b == 0.0) then
    if (c == 0.0) then
      write(0,*) 'A trivial identity!'
    else
      write(0,*) 'Plainly absurd!'
    end if
  else
    write(0,*) 'Too simple problem!'
  end if
else
  write(0,*) 'Too simple problem!'
end if
```

Can you see the program logic?

Try it now!

Did you check that normal cases still work? Good.
Miscellaneous remarks

- Nested `ifs` can be a problem
  - `else` marries innermost `if () then/end if` pair
  - Proper indentation is almost mandatory to sort it out
Nested *ifs* can be a problem

- *else* marries innermost *if* () *then/end if* pair
- Proper indentation is almost mandatory to sort it out

What’s this *write(0,*)* stuff?

- *write()* and *read()* let you specify an output (input) file ‘handle’ called a unit
- Unit 0 is usually connected to a special file, mandatory for error messages to the terminal (e.g. UNIX standard error)
- By the way, *write(*,*)* is a system independent idiom for what you’ll often find written as *write(6,*)*
- And *read(*,*)* is a system independent idiom for what you’ll often find written as *read(5,*)*
- And *stop error-message* is equivalent to: *write(0,*) error-message stop*
Nested *ifs* can be a problem

- *else* marries innermost *if* () *then/end if* pair
- Proper indentation is almost mandatory to sort it out

What’s this `write(0,*` stuff?

- `write()` and `read()` let you specify an output (input) file ‘handle’ called a unit
- Unit 0 is usually connected to a special file, mandatory for error messages to the terminal (e.g. UNIX standard error)
- By the way, `write(*,*)` is a system independent idiom for what you’ll often find written as `write(6,*)`
- And `read(*,*)` is a system independent idiom for what you’ll often find written as `read(5,*)`
- And `stop error-message` is equivalent to: `write(0,*` 
  `error-message`  
  `stop`

Best practice: if your program has to fail, always have it fail in a controlled way
Let’s give names to if constructs:

```plaintext
no2nd: if (a == 0.0) then
    no1st: if (b == 0.0) then
        no0th: if (c == 0.0) then
            write(0,*) 'A trivial identity!'
        else no0th
            write(0,*) 'Plainly absurd!'
        end if no0th
    else no1st
        write(0,*) 'Too simple problem!'
    end if no1st
else no1st
    stop
end if no2nd
```
Let’s give names to if constructs:

```fortran
no2nd: if (a == 0.0) then
    no1st: if (b == 0.0) then
        no0th: if (c == 0.0) then
            write(0,*) 'A trivial identity!'
        else no0th
            write(0,*) 'Plainly absurd!'
        end if no0th
    else no1st
        write(0,*) 'Too simple problem!'
    end if no1st
else no2nd
    stop
end if no2nd
```

Giving names to constructs makes program logic more explicit
Let’s give names to if constructs:

```plaintext
no2nd: if (a == 0.0) then
    no1st: if (b == 0.0) then
        no0th: if (c == 0.0) then
            write(0,*) 'A trivial identity!'
        else no0th
            write(0,*) 'Plainly absurd!'
        end if no0th
    else no1st
        write(0,*) 'Too simple problem!'
    end if no1st
else no2nd
```

- Giving names to constructs makes program logic more explicit
- Names are for readability purposes only, do not enforce pairing rules
Let's give names to if constructs:

```plaintext
no2nd: if (a == 0.0) then
    no1st: if (b == 0.0) then
        no0th: if (c == 0.0) then
            write(0,*) 'A trivial identity!'
            else no0th
                write(0,*) 'Plainly absurd!'
                end if no0th
            else no1st
                write(0,*) 'Too simple problem!'
                end if no1st
        else no2nd
    else no1st
end if no2nd
```

- Giving names to constructs makes program logic more explicit.
- Names are for readability purposes only, do not enforce pairing rules.

**Best practice:** always give names to constructs which span many lines of code or are deeply nested.
Fortran Code, in the Beginning of Times

- The one on the left, is the statement $Z(I) = Y + W(I)$
- The one in the middle, is an IBM punch card reader
- The one on the right, is a complete Fortran source program
- But you’ll only encounter these in museums, nowadays
C ROOTS OF A 2ND DEGREE EQUATION WITH REAL COEFFICIENTS

PROGRAM EQ2DEG

IMPLICIT NONE
REAL DELTA
REAL RP
REAL A, B, C

PRINT *, 'SOLVING AX^2+BX+C=0, ENTER A, B, C: '
READ(*,*) A, B, C

DELTA = B*B - 4.0*A*C
IF (DELTA.LT.0.0) STOP 'NO REAL ROOTS!'
DELTA = SQRT(DELTA)/(2.0*A)

RP = -B/(2.0*A)

PRINT *, 'REAL ROOTS: ', RP+DELTA, RP-DELTA

END
Legacy Code: Distinctive Characters

- Code is all capitals
  - First computers had only uppercase letters
- Fixed source form
  - The legacy of punch cards
  - Comment lines must be marked with a `c` or `*` in first column
  - First six columns on each line are reserved for labels and to mark continuation lines
  - Columns after the 72nd are ignored (cause of really nasty bugs!)
- No double colon on variable declarations
  - And no way to initialize a variable at declaration, for that matter
  - More on this later
- And this example is not that different...
A Bottle of Fortran, Vintage Year 1963

C SOLUTION OF QUADRATIC EQUATION
C (P. 122 OF A FORTRAN PRIMER BY E. ORGANICK)

1 READ INPUT TAPE 5, 51, ANAME, N
51 FORMAT(A6,I2)
   WRITE OUTPUT TAPE 6,52, ANAME
52 FORMAT(1H1,33HROOTS OF QUADRATIC EQUATIONS FROM A6)
   DO 21 I = 1, N
   READ INPUT TAPE 5, 53, A, B, C
53 FORMAT(3F10.2)
   WRITE OUTPUT TAPE 6,54, I, A, B, C
54 FORMAT(1H0,8HSET NO. I2/5H A = F8.2,12X,4HB = F8.2,12X,4HC = F8.2)
   IF(A) 10, 7, 10
7 RLIN = -C/B
   WRITE OUTPUT TAPE 6, 55, RLIN
55 FORMAT(7H LINEAR,25X,4HX = F10.3)
   GO TO 21
10 D = B**2 - 4.*A*C
   IF(D) 12, 17, 17
12 COMPR = -B/(2.*A)
   COMP1 = SQRTF(-D)/(2.*A)
   COMP2 = -COMP1
   WRITE OUTPUT TAPE 6, 56, COMPR, COMP1, COMP2
56 FORMAT(8H COMPLEX,21X,7HR(X1)= F10.3,11X,7HI(X1)= F10.3,/1H ,28X,
   17HR(X2)= F10.3,11X,7HI(X2)= F10.3)
   GO TO 21
16 GO TO 21
17 REAL1 = (-B + SQRTF(D))/(2.*A)
   REAL2 = (-B - SQRTF(D))/(2.*A)
20 WRITE OUTPUT TAPE 6, 57, REAL1, REAL2
57 FORMAT(6H REAL 25X,5HX1 = F10.3,13X,5HX2 = F10.3)
21 CONTINUE
   WRITE OUTPUT TAPE 6, 58, ANAME
58 FORMAT(8H0END OF A6)
   GO TO 1
END
Best Practice: Free Yourself

- Write new code in free source form
  - No limits on beginning of program statements
  - Each line may contain up to 132 default characters
  - Comments can be added at end of line
  - And it comes for free: just give your source file name an `.f90` extension

- Use new language features
  - Like new styles for declarations
  - Or naming of constructs
  - They are more powerful and readable

- We’ll focus on modern Fortran programming style
  - Making you aware of differences you are most likely to encounter
  - Look at compiler manuals or reference books to tame very old codes
Outline

Introduction

**Fortran Basics**
- My First Fortran Program
- Compiling and Linking Your First Program
- Making Choices
- More Types and Choices
- Wrapping it Up 1

More Fortran Basics

Integer Types and Iterating

More on Compiling and Linking

Homeworks
A Fortran Program is Made of:

- **Comments**
  - Compiler disregards them, but humans do not
  - Please, use them
  - Do not abuse them, please

- **Program units**
  - One, at least: `program`
  - Some of them (functions) are intrinsic to the language

- **Variables**
  - Named memory location you can store values into
  - Must be declared

- **Variables declarations**
  - Give name to memory location you can store values into
  - An initial value can be specified
A Fortran Program is Made of: II

- **Expressions**
  - Compute values to store in variables
  - Compute values to pass to functions and statements

- **Statements**
  - Units of executable work
  - Whose execution can be controlled by other constructs

- **if statements and constructs**
  - Allow for conditional and alternative execution
  - For both single statements and blocks of
Best Practices

- Use free source form
- `implicit none` statement
  - Turn off implicit declarations
- Use proper indentation
  - Compilers don’t care about
  - Readers visualize flow control
- Give names to complex control structures, readers will appreciate
- Do non-regression testing
  - Whenever functionalities are added
  - Whenever you rewrite a code in a different way
- Fail in a controlled way
  - Giving feedback to humans
Outline

Introduction

Fortran Basics

More Fortran Basics
  My First Fortran Functions
  Making it Correct
  Making it Robust
  Copying with Legacy
  Wrapping it Up 2

Integer Types and Iterating

More on Compiling and Linking

Homeworks
Introduction

Fortran Basics

More Fortran Basics
  My First Fortran Functions
  Making it Correct
  Making it Robust
  Copying with Legacy
  Wrapping it Up 2

Integer Types and Iterating

More on Compiling and Linking

Homeworks
function theta(x) !Heaviside function, useful in DSP
  implicit none
  real :: theta
  real :: x

  theta = 1.0
  if (x < 0.0 ) theta = 0.0
end function theta

function sinc(x) !sinc function as used in DSP
  implicit none
  real :: sinc
  real :: x
  real, parameter :: pi = acos(-1.0)

  x = x*pi
  sinc = 1.0
  if (x /= 0.0) sinc = sin(x)/x
end function sinc

function rect(t, tau) !generalized rectangular function, useful in DSP
  implicit none
  real :: rect
  real :: t, tau
  real :: abs_t, half_tau
  real, external :: theta

  abs_t = abs(t)
  half_tau = 0.5*tau
  rect = 0.5
  if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect
function theta(x) !Heaviside function, useful in DSP
  implicit none
  real :: theta
  real :: x

  theta = 1.0
  if (x < 0.0 ) theta = 0.0
end function theta

function sinc(x) !sinc function as used in DSP
  implicit none
  real :: sinc
  real :: x
  real, parameter :: pi = acos(-1.0)

  x = x*pi
  sinc = 1.0
  if (x /= 0.0) sinc = sin(x)/x
end function sinc

function rect(t, tau) !generalized rectangular function, useful in DSP
  implicit none
  real :: rect
  real :: t, tau
  real :: abs_t, half_tau
  real, external :: theta

  abs_t = abs(t)
  half_tau = 0.5*tau
  rect = 0.5
  if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect
Functions and their Definition

- Functions are program units
  - Function name must be a legal Fortran name
  - Functions specialty is performing computations and returning a value

- How to return a value
  - Just assign it to the function name, as if it were a variable
  - But this doesn't force function termination
  - Multiple assignments can be done
  - The last assigned value before function execution is complete will be returned
function theta(x) ! Heaviside function, useful in DSP
    implicit none
    real :: theta
    real :: x

    theta = 1.0
    if (x < 0.0 ) theta = 0.0
end function theta

function sinc(x) ! sinc function as used in DSP
    implicit none
    real :: sinc
    real :: x
    real, parameter :: pi = acos(-1.0)

    x = x*pi
    sinc = 1.0
    if (x /= 0.0) sinc = sin(x)/x
end function sinc

function rect(t, tau) ! generalized rectangular function, useful in DSP
    implicit none
    real :: rect
    real :: t, tau
    real :: abs_t, half_tau
    real, external :: theta

    abs_t = abs(t)
    half_tau = 0.5*tau
    rect = 0.5
    if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect
Functions and their Definition

- Functions are program units
  - Function name must be a legal Fortran name
  - Functions specialty is performing computations and returning a value

- Type of returned value must be declared
  - In the definition and in each unit calling them
  - Same as a variable declaration
  - Could be declared on the function heading, but it’s less flexible and less readable
  - More on this later...
Functions and their Definition

- Functions are program units
  - Function name must be a legal Fortran name
  - Functions specialty is performing computations and returning a value

- Type of returned value must be declared
  - In the definition and in each unit calling them
  - Same as a variable declaration
  - Could be declared on the function heading, but it’s less flexible and less readable
  - More on this later...

- How to return a value
  - Just assign it to the function name, as if it were a variable
  - But this doesn’t force function termination
  - Multiple assignments can be done
  - The last assigned value before function execution is complete will be returned
function theta(x) ! Heaviside function, useful in DSP
    implicit none
    real :: theta
    real :: x

    theta = 1.0
    if (x < 0.0 ) theta = 0.0
end function theta

function sinc(x) ! sinc function as used in DSP
    implicit none
    real :: sinc
    real :: x
    real, parameter :: pi = acos(-1.0)

    x = x*pi
    sinc = 1.0
    if (x /= 0.0) sinc = sin(x)/x
end function sinc

function rect(t, tau) ! generalized rectangular function, useful in DSP
    implicit none
    real :: rect
    real :: t, tau
    real :: abs_t, half_tau
    real, external :: theta

    abs_t = abs(t)
    half_tau = 0.5*tau
    rect = 0.5
    if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect
Function Arguments and Local Variables

- Functions have arguments
  - Declared like variables inside the function
  - Arguments are termed *dummy arguments* inside the function
  - The arguments passed to a function by a calling unit are termed *actual arguments*
Function Arguments and Local Variables

- Functions have arguments
  - Declared like variables inside the function
  - Arguments are termed *dummy arguments* inside the function
  - The arguments passed to a function by a calling unit are termed *actual arguments*

- What if two functions have arguments with identical names?
  - No conflicts of sort, they are completely independent
Function Arguments and Local Variables

- Functions have arguments
  - Declared like variables inside the function
  - Arguments are termed *dummy arguments* inside the function
  - The arguments passed to a function by a calling unit are termed *actual arguments*
- What if two functions have arguments with identical names?
  - No conflicts of sort, they are completely independent
- What if a dummy argument has the same name of a variable elsewhere in the program?
  - No conflicts of sort, they are completely independent
function theta(x) !Heaviside function, useful in DSP
    implicit none
    real :: theta
    real :: x

    theta = 1.0
    if (x < 0.0 ) theta = 0.0
end function theta

function sinc(x) !sinc function as used in DSP
    implicit none
    real :: sinc
    real :: x
    real, parameter :: pi = acos(-1.0)

    x = x*pi
    sinc = 1.0
    if (x /= 0.0) sinc = sin(x)/x
end function sinc

function rect(t, tau) !generalized rectangular function, useful in DSP
    implicit none
    real :: rect
    real :: t, tau
    real :: abs_t, half_tau
    real, external :: theta

    abs_t = abs(t)
    half_tau = 0.5*tau
    rect = 0.5
    if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect
Function Arguments and Local Variables

- Functions have arguments
  - Declared like variables inside the function
  - Arguments are termed *dummy arguments* inside the function
  - The arguments passed to a function by a calling unit are termed *actual arguments*

- What if two functions have arguments with identical names?
  - No conflicts of sort, they are completely independent

- What if a dummy argument has the same name of a variable elsewhere in the program?
  - No conflicts of sort, they are completely independent

- Variables can be defined inside functions
  - Again, they are local, thus completely independent from the rest of the program
function theta(x) ! Heaviside function, useful in DSP
    implicit none
    real :: theta
    real :: x

    theta = 1.0
    if (x < 0.0) theta = 0.0
end function theta

function sinc(x) ! sinc function as used in DSP
    implicit none
    real :: sinc
    real :: x
    real, parameter :: pi = acos(-1.0)

    x = x*pi
    sinc = 1.0
    if (x /= 0.0) sinc = sin(x)/x
end function sinc

function rect(t, tau) ! generalized rectangular function, useful in DSP
    implicit none
    real :: rect
    real :: t, tau
    real :: abs_t, half_tau
    real, external :: theta

    abs_t = abs(t)
    half_tau = 0.5*tau
    rect = 0.5
    if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect
Intrinsic vs. External

- Fortran sports a wealth (over a hundred!) of predefined functions and procedures
- These are termed *intrinsic*
  - **acos(x)** returns the arc cosine of $x$ such that $|x| \leq 1$ in the range $0 \leq \arccos(x) \leq \pi$
  - **sin(x)** returns the sine function value of $x$ in radians
  - **abs(x)** returns the absolute value of $x$
Intrinsic vs. External

- Fortran sports a wealth (over a hundred!) of predefined functions and procedures
- These are termed *intrinsic*
  - \( \text{acos}(x) \) returns the arc cosine of \( x \) such that \(|x| \leq 1\) in the range \( 0 \leq \arccos(x) \leq \pi \)
  - \( \sin(x) \) returns the sine function value of \( x \) in radians
  - \( \text{abs}(x) \) returns the absolute value of \( x \)

- What’s this *external* keyword?
- It’s one of the many attributes you can give to something you define
  - *external* tells the compiler *theta* is an external (i.e. non intrinsic) function
  - So the compiler is not forced to guess what it is from its use
  - And that way, masters can override intrinsic functions
function theta(x)  !Heaviside function, useful in DSP
    implicit none
    real :: theta
    real :: x

    theta = 1.0
    if (x < 0.0 ) theta = 0.0
end function theta

function sinc(x)  !sinc function as used in DSP
    implicit none
    real :: sinc
    real :: x
    real, parameter :: pi = acos(-1.0)

    x = x*pi
    sinc = 1.0
    if (x /= 0.0) sinc = sin(x)/x
end function sinc

function rect(t, tau)  !generalized rectangular function, useful in DSP
    implicit none
    real :: rect
    real :: t, tau
    real :: abs_t, half_tan
    real, external :: theta

    abs_t = abs(t)
    half_tan = 0.5*tau
    rect = 0.5
    if (abs_t /= half_tan) rect = theta(half_tan-abs_t)
end function rect
The **parameter** Attribute

- The **parameter** attribute is used to declare named constants
  - i.e. variables that cannot be modified after initialization (compiler will bark if you try)
The **parameter** attribute is used to declare named constants

- i.e. variables that cannot be modified after initialization (compiler will bark if you try)

- In initialization expressions:
The parameter Attribute

- The **parameter** attribute is used to declare named constants
  - i.e. variables that cannot be modified after initialization (compiler will bark if you try)

- In initialization expressions:
  - only constants (possibly other **parameters**) can be used
The **parameter** Attribute

- The **parameter** attribute is used to declare named constants
  - i.e. variables that cannot be modified after initialization (compiler will bark if you try)

- In initialization expressions:
  - only constants (possibly other **parameters**) can be used
  - only intrinsic operators or functions are allowed
The `parameter` attribute is used to declare named constants
- i.e. variables that cannot be modified after initialization (compiler will bark if you try)

In initialization expressions:
- only constants (possibly other `parameters`) can be used
- only intrinsic operators or functions are allowed

Best practice: always give name to constants
- Particularly if unobvious, like `1.0/137.0`
- It also helps to centralize updates (well, not for $\pi$)
Outline

Introduction

Fortran Basics

More Fortran Basics
  My First Fortran Functions
  Making it Correct
  Making it Robust
  Copying with Legacy
  Wrapping it Up 2

Integer Types and Iterating

More on Compiling and Linking

Homeworks
On To Testing

- Let’s put the code in a file named `dsp.f90`.
- Best practice: always put different groups of related functions in different files.
  - Helps to tame complexity.
  - You can always pass all source files to the compiler.
  - And you’ll learn to do better ...

And let’s write a program to test all functions.
And be wary, check again actual arguments after all function calls.

Best practice: always write a special purpose program to test each subset of functions.
Best to include in the program automated testing of all relevant cases.

Let’s do by hand with I/O for now, to make it short.
On To Testing

▶ Let’s put the code in a file named `dsp.f90`

▶ Best practice: always put different groups of related functions in different files
  - Helps to tame complexity
  - You can always pass all source files to the compiler
  - And you’ll learn to do better ...

▶ And let’s write a program to test all functions
  - And be wary, check again actual arguments after all function calls

▶ Best practice: always write a special purpose program to test each subset of functions
  - Best to include in the program automated testing of all relevant cases
  - Let’s do by hand with I/O for now, to make it short
function theta(x) !Heaviside function, useful in DSP
  implicit none
  real :: theta
  real :: x

  theta = 1.0
  if (x < 0.0 ) theta = 0.0
end function theta

function sinc(x) !sinc function as used in DSP
  implicit none
  real :: sinc
  real :: x
  real, parameter :: pi = acos(-1.0)

  x = x*pi
  sinc = 1.0
  if (x /= 0.0) sinc = sin(x)/x
end function sinc

function rect(t, tau) !generalized rectangular function, useful in DSP
  implicit none
  real :: rect
  real :: t, tau
  real :: abs_t, half_tau
  real, external :: theta

  abs_t = abs(t)
  half_tau = 0.5*tau
  rect = 0.5
  if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect
DSP test program

- We have collected DSP functions in `dsp.f90` source file

```fortran
program dsp_test
  real :: i, j, k
  real :: rtheta, rsinc, rrect
  real, external :: theta, sinc, rect
  print *, 'Enter i, j, k:
  read(*,*) i, j, k
  rtheta = theta(i)
  rsinc = sinc(i)
  rrect = rect(j, k)
  write(*,*) 'theta(', i, ')= ', rtheta
  write(*,*) 'sinc(', i, ')= ', rsinc
  write(*,*) 'rect(', j, ',', k, ')= ', rrect
end program dsp_test
```
DSP test program

- We have collected DSP functions in `dsp.f90` source file
- We want to test these functions
We have collected DSP functions in `dsp.f90` source file

We want to test these functions

Let’s write a `dsp_test.f90` program:

```fortran
program dsp_test
    real :: i,j,k
    real :: rtheta, rsinc, rrect
    real, external :: theta, sinc, rect
    print *, 'Enter i, j, k:'
    read(*,*) i, j, k
    rtheta = theta(i)
    rsinc = sinc(i)
    rrect = rect(j, k)
    write(*,*) 'theta(', i, ')= ', rtheta
    write(*,*) 'sinc(', i , ')= ', rsinc
    write(*,*) 'rect(', j, ',', k, ')= ', rrect
end program dsp_test
```
DSP test program

- We have collected DSP functions in `dsp.f90` source file
- We want to test these functions
- Let's write a `dsp_test.f90` program:

```fortran
program dsp_test
  real :: i, j, k
  real, external :: theta, sinc, rect
  print *, 'Enter i, j, k:
  read(*,*) i, j, k
  rtheta = theta(i)
  rsinc = sinc(i)
  rrect = rect(j, k)
  write(*,*) 'theta(', i, ')= ', rtheta
  write(*,*) 'sinc(', i, ')= ', rsinc
  write(*,*) 'rect(', j, ',', k, ')= ', rrect
end program dsp_test
```
DSP test program

- We have collected DSP functions in `dsp.f90` source file
- We want to test these functions
- Let's write a `dsp_test.f90` program:

```fortran
program dsp_test

  real :: i, j, k
  real :: rtheta, rsinc, rrect
  real, external :: theta, sinc, rect

  print *, 'Enter i, j, k:
  read(*,*) i, j, k
  rtheta = theta(i)
  rsinc = sinc(i)
  rrect = rect(j, k)
  write(*,*) 'theta(', i, ')= ', rtheta
  write(*,*) 'sinc(', i , ')= ', rsinc
  write(*,*) 'rect(', j, ',', k, ')= ', rrect

end program dsp_test
```
DSP test program

- We have collected DSP functions in `dsp.f90` source file
- We want to test these functions
- Let's write a `dsp_test.f90` program:

```fortran
program dsp_test
    
    real :: i, j, k
    real :: rtheta, rsinc, rrect
    real, external :: theta, sinc, rect
    
    print *, 'Enter i, j, k:'
    read(*,*) i, j, k

end program dsp_test
```
We have collected DSP functions in `dsp.f90` source file

We want to test these functions

Let’s write a `dsp_test.f90` program:

```fortran
program dsp_test
    real :: i,j,k
    real :: rtheta, rsinc, rrect
    real, external :: theta, sinc, rect
    print *, 'Enter i, j, k:'
    read(*,*) i, j, k

    rtheta = theta(i)
    rsinc = sinc(i)
    rrect = rect(j, k)
end program dsp_test
```
We have collected DSP functions in `dsp.f90` source file

We want to test these functions

Let’s write a `dsp_test.f90` program:

```fortran
program dsp_test
    
    real :: i,j,k
    real :: rtheta, rsinc, rrect
    real, external :: theta, sinc, rect
    
    print *, 'Enter i, j, k:'
    read(*,*) i, j, k
    
    rtheta = theta(i)
    rsinc = sinc(i)
    rrect = rect(j, k)
    
    write(*,*) 'theta(', i, ')= ', rtheta
    write(*,*) 'sinc(', i , ')= ', rsinc
    write(*,*) 'rect(', j, ',', k, ')= ', rrect

end program dsp_test
```
Testing DSP Functions

- Let’s build our test program putting all together:

  ```bash
  user@cineca$> gfortran dsp.f90 dsp_test.f90 -o dsp_test
  ```

  - `-o` option specifies the name `dsp_test` for the executable

- Something is going wrong, isn’t it?
- Seems like one function changed its actual argument!
Let’s build our test program putting all together:

```
user@cineca$> gfortran dsp.f90 dsp_test.f90 -o dsp_test
```

- `-o` option specifies the name `dsp_test` for the executable

Now run the program:

```
user@cineca$> ./dsp_test
Enter i, j, k:
-1 0 1
theta( -3.1415927 ) = 0.0000000
sinc( -3.1415927 ) = -2.78275341E-08
rect( 0.0000000 , 1.0000000 ) = 1.0000000
```

Something is going wrong, isn’t it?

Seems like one function changed its actual argument!
Testing DSP Functions

- Let’s build our test program putting all together:

  user@cineca$> gfortran dsp.f90 dsp_test.f90 -o dsp_test

  -o option specifies the name dsp_test for the executable

- Now run the program:

  user@cineca$> ./dsp_test
  Enter i, j, k:
  -1 0 1

  theta ( -3.1415927 ) = 0.0000000
  sinc ( -3.1415927 ) = -2.78275341E-08
  rect ( 0.0000000 , 1.0000000 ) = 1.0000000

  Something is going wrong, isn’t it?
  Seems like one function changed its actual argument!
Testing DSP Functions

Let’s build our test program putting all together:

```
user@cineca$> gfortran dsp.f90 dsp_test.f90 -o dsp_test
```

- `-o` option specifies the name `dsp_test` for the executable

Now run the program:

```
user@cineca$> ./dsp_test
Enter i, j, k:
-1 0 1
theta( 3.1415927 ) = 0.0000000
sinc( 3.1415927 ) = -2.78275341E-08
rect( 0.0000000 , 1.0000000 ) = 1.0000000
```
Testing DSP Functions

- Let’s build our test program putting all together:
  ```
  user@cineca$> gfortran dsp.f90 dsp_test.f90 -o dsp_test
  ```
  - `–o` option specifies the name `dsp_test` for the executable

- Now run the program:
  ```
  user@cineca$> ./dsp_test
  Enter i, j, k:
  -1 0 1
  theta( -3.1415927 ) = 0.0000000
  sinc( -3.1415927 ) = -2.78275341E-08
  rect( 0.0000000 , 1.0000000 ) = 1.0000000
  ```
  - Something is going wrong, isn’t it?
    - Seems like one function changed its actual argument!
Let’s put the code in a file named `dsp.f90`

Best practice: always put different groups of related functions in different files

- Helps to tame complexity
- You can always pass all source files to the compiler
- And you’ll learn to do better ...

And let’s write a program to test all functions

- And be wary, check again actual arguments after all function calls

Best practice: always write a special purpose program to test each subset of functions

- Best to include in the program automated testing of all relevant cases
- Let’s do by hand with I/O for now, to make it short
function theta(x) ! Heaviside function, useful in DSP
    implicit none
    real :: theta
    real, intent(in) :: x

    theta = 1.0
    if (x < 0.0) theta = 0.0
end function theta

function sinc(x) ! sinc function as used in DSP
    implicit none
    real :: sinc
    real, intent(in) :: x
    real, parameter :: pi = acos(-1.0)

    x = x*pi
    sinc = 1.0
    if (x /= 0.0) sinc = sin(x)/x
end function sinc

function rect(t, tau) ! generalized rectangular function, useful in DSP
    implicit none
    real :: rect
    real, intent(in) :: t, tau
    real :: abs_t, half_tau
    real, external :: theta

    abs_t = abs(t)
    half_tau = 0.5*tau
    rect = 0.5
    if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect
Testing DSP Functions Again

- Try to recompile `dsp.f90`...

```
user@cineca$ gfortran -o dsp_test dsp_test.f90 dsp.f90
```

```
dsp.f90:16.2:
x = x*pi
1
Error: Cannot assign to INTENT(IN) variable 'x' at (1)
```

- Got a compiler error message? Good!
Testing DSP Functions Again

- Try to recompile `dsp.f90`...
- Now compiler will check if you respect your stated intents:

```bash
user@cineca$> gfortran -o dsp_test dsp_test.f90 dsp.f90
dsp.f90:16.2:
    x = x*π
    1
Error: Cannot assign to INTENT(IN) variable ‘x’ at (1)
```
Testing DSP Functions Again

- Try to recompile `dsp.f90`...
- Now compiler will check if you respect your stated intents:

```bash
user@cineca$> gfortran -o dsp_test dsp_test.f90 dsp.f90

dsp.f90:16.2:
  x = x*pi
  1
Error: Cannot assign to INTENT(IN) variable 'x' at (1)
```

- Got a compiler error message? Good!
It’s Pass by Reference!

- Arguments are passed *by reference* in Fortran
It’s Pass by Reference!

- Arguments are passed by reference in Fortran
  - Dummy and actual arguments share the same memory locations
Arguments are passed *by reference* in Fortran

- Dummy and actual arguments share the same memory locations
- (And if you pass a constant or expression, an unnamed variable is created for you)
It’s Pass by Reference!

- Arguments are passed *by reference* in Fortran
  - Dummy and actual arguments share the same memory locations
  - (And if you pass a constant or expression, an unnamed variable is created for you)
  - When a dummy argument is assigned to, the actual argument is assigned to
Arguments are passed *by reference* in Fortran
- Dummy and actual arguments share the same memory locations
- (And if you pass a constant or expression, an unnamed variable is created for you)
- When a dummy argument is assigned to, the actual argument is assigned to
- This is a great feature, but a source of bugs too (particularly for C programmers)
It’s Pass by Reference!

- Arguments are passed *by reference* in Fortran
  - Dummy and actual arguments share the same memory locations
  - (And if you pass a constant or expression, an unnamed variable is created for you)
  - When a dummy argument is assigned to, the actual argument is assigned to
  - This is a great feature, but a source of bugs too (particularly for C programmers)
  - And it’s one possible *side effect* you’ll have to watch over
It’s Pass by Reference!

- Arguments are passed *by reference* in Fortran
  - Dummy and actual arguments share the same memory locations
  - (And if you pass a constant or expression, an unnamed variable is created for you)
  - When a dummy argument is assigned to, the actual argument is assigned to
  - This is a great feature, but a source of bugs too (particularly for C programmers)
  - And it’s one possible *side effect* you’ll have to watch over

- Best practice: always give dummy arguments the proper attribute
Arguments are passed *by reference* in Fortran

- Dummy and actual arguments share the same memory locations
- (And if you pass a constant or expression, an unnamed variable is created for you)
- When a dummy argument is assigned to, the actual argument is assigned to
- This is a great feature, but a source of bugs too (particularly for C programmers)
- And it’s one possible *side effect* you’ll have to watch over

Best practice: always give dummy arguments the proper attribute

- `intent(in)` for those you only plan to read values from
It’s Pass by Reference!

- Arguments are passed *by reference* in Fortran
  - Dummy and actual arguments share the same memory locations
  - (And if you pass a constant or expression, an unnamed variable is created for you)
  - When a dummy argument is assigned to, the actual argument is assigned to
  - This is a great feature, but a source of bugs too (particularly for C programmers)
  - And it’s one possible *side effect* you’ll have to watch over

- Best practice: always give dummy arguments the proper attribute
  - *intent*(in) for those you only plan to read values from
  - *intent*(out) for those you only plan to write values to
It’s Pass by Reference!

- Arguments are passed *by reference* in Fortran
  - Dummy and actual arguments share the same memory locations
  - (And if you pass a constant or expression, an unnamed variable is created for you)
  - When a dummy argument is assigned to, the actual argument is assigned to
  - This is a great feature, but a source of bugs too (particularly for C programmers)
  - And it’s one possible *side effect* you’ll have to watch over

- Best practice: always give dummy arguments the proper attribute
  - `intent(in)` for those you only plan to read values from
  - `intent(out)` for those you only plan to write values to
  - `intent(inout)` (default) for those you plan to do both
function theta(x) !Heaviside function, useful in DSP
  implicit none
  real :: theta
  real, intent(in) :: x

  theta = 1.0
  if (x < 0.0 ) theta = 0.0
end function theta

function sinc(x) !sinc function as used in DSP
  implicit none
  real :: sinc, xpi
  real, intent(in) :: x
  real, parameter :: pi = acos(-1.0)

  xpi = x*pi
  sinc = 1.0
  if (xpi /= 0.0) sinc = sin(xpi)/xpi
end function sinc

function rect(t, tau) !generalized rectangular function, useful in DSP
  implicit none
  real :: rect
  real, intent(in) :: t, tau
  real :: abs_t, half_tau
  real, external :: theta

  abs_t = abs(t)
  half_tau = 0.5*tau
  rect = 0.5
  if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect
Testing DSP Function the Last Time

► Way much better!

user@cineca$> gfortran -o dsp_test dsp_test.f90 dsp.f90
user@cineca$> ./dsp_test
  Enter i, j, k:
-1 0 1
  theta(  -1.0000000       ) =  0.0000000
  sinc(   -1.0000000       ) =  -2.78275341E-08
  rect(    0.0000000 ,  1.0000000     ) =  1.0000000
Testing DSP Function the Last Time

- Way much better!

```
user@cineca$> gfortran -o dsp_test dsp_test.f90 dsp.f90
user@cineca$> ./dsp_test
    Enter i, j, k:
    -1 0 1
    theta(  -1.0000000  ) =  0.0000000
    sinc(   -1.0000000  ) =  -2.78275341E-08
    rect(  0.0000000 ,  1.0000000  ) =  1.0000000
```

- Now comment out `real :: i, j, k` in `dsp_test.f90`, recompile and rerun
Testing DSP Function the Last Time

- Way much better!

```bash
user@cineca$> gfortran -o dsp_test dsp_test.f90 dsp.f90
user@cineca$> ./dsp_test
  Enter i, j, k:
-1 0 1
  theta(-1.0000000000) = 0.0000000
  sinc(-1.0000000000) = -2.78275341E-08
  rect(0.0000000000, 1.0000000000) = 1.0000000
```

- Now comment out `real :: i, j, k` in `dsp_test.f90`, recompile and rerun

- Now add `implicit none` to `dsp_test.f90` and do it again
Try to pass integer variables as actual arguments to \texttt{theta()}, \texttt{sinc()}, and \texttt{rect()}
Try to pass `integer` variables as actual arguments to `theta()`, `sinc()`, and `rect()`

Got some surprising behavior?
Try to pass `integer` variables as actual arguments to `theta()`, `sinc()`, and `rect()`

Got some surprising behavior?

Our testing program doesn’t know enough about external functions it is calling

- It is knowledgeable about return types
- It is totally ignorant about argument types
Try to pass `integer` variables as actual arguments to `theta()`, `sinc()`, and `rect()`

Got some surprising behavior?

Our testing program doesn’t know enough about external functions it is calling

- It is knowledgeable about return types
- It is totally ignorant about argument types

We can make it aware using `interface` blocks
program dsp

  implicit none

  real :: i,j,k

  real, external :: theta, sinc, rect

  print *, 'Enter i, j, k:'
  read(*,*) i, j, k

  write(*,*) 'theta(', i, ')= ', theta(i)
  write(*,*) 'sinc(', i , ')= ', sinc(i)
  write(*,*) 'rect(', j, ',', k, ')= ', rect(j,k)

end program dsp
program dsp

  implicit none

  real :: i, j, k

  interface
    function theta(x)
      real :: theta, x
    end function theta
  end interface

  interface
    function sinc(x)
      real :: sinc, x
    end function sinc
  end interface

  interface
    function rect(t, tau)
      real :: rect, t, tau
    end function rect
  end interface

  print *, 'Enter i, j, k:
  read(*,*) i, j, k

  write(*,*) 'theta(', i, ')= ', theta(i)
  write(*,*) 'sinc(', i ,')= ', sinc(i)
  write(*,*) 'rect(', j, ',', k, ')= ', rect(j,k)

end program dsp
Ignorance is Evil

- Try to pass `integer` variables as actual arguments to `theta()`, `sinc()`, and `rect()`
- Got some surprising behavior?
- Our testing program doesn’t know enough about external functions it is calling
  - It is knowledgeable about return types
  - It is totally ignorant about argument types

- We can make it aware using `interface` blocks
  - Just type it in each program unit calling dsp functions
Try to pass `integer` variables as actual arguments to `theta()`, `sinc()`, and `rect()`

Got some surprising behavior?

Our testing program doesn’t know enough about external functions it is calling

- It is knowledgeable about return types
- It is totally ignorant about argument types

We can make it aware using `interface` blocks

- Just type it in each program unit calling dsp functions
- Or, if your life is too short for typing, copy and paste it
Try to pass `integer` variables as actual arguments to `theta()`, `sinc()`, and `rect()`

Got some surprising behavior?

Our testing program doesn’t know enough about external functions it is calling
  - It is knowledgeable about return types
  - It is totally ignorant about argument types

We can make it aware using `interface` blocks
  - Just type it in each program unit calling `dsp` functions
  - Or, if your life is too short for typing, copy and paste it
  - But life is too short to modify interfaces spread around 56 program units
Ignorance is Evil

- Try to pass `integer` variables as actual arguments to `theta()`, `sinc()`, and `rect()`
- Got some surprising behavior?
- Our testing program doesn’t know enough about external functions it is calling
  - It is knowledgeable about return types
  - It is totally ignorant about argument types

- We can make it aware using `interface` blocks
  - Just type it in each program unit calling dsp functions
  - Or, if your life is too short for typing, copy and paste it
  - But life is too short to modify interfaces spread around 56 program units
  - Good, but still error prone, no better way?
**use Modules, Instead!**

- Modules are the Fortran way to complete and robust management of sets of related routines and more
module dsp
    implicit none
contains
    function theta(x) ! Heaviside function, useful in DSP
        real :: theta
        real, intent(in) :: x

        theta = 1.0
        if (x < 0.0 ) theta = 0.0
    end function theta

    function sinc(x) ! sinc function as used in DSP
        real :: sinc, xpi
        real, intent(in) :: x
        real, parameter :: pi = acos(-1.0)

        xpi = x*pi
        sinc = 1.0
        if (xpi /= 0.0) sinc = sin(xpi)/xpi
    end function sinc

    function rect(t, tau) ! generalized rectangular function, useful in DSP
        real :: rect
        real, intent(in) :: t, tau
        real :: abs_t, half_tau

        abs_t = abs(t)
        half_tau = 0.5*tau
        rect = 0.5
        if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
    end function rect
end module dsp
module dsp
  implicit none
contains
  function theta(x) ! Heaviside function, useful in DSP
    real :: theta
    real, intent(in) :: x

    theta = 1.0
    if (x < 0.0 ) theta = 0.0
  end function theta

  function sinc(x) ! sinc function as used in DSP
    real :: sinc, xpi
    real, intent(in) :: x
    real, parameter :: pi = acos(-1.0)

    xpi = x*pi
    sinc = 1.0
    if (xpi /= 0.0) sinc = sin(xpi)/xpi
  end function sinc

  function rect(t, tau) ! generalized rectangular function, useful in DSP
    real :: rect
    real, intent(in) :: t, tau
    real :: abs_t, half_tau

    abs_t = abs(t)
    half_tau = 0.5*tau
    rect = 0.5
    if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
  end function rect
end module dsp
use Modules, Instead!

- Modules are the Fortran way to complete and robust management of sets of related routines and more
- Interfaces are automatically defined for each procedure a module contains
- To use `theta()`, `sinc()`, and `rect()` in a program unit:
  - just add a `use dsp` statement
  - before you declare anything else in the unit
use Modules, Instead!

- Modules are the Fortran way to complete and robust management of sets of related routines and more
- Interfaces are automatically defined for each procedure a module contains
- To use `theta()`, `sinc()`, and `rect()` in a program unit:
  - just add a `use dsp` statement
  - before you declare anything else in the unit
- Try it now!
module dsp
  implicit none
contains
  function theta(x) !Heaviside function, useful in DSP
    real :: theta
    real, intent(in) :: x

    theta = 1.0
    if (x < 0.0 ) theta = 0.0
  end function theta

  function sinc(x) !sinc function as used in DSP
    real :: sinc, xpi
    real, intent(in) :: x
    real, parameter :: pi = acos(-1.0)

    xpi = x*pi
    sinc = 1.0
    if (xpi /= 0.0) sinc = sin(xpi)/xpi
  end function sinc

  function rect(t, tau) !generalized rectangular function, useful in DSP
    real :: rect
    real, intent(in) :: t, tau
    real :: abs_t, half_tau

    abs_t = abs(t)
    half_tau = 0.5*tau
    rect = 0.5
    if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
  end function rect
end module dsp
use Modules, Instead!

- Modules are the Fortran way to complete and robust management of sets of related routines and more
- Interfaces are automatically defined for each procedure a module contains
- To use \texttt{theta()}, \texttt{sinc()}, and \texttt{rect()} in a program unit:
  - just add a \texttt{use dsp} statement
  - before you declare anything else in the unit
- Try it now!
- Best practices
use Modules, Instead!

- Modules are the Fortran way to complete and robust management of sets of related routines and more
- Interfaces are automatically defined for each procedure a module contains
- To use \texttt{theta()}, \texttt{sinc()}, and \texttt{rect()} in a program unit:
  - just add a \texttt{use dsp} statement
  - before you declare anything else in the unit
- Try it now!
- Best practices
  - If you have a set of related procedures, always make a module
use Modules, Instead!

- Modules are the Fortran way to complete and robust management of sets of related routines and more.
- Interfaces are automatically defined for each procedure a module contains.
- To use \texttt{theta()}, \texttt{sinc()}, and \texttt{rect()} in a program unit:
  - just add a \texttt{use dsp} statement
  - before you declare anything else in the unit
- Try it now!

- Best practices
  - If you have a set of related procedures, always make a module.
  - If you have a single procedure, just to tame code complexity, called by a single program unit, a module could be overkill.
use Modules, Instead!

- Modules are the Fortran way to complete and robust management of sets of related routines and more
- Interfaces are automatically defined for each procedure a module contains
- To use \texttt{theta()}, \texttt{sinc()}, and \texttt{rect()} in a program unit:
  - just add a \texttt{use dsp} statement
  - before you declare anything else in the unit
- Try it now!
- Best practices
  - If you have a set of related procedures, always make a module
  - If you have a single procedure, just to tame code complexity, called by a single program unit, a module could be overkill
- But there is a lot more to say about modules
A nice colleague handed you the `dsp` module...

but you prefer your own version of `rect()`, which returns 1 on borders:

- don’t change the module source
A nice colleague handed you the `dsp` module...

but you prefer your own version of `rect()`, which returns 1 on borders:

- don’t change the module source
- use `dsp`, only: `theta`, `sinc` and keep using your own `rect()"
Modules Give You Fine Control

- A nice colleague handed you the `dsp` module...

- but you prefer your own version of `rect()`, which returns 1 on borders:
  - don’t change the module source
  - use `dsp, only : theta, sinc`
    and keep using your own `rect()`

- or you already have a function called `theta()`, called all over your code, and don’t want to change it:
Modules Give You Fine Control

▶ A nice colleague handed you the dsp module...

▶ but you prefer your own version of rect(), which returns 1 on borders:
  ▶ don’t change the module source
  ▶ use dsp, only: theta, sinc
    and keep using your own rect()

▶ or you already have a function called theta(), called all over your code, and don’t want to change it:
  ▶ rename the theta() function in dsp like this:
    use dsp, heaviside=>theta
A nice colleague handed you the **dsp** module...

but you prefer your own version of **rect()**, which returns 1 on borders:

- don’t change the module source
- **use dsp, only : theta, sinc**
  and keep using your own **rect()**

or you already have a function called **theta()**, called all over your code, and don’t want to change it:

- rename the **theta()** function in **dsp** like this:
  
  ```
  use dsp, heaviside=>theta
  ```

or maybe both:
A nice colleague handed you the `dsp` module...

but you prefer your own version of `rect()`, which returns 1 on borders:

- don’t change the module source
- use `dsp, only : theta, sinc`
  and keep using your own `rect()`

or you already have a function called `theta()`, called all over your code, and don’t want to change it:

- rename the `theta()` function in `dsp` like this:
  ```
  use dsp, heaviside=>theta
  ```

or maybe both:

- use `dsp, only : heaviside=>theta, sinc`
function rect(t, tau)
    implicit none
    real :: rect
    real, intent(in) :: t, tau
    real :: abs_t, half_tau

    abs_t = abs(t)
    half_tau = 0.5*tau
    rect = 0.5
    if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect

▶ What if \texttt{rect()} is passed a negative argument for \texttt{tau}?
Managing Wrong Arguments

function rect(t, tau)
    implicit none
    real :: rect
    real, intent(in) :: t, tau
    real :: abs_t, half_tau

    abs_t = abs(t)
    half_tau = 0.5*tau
    rect = 0.5
    if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect

▶ What if \texttt{rect()} is passed a negative argument for \texttt{tau}?
▶ Wrong results
function rect(t, tau)
    implicit none
    real :: rect
    real, intent(in) :: t, tau
    real :: abs_t, half_tau

    abs_t = abs(t)
    half_tau = 0.5*tau
    rect = 0.5
    if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect

▶ What if rect() is passed a negative argument for tau?
    ▶ Wrong results
    ▶ Taking the absolute value of tau it’s a possibility
Managing Wrong Arguments

```fortran
function rect(t, tau)
    implicit none
    real :: rect
    real, intent(in) :: t, tau
    real :: abs_t, half_tau

    abs_t = abs(t)
    half_tau = 0.5*tau
    rect = 0.5
    if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect
```

- What if `rect()` is passed a negative argument for `tau`?
  - Wrong results
- Taking the absolute value of `tau` it’s a possibility
- But not a good one, because:
  - a negative rectangle width is nonsensical
  - probably flags a mistake in the calling code
  - and a zero rectangle width is also a problem
function rect(t, tau)
    implicit none
    real :: rect
    real, intent(in) :: t, tau
    real :: abs_t, half_taus

    if (tau <= 0.0) stop 'rect() non positive second argument'
    abs_t = abs(t)
    half_taus = 0.5*tau
    rect = 0.5
    if (abs_t /= half_taus) rect = theta(half_taus-abs_t)
end function rect

▶ A known approach...
function rect(t, tau)
    implicit none
    real :: rect
    real, intent(in) :: t, tau
    real :: abs_t, half_tau

    if (tau <= 0.0) stop 'rect() non positive second argument'
    abs_t = abs(t)
    half_tau = 0.5*tau
    rect = 0.5
    if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect

▶ A known approach...
▶ but too crude!
▶ No clue at the argument value
▶ No clue at which call to rect() was wrong
▶ And stopping a program in a procedure, called by another procedure, called by another procedure, ..., is widely reputed bad programming practice
module dsp
  implicit none
  integer :: dsp_info
  integer, parameter :: DSPERR_DOMAIN = 1
contains
  function theta(x) ! Heaviside function, useful in DSP
  ! code as in previous examples...
  end function theta

  function sinc(x) ! sinc function as used in DSP
  ! code as in previous examples...
  end function sinc

  function rect(t, tau) ! generalized rectangular function, useful in DSP
  real :: rect
  real, intent(in) :: t, tau
  real :: abs_t, half_tau

  if (tau <= 0.0) then
    dsp_info = DSPERR_DOMAIN
    rect = 0.0
    return
  end if

  abs_t = abs(t)
  half_tau = 0.5*tau
  rect = 0.5
  if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect
end module dsp
module dsp
    implicit none
    integer :: dsp_info
    integer, parameter :: DSPERR_DOMAIN = 1
contains
    function theta(x) ! Heaviside function, useful in DSP
    ! code as in previous examples...
    end function theta

    function sinc(x) ! sinc function as used in DSP
    ! code as in previous examples...
    end function sinc

    function rect(t, tau) ! generalized rectangular function, useful in DSP
    real :: rect
    real, intent(in) :: t, tau
    real :: abs_t, half_tau

    if (tau <= 0.0) then
        dsp_info = DSPERR_DOMAIN
        rect = 0.0
        return
    end if

    abs_t = abs(t)
    half_tau = 0.5*tau
    rect = 0.5
    if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect
end module dsp
More Module Power, and More Types

- Yes, a module can define variables, too

- And they will be accessible to all program units using it

- Yes, integer is another Fortran type for variables hosting integer numerical values

- More on this later...

- return forces function execution to terminate and return to calling unit
More Module Power, and More Types

- Yes, a module can define variables, too
- And they will be accessible to all program units using it
module dsp
  implicit none
  integer :: dsp_info
  integer, parameter :: DSPERR_DOMAIN = 1
contains
  function theta(x) !Heaviside function, useful in DSP
  ! code as in previous examples...
  end function theta
  
  function sinc(x) !sinc function as used in DSP
  ! code as in previous examples...
  end function sinc
  
  function rect(t, tau) !generalized rectangular function, useful in DSP
  real :: rect
  real, intent(in) :: t, tau
  real :: abs_t, half_tau

  if (tau <= 0.0) then
    dsp_info = DSPERR_DOMAIN
    rect = 0.0
    return
  end if
  
  abs_t = abs(t)
  half_tau = 0.5*tau
  rect = 0.5
  if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
  end function rect
end module dsp
More Module Power, and More Types

- Yes, a module can define variables, too
- And they will be accessible to all program units using it
- And yes, `integer` it’s another Fortran type
  - For variables hosting integer numerical values
  - More on this later...
module dsp
    implicit none
    integer :: dsp_info
    integer, parameter :: DSPERR_DOMAIN = 1
contains
    function theta(x) ! Heaviside function, useful in DSP
    ! code as in previous examples...
    end function theta

    function sinc(x) ! sinc function as used in DSP
    ! code as in previous examples...
    end function sinc

    function rect(t, tau) ! generalized rectangular function, useful in DSP
    real :: rect
    real, intent(in) :: t, tau
    real :: abs_t, half_tau

    if (tau <= 0.0) then
        dsp_info = DSPERR_DOMAIN
        rect = 0.0
        return
    end if

    abs_t = abs(t)
    half_tau = 0.5*tau
    rect = 0.5
    if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect
end module dsp
More Module Power, and More Types

- Yes, a module can define variables, too
- And they will be accessible to all program units using it
- And yes, integer it’s another Fortran type
  - For variables hosting integer numerical values
  - More on this later...
- And yes, return forces function execution to terminate and return to calling unit
module dsp
    implicit none
    integer :: dsp_info
    integer, parameter :: DSPERR_DOMAIN = 1
contains
    function theta(x) ! Heaviside function, useful in DSP
    end function theta

    function sinc(x) ! sinc function as used in DSP
    end function sinc

    function rect(t, tau) ! generalized rectangular function, useful in DSP
    real :: rect
    real, intent(in) :: t, tau
    real :: abs_t, half_tau

    if (tau <= 0.0) then
        dsp_info = DSPERR_DOMAIN
        rect = 0.0
        return
    end if

    abs_t = abs(t)
    half_tau = 0.5*tau
    rect = 0.5
    if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect
end module dsp
Error Management Strategy

- Set a module variable to a constant corresponding to the error class
- And return a sensible result

```fortran
dsp_info = 0
r = rect(x, width)
if (dsp_info == DSPERR_DOMAIN) then
  ! take corrective action or fail gracefully
end if
```

Note: even if Fortran ignores case, constants are often highlighted using all capitals
Error Management Strategy

- Set a module variable to a constant corresponding to the error class
- And return a sensible result
- Then a wise user would do something like this:

```fortran
  dsp_info = 0
  r = rect(x, width)
  if (dsp_info == DSPERR_DOMAIN) then
    ! take corrective action or fail gracefully
  end if
```
Error Management Strategy

- Set a module variable to a constant corresponding to the error class
- And return a sensible result
- Then a wise user would do something like this:

```fortran
dsp_info = 0
r = rect(x, width)
if (dsp_info == DSPERR_DOMAIN) then
  ! take corrective action or fail gracefully
end if
```

- Note: even if Fortran ignores case, constants are often highlighted using all capitals
module dsp
    implicit none
    integer, parameter :: DSPERR_DOMAIN = 1
contains

! ...

function rect(t, tau, info) ! generalized rectangular function, useful in DSP
    real :: rect
    real, intent(in) :: t, tau
    integer, intent(out) :: info
    real :: abs_t, half_tau

    info = 0
    if (tau <= 0.0) then
        info = DSPERR_DOMAIN
        rect = 0.0
        return
    end if

    abs_t = abs(t)
    half_tau = 0.5*tau
    rect = 0.5
    if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect
end module dsp
module dsp
    implicit none
    integer, parameter :: DSPERR_DOMAIN = 1
contains

    ! ...

    function rect(t, tau, info) ! generalized rectangular function, useful in DSP
        real :: rect
        real, intent(in) :: t, tau
        integer, intent(out) :: info
        real :: abs_t, half_tau

        info = 0
        if (tau <= 0.0) then
            info = DSPERR_DOMAIN
            rect = 0.0
            return
        end if

        abs_t = abs(t)
        half_tau = 0.5*tau
        rect = 0.5
        if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
    end function rect
end module dsp
Using Arguments to Return Error Codes

- Set a dedicated argument to a constant corresponding to the error class
- And return a sensible result
Using Arguments to Return Error Codes

- Set a dedicated argument to a constant corresponding to the error class
- And return a sensible result
- Then a wise user would do something like this:

```cpp
r = rect(x, width, rect_info)
if (rect_info == DSPERR_DOMAIN) then
  ! take corrective action or fail gracefully
end if
```
Using Arguments to Return Error Codes

- Set a dedicated argument to a constant corresponding to the error class
- And return a sensible result
- Then a wise user would do something like this:

```plaintext
r = rect(x, width, rect_info)
if (rect_info == DSPERR_DOMAIN) then
  ! take corrective action or fail gracefully
end if
```

- But this is annoying when the arguments are guaranteed to be correct
  - `info` can be given the `optional` attribute
  - and omitted when you feel it's safe: `rect(x, 5.0)`
module dsp
  implicit none
  integer, parameter :: DSPERR_DOMAIN = 1
contains

! ...

function rect(t, tau, info) !generalized rectangular function, useful in DSP
  real :: rect
  real, intent(in) :: t, tau
  integer, intent(out), optional :: info
  real :: abs_t, half_tau

  if (present(info)) info = 0
  if (tau <= 0.0) then
    if (present(info)) info = DSPERR_DOMAIN
    rect = 0.0
    return
  end if

  abs_t = abs(t)
  half_tau = 0.5*tau
  rect = 0.5
  if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect
end module dsp
module dsp
    implicit none
    integer, parameter :: DSPERR_DOMAIN = 1
contains

! ...

function rect(t, tau, info) ! generalized rectangular function, useful in DSP
    real :: rect
    real, intent(in) :: t, tau
    integer, intent(out), optional :: info
    real :: abs_t, half_tau

    if (present(info)) info = 0
    if (tau <= 0.0) then
        if (present(info)) info = DSPERR_DOMAIN
        rect = 0.0
        return
    end if

    abs_t = abs(t)
    half_tau = 0.5*tau
    rect = 0.5
    if (abs_t /= half_tau) rect = theta(half_tau-abs_t)
end function rect
end module dsp
Total Robustness

- Your platform could support IEEE floating point standard
  - Most common ones do, at least in a good part

- This means more bad cases:
  - One of the arguments is a NaN
  - Both arguments are infinite (they are not ordered!)

- Best strategy: return a NaN and set `dsp_info` in these bad cases
  - And do it also for non positive values of `tau`

- But then the floating point environment configuration should be checked, proper floating point exceptions set...

- Being absolutely robust is difficult
  - Too advanced stuff to cover in this course
  - But not an excuse, some robustness is better than none

- It's a process to do in steps
  - Always comment in your code bad cases you don't cover yet!
Total Robustness

- Your platform could support IEEE floating point standard
  - Most common ones do, at least in a good part

- This means more bad cases:
  - one of the arguments is a NaN
  - both arguments are infinite (they are not ordered!)

- Best strategy: return a NaN and set `dsp_info` in these bad cases
- And do it also for non positive values of `tau`
- But then the floating point environment configuration should be checked, proper floating point exceptions set...

- Being absolutely robust is difficult
- Too advanced stuff to cover in this course
- But not an excuse, some robustness is better than none
- It’s a process to do in steps
- Always comment in your code bad cases you don’t cover yet!
Total Robustness

- Your platform could support IEEE floating point standard
  - Most common ones do, at least in a good part
- This means more bad cases:
  - one of the arguments is a NaN
  - both arguments are infinite (they are not ordered!)
- Best strategy: return a NaN and set `dsp_info` in these bad cases
  - And do it also for non positive values of `tau`
  - But then the floating point environment configuration should be checked, proper floating point exceptions set...
Total Robustness

- Your platform could support IEEE floating point standard
  - Most common ones do, at least in a good part
- This means more bad cases:
  - one of the arguments is a NaN
  - both arguments are infinite (they are not ordered!)
- Best strategy: return a NaN and set `dsp_info` in these bad cases
  - And do it also for non positive values of `tau`
  - But then the floating point environment configuration should be checked, proper floating point exceptions set...

- Being absolutely robust is difficult
  - Too advanced stuff to cover in this course
  - But not an excuse, some robustness is better than none
  - It’s a process to do in steps
  - Always comment in your code bad cases you don’t cover yet!
Outline

Introduction

Fortran Basics

More Fortran Basics
  My First Fortran Functions
  Making it Correct
  Making it Robust
  Copying with Legacy
  Wrapping it Up 2

Integer Types and Iterating

More on Compiling and Linking

Homeworks
FUNCTION SINC(X)
  IMPLICIT NONE
  REAL SINC, X, XPI
  REAL PI
  PARAMETER (PI = 3.1415926)

  XPI = X*PI
  SINC = 1.0
  IF (XPI .NE. 0.0) SINC = SIN(XPI)/XPI
END

FUNCTION RECT(T, TAU)
  IMPLICIT NONE
  REAL RECT, T, TAU
  REAL ABS_T, HALF_TAU
  REAL THETA
  EXTERNAL THETA
  INTEGER DSPINFO
  COMMON /DSP/ DSPINFO

  IF (TAU .LE. 0.0) THEN
    DSPINFO = 1
    RECT = 0.0;
    RETURN
  END IF

  ABS_T = ABS(T)
  HALF_TAU = 0.5*TAU
  RECT = 0.5
  IF (ABS_T .NE. HALF_TAU) RECT = THETA(HALF_TAU-ABS_T)
END
Many Things are Missing

- Strange looking relational operators
- No attributes
  - Declarations spread over many lines, error prone
- No initialization expressions
  - You had to type in the actual number
- No **intent** i.e. no defense from subtle bugs
- No **interface**
Many Things are Missing

- Strange looking relational operators
- No attributes
  - Declarations spread over many lines, error prone
- No initialization expressions
  - You had to type in the actual number
- No intent i.e. no defense from subtle bugs
- No interface

No easy way to share variables among program units
- To share you had to use common statements
- And type in variable types and common statements in each unit
- And the smallest mistake can turn into a nightmare
Many Things are Missing

- Strange looking relational operators
- No attributes
  - Declarations spread over many lines, error prone
- No initialization expressions
  - You had to type in the actual number
- No intent i.e. no defense from subtle bugs
- No interface
- No easy way to share variables among program units
  - To share you had to use `common` statements
  - And type in variable types and `common` statements in each unit
  - And the smallest mistake can turn into a nightmare
- Bottom line:
  - Is `common` good or bad? The jury is still out
  - We’ll not cover them, but you’ll encounter them
  - Read the fine print, or better switch to modules, they are way much better
You are lucky, and inherit a 4000 lines of code library, coming from the dark ages
  ▶ Tested and tried
Refurbishing Old Code

- You are lucky, and inherit a 4000 lines of code library, coming from the dark ages
  - Tested and tried
- But no interface
  - Thus no compiler checks when you call it
  - And rewriting a working code in modern language is soooo dangerous...
Refurbishing Old Code

- You are lucky, and inherit a 4000 lines of code library, coming from the dark ages
  - Tested and tried
- But no interface
  - Thus no compiler checks when you call it
  - And rewriting a working code in modern language is soooo dangerous...

- Modules come to rescue
  - They don’t need to include the actual code
  - But they can publish an interface for code which is elsewhere
  - And then you can use the module in calling program units
module dspmod

  implicit none

  interface
    function theta(x)
      real :: theta
      real, intent(in) :: x
    end function theta
  end interface

  interface
    function sinc(x)
      real :: sinc
      real, intent(in) :: x
    end function sinc
  end interface

  interface
    function rect(t, tau)
      real :: rect
      real, intent(in) :: t, tau
    end function rect
  end interface

end module dspmod
Outline

Introduction

Fortran Basics

More Fortran Basics
   My First Fortran Functions
   Making it Correct
   Making it Robust
   Copying with Legacy
   Wrapping it Up 2

Integer Types and Iterating

More on Compiling and Linking

Homeworks
A program can be subdivided in more source files
Functions and their arguments
Arguments are passed to functions by reference
\texttt{intent} attribute is precious to prevent subtle bugs
Intrinsic and external procedures are two different things
\texttt{parameter} variables
Explicit interfaces
Modules allow complete management of procedures
Modules allow access to variables from many program units
Modules can be used to make proper use of legacy, reliable codes
Best Practices

- Always name constants
- Test every function you write
  - Writing specialized programs to do it
- Use language support and compiler to catch mistakes
- Use explicit interfaces
- Use modules
- Describe all attributes of a variable at declaration
- Anticipate causes of problems
  - Find a rational way to react
  - Fail predictably and in a user friendly way
  - Robustness it’s a long way to do in steps
  - Comment in your code issues still to address
Outline

Introduction

Fortran Basics

More Fortran Basics

Integer Types and Iterating
  Play it Again, Please
  Testing and Fixing it
  Hitting Limits
  Wider Integer Types
  How Bad it Used to Be
  Wrapping it Up 3

More on Compiling and Linking

Homeworks
Greatest Common Divisor

- **Euclid’s Algorithm**
  1. Take two integers \( a \) and \( b \)
  2. Let \( r \leftarrow a \mod b \)
  3. Let \( a \leftarrow b \)
  4. Let \( b \leftarrow r \)
  5. If \( b \) is not zero, go back to step 2
  6. \( a \) is the GCD

- Let’s implement it and learn some more Fortran
module number_theory
    implicit none
    contains
    function gcd(a, b) ! Greatest Common Divisor
        integer :: gcd
        integer, intent(in) :: a, b
        integer :: gb, t

        gcd = a
        gb = b

        do
            t = mod(gcd,gb)
            gcd = gb
            if (t == 0) exit
            gb = t
        end do
    end function gcd

    function lcm(a, b) ! Least Common Multiple
        integer :: lcm
        integer, intent(in) :: a, b

        lcm = a*b/gcd(a,b)
    end function lcm
end module number_theory
module number_theory

    implicit none
    contains

    function gcd(a, b) ! Greatest Common Divisor
        integer :: gcd
        integer, intent(in) :: a, b
        integer :: gb, t

        gcd = a
        gb = b

        do
            t = mod(gcd,gb)
            gcd = gb
            if (t == 0) exit
            gb = t
        end do
    end function gcd

    function lcm(a, b) ! Least Common Multiple
        integer :: lcm
        integer, intent(in) :: a, b

        lcm = a*b/gcd(a,b)
    end function lcm

end module number_theory
The Integer Type

- As we said, `integer` means that a value is an integer
  - Only integer values, positive, negative or zero
  - On most platforms, `integer` means a 32 bits value, ranging from $-2^{31}$ to $2^{31} - 1$
The Integer Type

- As we said, `integer` means that a value is an integer
  - Only integer values, positive, negative or zero
  - On most platforms, `integer` means a 32 bits value, ranging from $-2^{31}$ to $2^{31} - 1$

- Want to know the actual size?
  - The standard is absolutely generic on this
The Integer Type

- As we said, `integer` means that a value is an integer
  - Only integer values, positive, negative or zero
  - On most platforms, `integer` means a 32 bits value, ranging from $-2^{31}$ to $2^{31} - 1$
- Want to know the actual size?
  - The standard is absolutely generic on this
  - But we’ll tell you a secret...
The Integer Type

- As we said, `integer` means that a value is an integer
  - Only integer values, positive, negative or zero
  - On most platforms, `integer` means a 32 bits value, ranging from \(-2^{31}\) to \(2^{31} - 1\)

- Want to know the actual size?
  - The standard is absolutely generic on this
  - But we’ll tell you a secret...
  - ...on all platforms we know of, the intrinsic function `kind()` will return the size in bytes of any integer expression you’ll pass as an argument
The Integer Type

- As we said, `integer` means that a value is an integer
  - Only integer values, positive, negative or zero
  - On most platforms, `integer` means a 32 bits value, ranging from $-2^{31}$ to $2^{31} - 1$

- Want to know the actual size?
  - The standard is absolutely generic on this
  - But we’ll tell you a secret...
  - ...on all platforms we know of, the intrinsic function `kind()` will return the size in bytes of any integer expression you’ll pass as an argument
  - Try with `kind(0)`, to know the size of a normal `integer`
The Integer Type

- As we said, integer means that a value is an integer
  - Only integer values, positive, negative or zero
  - On most platforms, integer means a 32 bits value, ranging from $-2^{31}$ to $2^{31} - 1$

- Want to know the actual size?
  - The standard is absolutely generic on this
  - But we’ll tell you a secret...
  - ...on all platforms we know of, the intrinsic function `kind()` will return the size in bytes of any integer expression you’ll pass as an argument
  - Try with `kind(0)`, to know the size of a normal integer
  - And works for real values too, or values of any type, for that matter

More on this later

Want to know more?

Intrinsic function `huge(0)` returns the greatest positive value an integer can assume

Again, we’ll be back at this
The Integer Type

- As we said, integer means that a value is an integer
  - Only integer values, positive, negative or zero
  - On most platforms, integer means a 32 bits value, ranging from $-2^{31}$ to $2^{31} - 1$

- Want to know the actual size?
  - The standard is absolutely generic on this
  - But we’ll tell you a secret...
  - ...on all platforms we know of, the intrinsic function \texttt{kind()} will return the size in bytes of any integer expression you’ll pass as an argument
  - Try with \texttt{kind(0)}, to know the size of a normal integer
  - And works for real values too, or values of any type, for that matter
  - More on this later
The Integer Type

- As we said, **integer** means that a value is an integer
  - Only integer values, positive, negative or zero
  - On most platforms, **integer** means a 32 bits value, ranging from $-2^{31}$ to $2^{31} - 1$

- Want to know the actual size?
  - The standard is absolutely generic on this
  - But we’ll tell you a secret...
  - ...on all platforms we know of, the intrinsic function `kind()` will return the size in bytes of any integer expression you’ll pass as an argument
  - Try with `kind(0)`, to know the size of a normal **integer**
  - And works for real values too, or values of any type, for that matter
  - More on this later

- Want to know more?
  - Intrinsic function `huge(0)` returns the greatest positive value an **integer** can assume
  - Again, we’ll be back at this
Outline

Introduction

Fortran Basics

More Fortran Basics

Integer Types and Iterating
  Play it Again, Please
  Testing and Fixing it
  Hitting Limits
  Wider Integer Types
  How Bad it Used to Be
  Wrapping it Up 3

More on Compiling and Linking

Homeworks
module number_theory
    implicit none
contains
    function gcd(a, b) ! Greatest Common Divisor
        integer :: gcd
        integer, intent(in) :: a, b
        integer :: gb, t

        gcd = a
        gb = b

        do
            t = mod(gcd, gb)
            gcd = gb
            if (t == 0) exit
            gb = t
        end do
    end function gcd

    function lcm(a, b) ! Least Common Multiple
        integer :: lcm
        integer, intent(in) :: a, b

        lcm = a*b/gcd(a,b)
    end function lcm
end module number_theory
Iterating with do ... end do

- do
  block of statements
end do

1. Executes again and again the block of statements
2. And does this forever...
3. ... unless exit is executed, forcing execution to proceed at code following end do
Iterating with `do ... end do`

- `do`
  - `block of statements`
  - `end do`
  1. Executes again and again the `block of statements`
  2. And does this forever...
  3. ... unless `exit` is executed, forcing execution to proceed at code following `end do`

- In this specific example:
Iterating with `do  ...  end do`

- `do`
  - `block of statements`
  - `end do`
    1. Executes again and again the `block of statements`
    2. And does this forever...
    3. ... unless `exit` is executed, forcing execution to proceed at code following `end do`

- In this specific example:
  - the code following `end do` is the end of the function
  - thus, we could use `return` instead of `exit`, which is legal,
  - but generally regarded bad practice
Iterating with `do ... end do`

- `do`
  - block of statements
- `end do`
  1. Executes again and again the block of statements
  2. And does this forever...
  3. ... unless `exit` is executed, forcing execution to proceed at code following `end do`

- In this specific example:
  - the code following `end do` is the end of the function
  - thus, we could use `return` instead of `exit`, which is legal,
  - but generally regarded bad practice

- Best practice: do not bail out of a function from inside a loop, particularly a long one
Outline

Introduction

Fortran Basics

More Fortran Basics

Integer Types and Iterating
  Play it Again, Please
  Testing and Fixing it
  Hitting Limits
  Wider Integer Types
  How Bad it Used to Be
  Wrapping it Up 3

More on Compiling and Linking

Homeworks
Put the code in file `numbertheory.f90`

Write a program to test both `gcd()` and `lcm()` on a pair of integer numbers

Test it:
- with pairs of small positive integers
- with the following pairs: 15, 18; -15, 18; 15, -18; -15, -18; 0, 15; 15, 0; 0, 0
module number_theory
  implicit none
  contains
  function gcd(a, b) ! Greatest Common Divisor
    integer :: gcd
    integer, intent(in) :: a, b
    integer :: gb, t

    gcd = a
    gb = b

    do
      t = mod(gcd,gb)
      gcd = gb
      if (t == 0) exit
      gb = t
    end do
  end function gcd

  function lcm(a, b) ! Least Common Multiple
    integer :: lcm
    integer, intent(in) :: a, b

    lcm = a*b/gcd(a,b)
  end function lcm
end module number_theory
Hands-on Session #3

- Put the code in file `numbertheory.f90`
- Write a program to test both `gcd()` and `lcm()` on a pair of integer numbers
- Test it:
  - with pairs of small positive integers
  - with the following pairs: 15, 18; -15, 18; 15, -18; -15, -18; 0, 15; 15, 0; 0, 0
- In some cases, we get wrong results or runtime errors
  - Euclid’s algorithm is only defined for positive integers
Let’s Generalize to the Whole Integer Set

- gcd($a, b$) is non negative, even if $a$ or $b$ is less than zero
  - Taking the absolute value of $a$ and $b$ using `abs()` will do
module number_theory
  implicit none
contains
  function gcd(a, b) ! Greatest Common Divisor
    integer :: gcd
    integer, intent(in) :: a, b
    integer :: gb, t

    gcd = abs(a)
    gb = abs(b)

    do
      t = mod(gcd,gb)
      gcd = gb
      if (t == 0) exit
      gb = t
    end do
  end function gcd

  function lcm(a, b) ! Least Common Multiple
    integer :: lcm
    integer, intent(in) :: a, b

    lcm = a*b/gcd(a,b)
  end function lcm
end module number_theory
Let’s Generalize to the Whole Integer Set

- \( \gcd(a, b) \) is non-negative, even if \( a \) or \( b \) is less than zero
  - Taking the absolute value of \( a \) and \( b \) using \( \text{abs}() \) will do
- \( \gcd(a, 0) \) is \( |a| \)
  - Conditional statements will do
module number_theory
    implicit none
    contains
    function gcd(a, b) ! Greatest Common Divisor
        integer :: gcd
        integer, intent(in) :: a, b
        integer :: gb, t

        gcd = abs(a)
        gb = abs(b)

        if (a == 0) gcd = gb
        if (a == 0 .or. b == 0) return

        do
            t = mod(gcd, gb)
            gcd = gb
            if (t == 0) exit
            gb = t
        end do
    end function gcd

    function lcm(a, b) ! Least Common Multiple
        integer :: lcm
        integer, intent(in) :: a, b

        lcm = a*b/gcd(a, b)
    end function lcm
end module number_theory
Let’s Generalize to the Whole Integer Set

- \( \gcd(a, b) \) is non negative, even if \( a \) or \( b \) is less than zero
  - Taking the absolute value of \( a \) and \( b \) using \( \text{abs()} \) will do
- \( \gcd(a, 0) \) is \( |a| \)
  - Conditional statements will do
- \( \gcd(0, 0) \) is 0
  - Already covered by the previous item, but let’s pay attention to \( \text{lcm()} \)
module number_theory
  implicit none
  contains
    function gcd(a, b) ! Greatest Common Divisor
      integer :: gcd
      integer, intent(in) :: a, b
      integer :: gb, t

      gcd = abs(a)
      gb = abs(b)

      if (a == 0) gcd = gb
      if (a == 0 .or. b == 0) return

      do
        t = mod(gcd,gb)
        gcd = gb
        if (t == 0) exit
        gb = t
      end do
    end function gcd

    function lcm(a, b) ! Least Common Multiple
      integer :: lcm
      integer, intent(in) :: a, b

      if (a == 0 .and. b == 0) then
        lcm = 0 ; return
      end if

      lcm = a*b/gcd(a,b)
    end function lcm
  end module number_theory
Let’s Generalize to the Whole Integer Set

- $\text{gcd}(a, b)$ is non negative, even if $a$ or $b$ is less than zero
  - Taking the absolute value of $a$ and $b$ using $\text{abs}(\cdot)$ will do
- $\text{gcd}(a, 0)$ is $|a|
  - Conditional statements will do
- $\text{gcd}(0, 0)$ is 0
  - Already covered by the previous item, but let’s pay attention to $\text{lcm}(\cdot)$

- By the way:
  - `.and.` and `.or.` combine two logical conditions
  - `;` makes for two statements on the same line: but its use is only justified when space is at a premium, like in slides

Try and test it:

- with pairs of small positive integers
- with the following pairs: 15, 18; -15, 18; 15, -18; -15, -18; 0, 15; 15, 0; 0, 0
- and with the pair: 1000000, 1000000
Let’s Generalize to the Whole Integer Set

- \( \text{gcd}(a, b) \) is non-negative, even if \( a \) or \( b \) is less than zero
  - Taking the absolute value of \( a \) and \( b \) using \( \text{abs()} \) will do
- \( \text{gcd}(a, 0) \) is \( |a| \)
  - Conditional statements will do
- \( \text{gcd}(0, 0) \) is 0
  - Already covered by the previous item, but let’s pay attention to \( \text{lcm()} \)

By the way:

- \( \text{and} \) and \( \text{or} \) combine two logical conditions
- \( ; \) makes for two statements on the same line: but its use is only justified when space is at a premium, like in slides

Try and test it:

- with pairs of small positive integers
- with the following pairs: 15, 18; -15, 18; 15, -18; -15, -18; 0, 15; 15, 0; 0, 0
module number_theory
    implicit none
    contains

    function gcd(a, b) ! Greatest Common Divisor
        integer :: gcd
        integer, intent(in) :: a, b
        integer :: gb, t

        gcd = abs(a)
        gb = abs(b)

        if (a == 0) gcd = gb
        if (a == 0 .or. b == 0) return

        do
            t = mod(gcd, gb)
            gcd = gb
            if (t == 0) exit
            gb = t
        end do
    end function gcd

    function lcm(a, b) ! Least Common Multiple
        integer :: lcm
        integer, intent(in) :: a, b

        if (a == 0 .and. b == 0) then
            lcm = 0 ; return
        end if

        lcm = a*b/gcd(a,b)
    end function lcm
end module number_theory
Let’s Generalize to the Whole Integer Set

- \( \gcd(a, b) \) is non negative, even if \( a \) or \( b \) is less than zero
  - Taking the absolute value of \( a \) and \( b \) using \( \text{abs()} \) will do

- \( \gcd(a, 0) \) is \( |a| \)
  - Conditional statements will do

- \( \gcd(0, 0) \) is 0
  - Already covered by the previous item, but let’s pay attention to \( \text{lcm()} \)

By the way:

- \( .\text{and.} \) and \( .\text{or.} \) combine two logical conditions
- \( ; \) makes for two statements on the same line: but its use is only justified when space is at a premium, like in slides

Try and test it:

- with pairs of small positive integers
- with the following pairs: 15, 18; -15, 18; 15, -18; -15, -18; 0, 15; 15, 0; 0, 0
- and with the pair: 1000000, 1000000
Outline

Introduction

Fortran Basics

More Fortran Basics

Integer Types and Iterating
  Play it Again, Please
  Testing and Fixing it

Hitting Limits
  Wider Integer Types
  How Bad it Used to Be
  Wrapping it Up 3

More on Compiling and Linking

Homeworks
Beware of Type Ranges

- \( a \times b / \text{gcd}(a, b) \) same as \( (a \times b) / \text{gcd}(a, b) \)
Beware of Type Ranges

- $a \times b / \gcd(a, b)$ same as $(a \times b) / \gcd(a, b)$

- What if the result of a calculation cannot be represented in the given type?
  - Technically, you get an arithmetic overflow
  - To Fortran, it’s your fault: you are on your own
  - Best practice: be very careful of intermediate results

- $\gcd(a, b)$ is an exact divisor of $b$
Beware of Type Ranges

- \( a \times b / \text{gcd}(a, b) \) same as \( (a \times b) / \text{gcd}(a, b) \)

- What if the result of a calculation cannot be represented in the given type?
  - Technically, you get an arithmetic *overflow*
  - To Fortran, it’s your fault: you are on your own
  - Best practice: be very careful of intermediate results

- Easy fix: \( \text{gcd}(a, b) \) is an exact divisor of \( b \)
module number_theory
    implicit none
    contains
    function gcd(a, b) ! Greatest Common Divisor
        integer :: gcd
        integer, intent(in) :: a, b
        integer :: gb, t

        gcd = abs(a)
        gb = abs(b)

        if (a == 0) gcd = gb
        if (a == 0 .or. b == 0) return

        do
            t = mod(gcd,gb)
            gcd = gb
            if (t == 0) exit
            gb = t
        end do
    end function gcd

    function lcm(a, b) ! Least Common Multiple
        integer :: lcm
        integer, intent(in) :: a, b

        if (a == 0 .and. b == 0) then
            lcm = 0 ; return
        end if

        lcm = a*(b/gcd(a,b))
    end function lcm
end module number_theory
Beware of Type Ranges

- $a\times b / \gcd(a,b)$ same as $(a\times b) / \gcd(a,b)$

- What if the result of a calculation cannot be represented in the given type?
  - Technically, you get an arithmetic overflow
  - To Fortran, it’s your fault: you are on your own
  - Best practice: be very careful of intermediate results

- Easy fix: $\gcd(a, b)$ is an exact divisor of $b$

- Try and test it:
  - with pairs of small positive integers
  - on the following pairs: 15, 18; -15, 18; 15, -18; -15, -18; 0, 15; 15, 0; 0, 0
  - with the pair: 1000000, 1000000
Beware of Type Ranges

- \(a \times b / \text{gcd}(a, b)\) same as \((a \times b) / \text{gcd}(a, b)\)

- What if the result of a calculation cannot be represented in the given type?
  - Technically, you get an arithmetic overflow
  - To Fortran, it’s your fault: you are on your own
  - Best practice: be very careful of intermediate results

- Easy fix: \(\text{gcd}(a, b)\) is an exact divisor of \(b\)

- Try and test it:
  - with pairs of small positive integers
  - on the following pairs: 15, 18; -15, 18; 15, -18; -15, -18; 0, 15; 15, 0; 0, 0
  - with the pair: 1000000, 1000000
  - and let’s test also with: 1000000, 1000001
Outline

Introduction

Fortran Basics

More Fortran Basics

Integer Types and Iterating
  Play it Again, Please
  Testing and Fixing it
  Hitting Limits
  Wider Integer Types
  How Bad it Used to Be
  Wrapping it Up 3

More on Compiling and Linking

Homeworks
Wider Integer Types

- On most nowadays platforms:

  - `integers` have 32 bits and
    - `huge(0)` returns 2147483647
  - `range(0)` returns 9, i.e. you can store 10
    - `10` in an integer
  - but 64 bits wide integers can safely host 10
    - `10`
  - `selected_int_kind(n)`:
    - returns a kind type parameter corresponding to an internal representation capable to host the value 10
    - `n` or `-1` if none is wide enough
  - `integer` accepts an optional kind type parameter
    - `integer(kind=selected_int_kind(9)) :: di`
      - usually makes `di` a 32 bits wide variable
    - `integer(kind=selected_int_kind(18)) :: wi`
      - makes `wi` a 64 bits wide variable
    - `integer(selected_int_kind(18)) :: wi`
      - will also do
Wider Integer Types

- On most nowadays platforms:
  - integers have 32 bits and \texttt{huge(0)} returns 2147483647
Wider Integer Types

- On most nowadays platforms:
  - integers have 32 bits and `huge(0)` returns 2147483647
  - `range(0)` returns 9, i.e. you can store $10^9$ in an integer
Wider Integer Types

- On most nowadays platforms:
  - integers have 32 bits and `huge(0)` returns 2147483647
  - `range(0)` returns 9, i.e. you can store $10^9$ in an integer
  - but 64 bits wide integers can safely host $10^{18}$
Wider Integer Types

- On most nowadays platforms:
  - integers have 32 bits and `huge(0)` returns 2147483647
  - `range(0)` returns 9, i.e. you can store $10^9$ in an integer
  - but 64 bits wide integers can safely host $10^{18}$

- `selected_int_kind(n)`:

```plaintext
integer(kind=selected_int_kind(9)) :: di
usually makes di a 32 bits wide variable
integer(kind=selected_int_kind(18)) :: wi
makes wi a 64 bits wide variable
integer(selected_int_kind(18)) :: wi
will also do
```
Wider Integer Types

- On most nowadays platforms:
  - integers have 32 bits and `huge(0)` returns 2147483647
  - `range(0)` returns 9, i.e. you can store $10^9$ in an integer
  - but 64 bits wide integers can safely host $10^{18}$

- `selected_int_kind(n)`:
  - returns a *kind type parameter* corresponding to an internal representation capable to host the value $10^n$
On most nowadays platforms:

- integers have 32 bits and \texttt{huge(0)} returns 2147483647
- \texttt{range(0)} returns 9, i.e. you can store $10^9$ in an integer
- but 64 bits wide integers can safely host $10^{18}$

\texttt{selected\_int\_kind(n)}:

- returns a \textit{kind type parameter} corresponding to an internal representation capable to host the value $10^n$
- or $-1$ if none is wide enough
On most nowadays platforms:

- integers have 32 bits and `huge(0)` returns 2147483647
- `range(0)` returns 9, i.e. you can store $10^9$ in an integer
- but 64 bits wide integers can safely host $10^{18}$

`selected_int_kind(n)`:

- returns a *kind type parameter* corresponding to an internal representation capable to host the value $10^n$
- or $-1$ if none is wide enough

`integer` accepts an optional *kind type parameter*

- `integer(kind=selected_int_kind(9)) :: di` usually makes `di` a 32 bits wide variable
- `integer(kind=selected_int_kind(18)) :: wi` makes `wi` a 64 bits wide variable
- `integer(selected_int_kind(18)) :: wi` will also do
module number_theory
  implicit none
  contains
    function gcd9(a, b) ! Greatest Common Divisor
      integer(selected_int_kind(9)) :: gcd9
      integer(selected_int_kind(9)), intent(in) :: a, b
      integer(selected_int_kind(9)) :: gb, t

      gcd9 = abs(a)
      gb = abs(b)

      if (a == 0) gcd9 = gb
      if (a == 0 .or. b == 0) return

      do
        t = mod(gcd9,gb)
        gcd9 = gb
        if (t == 0) exit
        gb = t
      end do
    end function gcd9

    function lcm9(a, b) ! Least Common Multiple
      integer(selected_int_kind(9)) :: lcm9
      integer(selected_int_kind(9)), intent(in) :: a, b

      if (a == 0 .and. b == 0) then
        lcm9 = 0 ; return
      end if

      lcm9 = a*(b/gcd9(a,b))
    end function lcm9
end module number_theory
And let’s add support for a wider integer range
GCD & LCM: Let’s Add Headroom

! add right after: end function lcm9

function gcd18(a, b) ! Greatest Common Divisor
    integer(selected_int_kind(18)) :: gcd18
    integer(selected_int_kind(18)) :: gb, t

    gcd18 = abs(a)
    gb = abs(b)

    if (a == 0) gcd18 = gb
    if (a == 0 .or. b == 0) return

    do
        t = mod(gcd18, gb)
        gcd18 = gb
        if (t == 0) exit
        gb = t
    end do
end function gcd18

function lcm18(a, b) ! Least Common Multiple
    integer(selected_int_kind(18)) :: lcm18

    if (a == 0 .and. b == 0) then
        lcm18 = 0 ; return
    end if

    lcm18 = a*(b/gcd18(a,b))
end function lcm18
Being More General and Generic

- And let’s add support for a wider integer range

- Wait!
  - Now we have to remember to call the right function, depending on the integer kind
  - But this is not Fortran style: we didn’t have to change the call to intrinsic `abs()` , it’s name is generic
  - Can we do better?

- interface blocks come to rescue
- Beware: specific functions under a same generic interface must differ in type of at least one argument
- and module procedure spares us typing and inconsistencies
- and private allows us to hide implementation details

- Best practices for robustness:
  - write generic procedures, whenever possible
  - hide implementation details, whenever possible
And let’s add support for a wider integer range

Wait!

- Now we have to remember to call the right function, depending on the integer kind
- But this is not Fortran style: we didn’t have to change the call to intrinsic \texttt{abs()} , it’s name is generic
- Can we do better?

Yes, we can do better!
Being More General and Generic

- And let’s add support for a wider integer range

- Wait!
  - Now we have to remember to call the right function, depending on the integer kind
  - But this is not Fortran style: we didn’t have to change the call to intrinsic `abs()`, it’s name is generic
  - Can we do better?

- Yes, we can do better!
  - `interface` blocks come to rescue
module number_theory
    implicit none

    private gcd9, lcm9, gcd18, lcm18

    interface gcd
        module procedure gcd9, gcd18
    end interface

    interface lcm
        module procedure lcm9, lcm18
    end interface

contains

    function gcd9(a, b) ! Greatest Common Divisor
        ! code as before
    end function gcd9

    function lcm9(a,b) ! Least Common Multiple
        ! code as before
    end function lcm9

    function gcd18(a, b) ! Greatest Common Divisor
        ! code as before
    end function gcd18

    function lcm18(a,b) ! Least Common Multiple
        ! code as before
    end function lcm18

end module number_theory
And let’s add support for a wider integer range

Wait!

- Now we have to remember to call the right function, depending on the integer kind
- But this is not Fortran style: we didn’t have to change the call to intrinsic \texttt{abs()}, it’s name is generic
- Can we do better?

Yes, we can do better!

- \texttt{interface} blocks come to rescue
- Beware: specific functions under a same generic interface must differ in type of at least one argument
module number_theory
  implicit none

  private gcd9, lcm9, gcd18, lcm18

  interface gcd
    module procedure gcd9, gcd18
  end interface

  interface lcm
    module procedure lcm9, lcm18
  end interface

contains

  function gcd9(a, b) ! Greatest Common Divisor
    ! code as before
  end function gcd9

  function lcm9(a,b) ! Least Common Multiple
    ! code as before
  end function lcm9

  function gcd18(a, b) ! Greatest Common Divisor
    ! code as before
  end function gcd18

  function lcm18(a,b) ! Least Common Multiple
    ! code as before
  end function lcm18

end module number_theory
Being More General and Generic

➤ And let’s add support for a wider integer range

➤ Wait!
  ➤ Now we have to remember to call the right function, depending on the integer kind
  ➤ But this is not Fortran style: we didn’t have to change the call to intrinsic `abs()`; it’s name is generic
  ➤ Can we do better?

➤ Yes, we can do better!
  ➤ `interface` blocks come to rescue
  ➤ Beware: specific functions under a same generic interface must differ in type of at least one argument
  ➤ and `module procedure` spares us typing and inconsistencies
module number_theory
    implicit none

    private gcd9, lcm9, gcd18, lcm18

    interface gcd
        module procedure gcd9, gcd18
    end interface

    interface lcm
        module procedure lcm9, lcm18
    end interface

contains

function gcd9(a, b) ! Greatest Common Divisor
    ! code as before
end function gcd9

function lcm9(a, b) ! Least Common Multiple
    ! code as before
end function lcm9

function gcd18(a, b) ! Greatest Common Divisor
    ! code as before
end function gcd18

function lcm18(a, b) ! Least Common Multiple
    ! code as before
end function lcm18

end module number_theory
And let’s add support for a wider integer range

Wait!

- Now we have to remember to call the right function, depending on the integer kind
- But this is not Fortran style: we didn’t have to change the call to intrinsic \texttt{abs()} , it’s name is generic
- Can we do better?

Yes, we can do better!

- \texttt{interface} blocks come to rescue
- Beware: specific functions under a same generic interface must differ in type of at least one argument
- and \texttt{module procedure} spares us typing and inconsistencies
- and \texttt{private} allows us to hide implementation details
Being More General and Generic

- And let’s add support for a wider integer range

- Wait!
  - Now we have to remember to call the right function, depending on the integer kind
  - But this is not Fortran style: we didn’t have to change the call to intrinsic `abs()` , it’s name is generic
  - Can we do better?

- Yes, we can do better!
  - `interface` blocks come to rescue
  - Beware: specific functions under a same generic interface must differ in type of at least one argument
  - and `module procedure` spares us typing and inconsistencies
  - and `private` allows us to hide implementation details

- Best practices for robustness:
And let’s add support for a wider integer range

Wait!

Now we have to remember to call the right function, depending on the integer kind

But this is not Fortran style: we didn’t have to change the call to intrinsic `abs()`; it’s name is generic

Can we do better?

Yes, we can do better!

`interface` blocks come to rescue

Beware: specific functions under a same generic interface must differ in type of at least one argument

and `module procedure` spares us typing and inconsistencies

and `private` allows us to hide implementation details

Best practices for robustness:

write generic procedures, whenever possible
And let’s add support for a wider integer range

Wait!
- Now we have to remember to call the right function, depending on the integer kind
- But this is not Fortran style: we didn’t have to change the call to intrinsic \texttt{abs()}, it’s name is generic
- Can we do better?

Yes, we can do better!
- \texttt{interface} blocks come to rescue
- Beware: specific functions under a same generic interface must differ in type of at least one argument
- and \texttt{module procedure} spares us typing and inconsistencies
- and \texttt{private} allows us to hide implementation details

Best practices for robustness:
- write generic procedures, whenever possible
- hide implementation details, whenever possible
Outline

Introduction

Fortran Basics

More Fortran Basics

Integer Types and Iterating
  Play it Again, Please
  Testing and Fixing it
  Hitting Limits
  Wider Integer Types
  How Bad it Used to Be
  Wrapping it Up 3

More on Compiling and Linking

Homeworks
FUNCTION GCD18(A, B)
   INTEGER*8 GCD18, A, B
   INTEGER*8 GB, T

   GCD18 = A
   GB = B

1   T = MOD(GCD18, GB)
   GCD18 = GB
   IF (T .EQ. 0) GO TO 2
   GB = T
   GO TO 1

2   CONTINUE
END

FUNCTION LCM18(A, B)
   INTEGER*8 LCM18, A, B
   INTEGER*8 GCD18
   EXTERNAL GCD18

   LCM18 = A*B/GCD18(A,B)
END
A Limited Language with Many Dialects

- No structured endless loops
  - Labels and GO TOs where used instead
- **CONTINUE** was a no-op
  - Used to mark destination of jumps
  - No comment
- **INTEGER*8** was used to declare an 8 bytes integer variable
  - Absolutely non standard
  - As are **INTEGER*1, INTEGER*2, INTEGER*4, REAL*4, REAL*8, COMPLEX*8, COMPLEX*16**

- Many dialects
  - Many proprietary extensions used to be developed
  - And then copied among vendors for compatibility reasons
  - Many extensions were eventually standardized
  - But not all of them!
  - They still lurk around, and can be tempting: resist!
Outline

Introduction

Fortran Basics

More Fortran Basics

Integer Types and Iterating
  Play it Again, Please
  Testing and Fixing it
  Hitting Limits
  Wider Integer Types
  How Bad it Used to Be
  Wrapping it Up 3

More on Compiling and Linking

Homeworks
More Types and Flow Control

- There are many integer types
  - With implementation dependent ranges
  - Selectable by kind type parameters
  - Whose limits can be devised using `huge()` or `range()`
- Library functions have generic names, good for most types
- And you can write your own generic interfaces
- Behavior on integer overflow is implementation defined
  - Some control is possible using parentheses
- Blocks of statements can be iterated forever...
  - ... and `exit` gets off the roundabout
- Logical conditions can be combined using `.or.` and `.and.` operators
Best Practices

- Do not rely on type sizes, they are implementation dependent
- Do not leave a function from inside a loop
- Think of intermediate results in expressions: they can overflow or underflow
- Be consistent with Fortran approach
  - E.g. writing generic interfaces
  - Even if it costs more work
  - Even if it costs learning more Fortran
  - Once again, you can do it in steps
  - You’ll appreciate it in the future
- Hide implementation details as much as possible
  - You’ll never regret
- Resist the temptation of old Fortran or non standard extensions
  - Will pay back in the future
Introduction

Fortran Basics

More Fortran Basics

Integer Types and Iterating

More on Compiling and Linking

Homeworks
Compiler Errors and Warnings

- Compiler stops on errors (grammar violation, syntactic errors, ...)

- Wall option turns on commonly used warning on gfortran but not -Wimplicit-interface for example

- Something is an error if not in Fortran 95 standard

- Use -std=f95 to force reference standard
Compiler Errors and Warnings

- Compiler stops on errors (grammar violation, syntactic errors, ...)
- Goes on if you write non-sensical code complying with the rules!
Compiler Errors and Warnings

- Compiler stops on errors (grammar violation, syntactic errors, ...)
- Goes on if you write non-sensical code complying with the rules!
- Compiler may perform extra checks and report warnings
  - Very useful in early development phases
  - ... sometimes pedantic
  - Read them carefully anyway

- `Wall` option turns on commonly used warnings on `gfortran` but not `Wimplicit-interface` for example
- If given earlier ...

- Something is an error if not in Fortran 95 standard
- Use `-std=f95` to force reference standard
Compiler Errors and Warnings

- Compiler stops on errors (grammar violation, syntactic errors, ...)
- Goes on if you write non-sensical code complying with the rules!
- Compiler may perform extra checks and report warnings
  - Very useful in early development phases
  - ... sometimes pedantic
  - Read them carefully anyway
- `-Wall` option turns on commonly used warning on `gfortran` but not `-Wimplicit-interface` for example
Compiler Errors and Warnings

- Compiler stops on errors (grammar violation, syntactic errors, ...)
- Goes on if you write non-sensical code complying with the rules!
- Compiler may perform extra checks and report warnings
  - Very useful in early development phases
  - ... sometimes pedantic
  - Read them carefully anyway
- `–Wall` option turns on commonly used warning on `gfortran` but not `–Wimplicit-interface` for example
- If given earlier ...
Compiler Errors and Warnings

- Compiler stops on errors (grammar violation, syntactic errors, ...)
- Goes on if you write non-sensical code complying with the rules!
- Compiler may perform extra checks and report warnings
  - Very useful in early development phases
  - ... sometimes pedantic
  - Read them carefully anyway
- `-Wall` option turns on commonly used warning on `gfortran` but not `-Wimplicit-interface` for example
- If given earlier ...
- Something is an error if not in Fortran 95 standard
Compiler Errors and Warnings

- Compiler stops on errors (grammar violation, syntactic errors, ...)
- Goes on if you write non-sensical code complying with the rules!
- Compiler may perform extra checks and report warnings
  - Very useful in early development phases
  - ... sometimes pedantic
  - Read them carefully anyway
- `-Wall` option turns on commonly used warning on `gfortran` but not `-Wimplicit-interface` for example
- If given earlier ...
- Something is an error if not in Fortran 95 standard
  - Use `-std=f95` to force reference standard
Creating an executable from source files is in general a three phase process:

- **pre-processing:**
  - each source file is read by the pre-processor
    - substitute (`#define`) MACROs
    - insert code by `#include` statements
    - insert or delete code evaluating `#ifdef`, `#if` ...

- **compiling:**
  - each source file is translated into an object code file
    - an object code file is an organised collection of symbols, referring to variables and functions defined or used in the source file

- **linking:**
  - object files should be combined together to build a single executable program
  - every symbol should be resolved
    - symbols can be defined in your object files
    - or available in other object code (external libraries)
Compiling with GNU gfortran

▶ When you give the command:

```
user@cineca$> gfortran dsp.f90 dsp_test.f90
```

▶ It's like going through three steps

- Pre-processing

  ```
  user@cineca$> gfortran -E -cpp dsp.f90
  user@cineca$> gfortran -E -cpp dsp_test.f90
  ```

  The `-E -cpp` option tells gfortran to stop after pre-processing. Simply calls `cpp` (automatically invoked if the file extension is F90). Output sent to standard output.

- Compiling sources

  ```
  user@cineca$> gfortran -c dsp.f90
  user@cineca$> gfortran -c dsp_test.f90
  ```

  The `-c` option tells gfortran to only compile the source. An object file `.o` is produced from each source file.
Compiling with GNU gfortran

- When you give the command:
  ```bash
  user@cineca$> gfortran dsp.f90 dsp_test.f90
  ```

- It’s like going through three steps
  
  - Pre-processing
    ```bash
    user@cineca$> gfortran -E -cpp dsp.f90
    user@cineca$> gfortran -E -cpp dsp_test.f90
    ```
    The `-E -cpp` option tells `gfortran` to stop after pre-processing, which simply calls `cpp` (automatically invoked if the file extension is `F90`).
    Output sent to standard output

  - Compiling sources
    ```bash
    user@cineca$> gfortran -c dsp.f90
    user@cineca$> gfortran -c dsp_test.f90
    ```
    The `-c` option tells `gfortran` to only compile the source. An object file `.o` is produced from each source file.
Compiling with GNU gfortran

▶ When you give the command:

```bash
user@cineca$> gfortran dsp.f90 dsp_test.f90
```

▶ It’s like going through three steps

▶ Pre-processing

```bash
user@cineca$> gfortran -E -cpp dsp.f90
gfortran -E -cpp dsp_test.f90
```

▶ `-E -cpp` option, tells `gfortran` to stop after pre-process
▶ Simply calls `cpp` (automatically invoked if the file extension is `F90`)
▶ Output sent to standard output
Compiling with GNU gfortran

- When you give the command:
  ```bash
  user@cineca$> gfortran dsp.f90 dsp_test.f90
  ```

- It’s like going through three steps
  - Pre-processing
    ```bash
    user@cineca$> gfortran -E -cpp dsp.f90
    user@cineca$> gfortran -E -cpp dsp_test.f90
    ```
    - `-E -cpp` option, tells `gfortran` to stop after pre-process
    - Simply calls `cpp` (automatically invoked if the file extension is `F90`)
    - Output sent to standard output

- Compiling sources
  ```bash
  user@cineca$> gfortran -c dsp.f90
  user@cineca$> gfortran -c dsp_test.f90
  ```
  - `-c` option tells `gfortran` to only compile the source
  - An object file `.o` is produced from each source file
Linking with GNU gfortran

- Linking object files together

  ```
  user@cineca$> gfortran dsp.o dsp_test.o
  ```

- To resolve symbols defined in external libraries, specify:
  - which libraries to use (`-l` option)
  - in which directories they are (`-L` option)

- How to link the library `libdsp.a` in `/mypath`

  ```
  user@cineca$> gfortran file1.o file2.o -L/mypath -ldsp
  ```

- How to create and link the DSP library:

  ```
  user@cineca$> gfortran -c dsp.f90
  ar curv libdsp.a dsp.o
  ranlib libdsp.a
  gfortran test_dsp.f90 -L. -ldsp
  ```

  - `ar` create the archive `libdsp.a` containing `dsp.o`
  - `ranlib` generate index to archive

- To include file like `.mod`, specify
  - in which directories they are (`-I` option)
Write a program that reads an integer value \texttt{limit} and prints the first \texttt{limit} prime numbers

- Use the GCD function to identify those numbers
- After testing the basic version, handle negative \texttt{limit} values: print an error message and attempt to read the value again
Write a module containing a function that takes an integer \( n \) as input, and returns the \( n \)-th element of the Fibonacci series \( f_n \).

Hint:

\[
\begin{align*}
F_0 &= 0 \\
F_1 &= 1 \\
F_n &= F_{n-1} + F_{n-2}
\end{align*}
\]

Write a main program to test your function

Read \( n \) from standard input

Try with \( n=2, 10, 40, 46, 48, \ldots \)

What's the greatest \( n := \text{maxn} \), for which \( f_n \) is representable by a default integer? (\texttt{huge} can help to find it out)

Use this information to handle too large values of \( n \) in your function:

If \( n > \text{maxn} \) print an error message and return -1
Part II

A Fortran Survey 2

Outline

More Flow Control
Numerical Integration
Wrapping it Up 4

Fortran Intrinsic Types, Variables and Math

Arrays
The code in this section is meant for didactical purposes only.

It is deliberately naive: focus is on language aspects, not on precision or accuracy.

As a consequence, it is prone to numerical problems.
Outline

More Flow Control
Numerical Integration
Wrapping it Up 4

Fortran Intrinsic Types, Variables and Math

Arrays
Let’s use the trapezoidal rule to estimate $\int_{a}^{b} f(x) \, dx$

Dividing the interval $[a, b]$ into $n$ equal sized slices, it boils down to:

$$\int_{a}^{b} f(x) \, dx \approx \frac{b-a}{n} \left( \frac{1}{2} f(a) + \frac{1}{2} f(b) + \sum_{k=1}^{n-1} f \left( a + k \frac{b-a}{n} \right) \right)$$

And to make it more juicy, let’s make a succession of estimates, doubling $n$ each time, until the estimate seems stable.
module integrals
    implicit none
contains
    function trap_int(a,b,f,tol) ! recursive approximation of integral
        real :: trap_int ! by trapezoidal rule
        real, intent(in) :: a, b, tol ! integration interval and tolerance
        interface
            real function f(x) ! function to integrate
                real, intent(in) :: x
            end function f
        end interface
        integer, parameter :: maxsteps = 2**23
        integer :: steps, i
        real :: acc, dx, prev_estimate, estimate

        steps = 2
        prev_estimate = 0.0 ; estimate = huge(0.0)
        dx = (b - a)*0.5
        acc = (f(a) + f(b))*0.5

        conv: do while (abs(estimate - prev_estimate) > tol)
            prev_estimate = estimate
            do i=1, steps, 2 ! only contributions from new points
                acc = acc + f(a + i*dx)
            end do
            estimate = acc*dx
            steps = steps * 2
            if (steps > maxsteps) exit conv
            dx = dx * 0.5
        end do conv

        trap_int = estimate
    end function trap_int
end module
module integrals
  implicit none
contains
  function trap_int(a,b,f,tol) ! recursive approximation of integral
    real :: trap_int ! by trapezoidal rule
    real, intent(in) :: a, b, tol ! integration interval and tolerance
    interface
      real function f(x) ! function to integrate
        real, intent(in) :: x
      end function f
    end interface
    integer, parameter :: maxsteps = 2**23
    integer :: steps, i
    real :: acc, dx, prev_estimate, estimate

    steps = 2
    prev_estimate = 0.0 ; estimate = huge(0.0)
    dx = (b - a)*0.5
    acc = (f(a) + f(b))*0.5

    conv: do while (abs(estimate - prev_estimate) > tol)
      prev_estimate = estimate
      do i=1, steps, 2 ! only contributions from new points
        acc = acc + f(a + i*dx)
      end do
      estimate = acc*dx
      steps = steps*2
      if (steps > maxsteps) exit conv
      dx = dx*0.5
    end do conv

    trap_int = estimate
  end function trap_int
end module
Function Arguments

- Yes, a function can be passed as an argument to another function!

- Simply pass the name on call, like this:
  \[ g = \text{trap}_{\text{int}}(-\pi, \pi, \text{sinc}, 0.0001) \]

- And then the function can be called using the dummy argument name

- And this can be done for any procedure

- And allows for very generic code to be written
  - i.e. reuse the same routine to integrate different functions in the same program

- Integer and real values can be mixed in expressions
  - As well as values of same type but different kind
   - Which is: when two values of different type/kind meet each other at a binary operator, the one with smaller numeric range is converted to the other
module integrals
  implicit none
contains
  function trap_int(a,b,f,tol) ! recursive approximation of integral
    real :: trap_int ! by trapezoidal rule
    real, intent(in) :: a, b, tol ! integration interval and tolerance
    interface
      real function f(x) ! function to integrate
        real, intent(in) :: x
      end function f
    end interface

    integer, parameter :: maxsteps = 2**23
    integer :: steps, i
    real :: acc, dx, prev_estimate, estimate

    steps = 2
    prev_estimate = 0.0 ; estimate = huge(0.0)
    dx = (b - a)*0.5
    acc = (f(a) + f(b))*0.5

    conv: do while (abs(estimate - prev_estimate) > tol)
      prev_estimate = estimate
      do i=1, steps, 2 ! only contributions from new points
        acc = acc + f(a + i*dx)
      end do
      estimate = acc*dx
      steps = steps*2
      if (steps > maxsteps) exit conv
      dx = dx*0.5
    end do conv

    trap_int = estimate
  end function trap_int
end module
Yes, a function can be passed as an argument to another function!

Simply pass the name on call, like this:

\[ g = \text{trap\_int}(-\pi, \pi, \text{sinc}, 0.0001) \]

And then the function can be called using the dummy argument name

And this can be done for any procedure

And allows for very generic code to be written

- i.e. reuse the same routine to integrate different functions in the same program

Integer and real values can be mixed in expressions

- As well as values of same type but different kind
- And the right thing will be done
- Which is: when two values of different type/kind meet each other at a binary operator, the one with smaller numeric range is converted to the other
module integrals
  implicit none
contains
  function trap_int(a,b,f,tol) ! recursive approximation of integral
    real :: trap_int ! by trapezoidal rule
    real, intent(in) :: a, b, tol ! integration interval and tolerance
    interface
      real function f(x) ! function to integrate
        real, intent(in) :: x
      end function f
    end interface
    integer, parameter :: maxsteps = 2**23
    integer :: steps, i
    real :: acc, dx, prev_estimate, estimate
    steps = 2
    prev_estimate = 0.0 ; estimate = huge(0.0)
    dx = (b - a)*0.5
    acc = (f(a) + f(b))*0.5
    conv: do while (abs(estimate - prev_estimate) > tol)
      prev_estimate = estimate
      do i=1, steps, 2 ! only contributions from new points
        acc = acc + f(a + i*dx)
      end do
      estimate = acc*dx
      steps = steps*2
      if (steps > maxsteps) exit conv
      dx = dx*0.5
    end do conv
    trap_int = estimate
  end function trap_int
end module
Iterating with `do while` ...

```plaintext
do while (logical-condition)
    block of statements
end do
```

1. Evaluates `logical-condition`
2. If `logical-condition` is false, goes to 5
3. Executes the `block of statements`
4. Goes back to 1
5. Execution proceeds to the statement following `end do`
module integrals
  implicit none
contains
  function trap_int(a,b,f,tol) ! recursive approximation of integral
    real :: trap_int ! by trapezoidal rule
    real, intent(in) :: a, b, tol ! integration interval and tolerance
    interface
      real function f(x) ! function to integrate
        real, intent(in) :: x
      end function f
    end interface
    integer, parameter :: maxsteps = 2**23
    integer :: steps, i
    real :: acc, dx, prev_estimate, estimate

    steps = 2
    prev_estimate = 0.0 ; estimate = huge(0.0)
    dx = (b - a)*0.5
    acc = (f(a) + f(b))*0.5

    conv: do while (abs(estimate - prev_estimate) > tol)
      prev_estimate = estimate
      do i=1, steps, 2 ! only contributions from new points
        acc = acc + f(a + i*dx)
      end do
      estimate = acc*dx
      steps = steps*2
      if (steps > maxsteps) exit conv
      dx = dx*0.5
    end do conv

    trap_int = estimate
  end function trap_int
end module
Iterating with `do while ... end do`

- **do while** *(logical-condition)*
  - *block of statements*
  ```
do while (logical-condition)

   block of statements

end do
```  
  1. Evaluates *logical-condition*
  2. If *logical-condition* is false, goes to 5
  3. Executes the *block of statements*
  4. Goes back to 1
  5. Execution proceeds to the statement following *end do*

- **do** loops too can be given a name
  1. And it can be used on `exit` statements to make the flow more evident
  2. Particularly for nested loops
Iterating with `do while ... end do`

- `do while (logical-condition)`
  - block of statements
- `end do`

  1. Evaluates `logical-condition`
  2. If `logical-condition` is false, goes to 5
  3. Executes the `block of statements`
  4. Goes back to 1
  5. Execution proceeds to the statement following `end do`

- `do` loops too can be given a name

  1. And it can be used on `exit` statements to make the flow more evident
  2. Particularly for nested loops

- Best practices:

  1. use names to mark loops when they are long or belong to a deep nest
  2. NEVER, NEVER permit your code to loop forever for some inputs
module integrals
  implicit none
contains
  function trap_int(a,b,f,tol) ! recursive approximation of integral
    real :: trap_int ! by trapezoidal rule
    real, intent(in) :: a, b, tol ! integration interval and tolerance
    interface
      real function f(x) ! function to integrate
        real, intent(in) :: x
      end function f
    end interface
    integer, parameter :: maxsteps = 2**23
    integer :: steps, i
    real :: acc, dx, prev_estimate, estimate

    steps = 2
    prev_estimate = 0.0 ; estimate = huge(0.0)
    dx = (b - a)*0.5
    acc = (f(a) + f(b))*0.5

    conv: do while (abs(estimate - prev_estimate) > tol)
      prev_estimate = estimate
      do i=1, steps, 2 ! only contributions from new points
        acc = acc + f(a + i*dx)
      end do
      estimate = acc*dx
      steps = steps*2
      if (steps > maxsteps) exit conv
      dx = dx*0.5
    end do conv

    trap_int = estimate
  end function trap_int
end module
Iterating with Counted \texttt{do}

\begin{itemize}
\item \texttt{do \ var = init, limit [, step]}
\item block of statements
\item end \texttt{do}
\end{itemize}

1. Sets \texttt{step} to 1, if none was specified
2. Assign the \texttt{init} value to \texttt{var}
3. Evaluates \[ n_{\text{iter}} = \max\{0, \lfloor (\texttt{limit} - \texttt{init} + \texttt{step}) / \texttt{step} \rfloor \} \]
4. If \( n_{\text{iter}} \) is zero goes to 6
5. Executes \( n_{\text{iter}} \) times the block of statements, adding \texttt{step} to \texttt{var} at the end of each block of statements
6. Execution proceeds to the statement following \texttt{end do}

\texttt{var, init, limit, and step} should be integers

\texttt{Mandatory in Fortran 2003}

\texttt{Reals can be used up to Fortran 95, but a bad idea, for both performance and reliability issues}

\texttt{Less flexible than a do while but more efficient execution (exit works, anyway)}

\texttt{Best practice: do not give name to very tight loops}
Iterating with Counted do

\[
\text{do \: var = init, limit [, step]} \\
\text{\hspace{1cm} block of statements} \\
\text{end do}
\]

1. Sets \(\text{step}\) to 1, if none was specified
Iterating with Counted do

- **do** `var = init, limit [, step]`
  
  *block of statements*

  **end do**

1. Sets `step` to 1, if none was specified
2. Assign the `init` value to `var`
Iterating with Counted do

\[
\begin{align*}
\text{do } & \text{ var } = \text{ init, limit [}, \text{ step}] \\
& \text{ block of statements} \\
\text{end do}
\end{align*}
\]

1. Sets \( \text{ step} \) to 1, if none was specified
2. Assign the \( \text{ init} \) value to \( \text{ var} \)
3. Evaluates \( n_{\text{iter}} = \max\{0, [(\text{limit} - \text{init} + \text{step})/\text{step}]\} \)
Iterating with Counted do

\[
\begin{align*}
\textbf{do } \text{var} &= \textit{init}, \textit{limit} [, \textit{step}] \\
\quad &\text{block of statements} \\
\textbf{end do}
\end{align*}
\]

1. Sets \textit{step} to 1, if none was specified
2. Assign the \textit{init} value to \textit{var}
3. Evaluates 
\[
\textbf{n}_{\text{iter}} = \max\{0, \lfloor (\textit{limit} - \textit{init} + \textit{step})/\textit{step} \rfloor \}
\]
4. If \( n_{\text{iter}} \) is zero goes to 6
Iterating with Counted do

- do var = init, limit [, step] block of statements end do

1. Sets step to 1, if none was specified
2. Assign the init value to var
3. Evaluates \( n_{iter} = \max\{0, \lfloor (limit - init + step)/step \rfloor \} \)
4. If \( n_{iter} \) is zero goes to 6
5. Executes \( n_{iter} \) times the block of statements, adding step to var at the end of each block of statements
Iterating with Counted do

\[ \textbf{do } \textit{var} = \textit{init}, \textit{limit} [, \textit{step}] \]
\[ \text{block of statements} \]
\[ \textbf{end do} \]

1. Sets \textit{step} to 1, if none was specified
2. Assign the \textit{init} value to \textit{var}
3. Evaluates \( n_{\text{iter}} = \max\{0, [(\textit{limit} - \textit{init} + \textit{step})/\textit{step}] \} \)
4. If \( n_{\text{iter}} \) is zero goes to 6
5. Executes \( n_{\text{iter}} \) times the \textit{block of statements}, adding \textit{step} to \textit{var} at the end of each \textit{block of statements}
6. Execution proceeds to the statement following \textbf{end do}
Iterating with Counted \texttt{do}

\begin{itemize}
\item \texttt{do var = init, limit [, step]}
\item \textit{block of statements}
\item \texttt{end do}
\end{itemize}

\begin{enumerate}
\item Sets \textit{step} to 1, if none was specified
\item Assign the \textit{init} value to \texttt{var}
\item Evaluates $n_{iter} = \max\{0, \lfloor (limit - init + step)/step \rfloor\}$
\item If $n_{iter}$ is zero goes to 6
\item Executes $n_{iter}$ times the \textit{block of statements}, adding \textit{step} to \texttt{var} at the end of each \textit{block of statements}
\item Execution proceeds to the statement following \texttt{end do}
\end{enumerate}

\begin{itemize}
\item \texttt{var, init, limit, and step} should be integers
Iterating with Counted \texttt{do}

\begin{itemize}
  \item \texttt{do \ var = init, limit [, step]}

  \begin{itemize}
    \item \texttt{block of statements}
  \end{itemize}

\end{itemize}
end \texttt{do}

1. Sets \textit{step} to 1, if none was specified
2. Assigns the \textit{init} value to \texttt{var}
3. Evaluates $n_{iter} = \max\{0, \lfloor (\textit{limit} - \textit{init} + \textit{step})/\textit{step} \rfloor \}$
4. If $n_{iter}$ is zero goes to 6
5. Executes $n_{iter}$ times the \texttt{block of statements}, adding \textit{step} to \texttt{var} at the end of each \texttt{block of statements}
6. Execution proceeds to the statement following \texttt{end do}

\begin{itemize}
  \item \texttt{var}, \textit{init}, \textit{limit}, and \textit{step} should be integers

  \begin{itemize}
    \item Mandatory in Fortran 2003
  \end{itemize}
\end{itemize}
Iterating with Counted do

```do var = init, limit [, step]
   block of statements
end do```

- Sets `step` to 1, if none was specified
- Assign the `init` value to `var`
- Evaluates \( n_{iter} = \max\{0, \lfloor (limit - init + step)/step \rfloor \} \)
- If \( n_{iter} \) is zero goes to 6
- Executes \( n_{iter} \) times the block of statements, adding `step` to `var` at the end of each block of statements
- Execution proceeds to the statement following `end do`

- `var`, `init`, `limit`, and `step` should be integers
  - Mandatory in Fortran 2003
  - Reals can be used up to Fortran 95, but a bad idea, for both performance and reliability issues
Iterating with Counted \texttt{do}

\texttt{do \ var = init, limit [, step]}
  block of statements
end do

1. Sets \texttt{step} to 1, if none was specified
2. Assign the \texttt{init} value to \texttt{var}
3. Evaluates \( n_{\text{iter}} = \max\{0, \lfloor (\texttt{limit} - \texttt{init} + \texttt{step})/\texttt{step} \rfloor \} \)
4. If \( n_{\text{iter}} \) is zero goes to 6
5. Executes \( n_{\text{iter}} \) times the block of statements, adding \texttt{step} to \texttt{var} at the end of each block of statements
6. Execution proceeds to the statement following end \texttt{do}

\textbf{var, init, limit, and step should be integers}

\begin{itemize}
  \item Mandatory in Fortran 2003
  \item Reals can be used up to Fortran 95, but a bad idea, for both performance and reliability issues
\end{itemize}

\textbf{Less flexible than a do while but more efficient execution (exit works, anyway)}
Iterating with Counted do

- **do** var = init, limit [, step]
  
  block of statements

end do

1. Sets step to 1, if none was specified
2. Assign the init value to var
3. Evaluates $n_{iter} = \max\{0, \lfloor (limit - init + step)/step\rfloor\}$
4. If $n_{iter}$ is zero goes to 6
5. Executes $n_{iter}$ times the block of statements, adding step to var at the end of each block of statements
6. Execution proceeds to the statement following end do

- var, init, limit, and step should be integers
  - Mandatory in Fortran 2003
  - Reals can be used up to Fortran 95, but a bad idea, for both performance and reliability issues

- Less flexible than a do while but more efficient execution (exit works, anyway)

- Best practice: do not give name to very tight loops
module integrals
  implicit none
contains
  function trap_int(a,b,f,tol) ! recursive approximation of integral
    real :: trap_int
    real, intent(in) :: a, b, tol ! integration interval and tolerance
    interface
      real function f(x) ! function to integrate
        real, intent(in) :: x
      end function f
    end interface
    integer, parameter :: maxsteps = 2**23
    integer :: steps, i
    real :: acc, dx, prev_estimate, estimate

    steps = 2
    prev_estimate = 0.0 ; estimate = huge(0.0)
    dx = (b - a)*0.5
    acc = (f(a) + f(b))*0.5

    conv: do while (abs(estimate - prev_estimate) > tol)
      prev_estimate = estimate
      do i=1, steps, 2 ! only contributions from new points
        acc = acc + f(a + i*dx)
      end do
      estimate = acc*dx
      steps = steps*2
      if (steps > maxsteps) exit conv
      dx = dx*0.5
    end do conv

    trap_int = estimate
  end function trap_int
end module
Time to Put it at Work

- Write a program to exercise `trap_int()` on functions with known integrals
Time to Put it at Work

- Write a program to exercise `trap_int()` on functions with known integrals

- Then take care of what was left out
Time to Put it at Work

▶ Write a program to exercise \texttt{trap\_int()} on functions with known integrals

▶ Then take care of what was left out

▶ Hints:
Time to Put it at Work

- Write a program to exercise `trap_int()` on functions with known integrals
- Then take care of what was left out
- Hints:
  - `trap_int()` arguments are naively handled: wrong results could be produced
Time to Put it at Work

- Write a program to exercise `trap_int()` on functions with known integrals
- Then take care of what was left out
- Hints:
  - `trap_int()` arguments are naively handled: wrong results could be produced
  - Robustness has been almost totally overlooked (except for the safety `exit`
Time to Put it at Work

- Write a program to exercise `trap_int()` on functions with known integrals
- Then take care of what was left out
- Hints:
  - `trap_int()` arguments are naively handled: wrong results could be produced
  - Robustness has been almost totally overlooked (except for the safety `exit`)
  - What if some arguments take a `NaN` value?
Write a program to exercise `trap_int()` on functions with known integrals

Then take care of what was left out

Hints:

- `trap_int()` arguments are naively handled: wrong results could be produced
- Robustness has been almost totally overlooked (except for the safety `exit`)
- What if some arguments take a NaN value?
- What if some arguments take an Inf value?
Time to Put it at Work

- Write a program to exercise `trap_int()` on functions with known integrals
- Then take care of what was left out
- Hints:
  - `trap_int()` arguments are naively handled: wrong results could be produced
  - Robustness has been almost totally overlooked (except for the safety `exit`)
  - What if some arguments take a `NaN` value?
  - What if some arguments take an `Inf` value?
  - What if some arguments take a ... value?
Procedure arguments and mixed-mode expressions were already there
Procedure arguments and mixed-mode expressions were already there

Counted loops looked like this:

```fortran
  do 10, i=1,10,3
    write(*,*) i
  10    continue
```
Procedure arguments and mixed-mode expressions were already there

Counted loops looked like this:

```fortran
do 10, i=1,10,3
    write(*,*) i
10    continue
```

do while, exit, end do weren’t there...

... at least in the standard...

but are often found in codes, as dialect extensions.
Outline

More Flow Control
  Numerical Integration
  Wrapping it Up 4

Fortran Intrinsic Types, Variables and Math

Arrays
More flow control
  - Procedure arguments
  - do while
  - Counted do

Mixed-mode expressions

Name your loops
  - Particularly if long or nested
  - Particularly if you exit them
  - But don’t do it for short ones

Prevent any loop from running forever for some program inputs
More Flow Control

Fortran Intrinsic Types, Variables and Math
  Integer Types
  Floating Types
  Expressions
  Arithmetic Conversions
  More Intrinsic Types

Arrays
Computing == manipulating data and calculating results
  ▶ Data are manipulated using internal, binary formats
  ▶ Data are kept in memory locations and CPU registers

Fortran doesn’t make assumptions on internal data representations
  ▶ And tries to abstract
  ▶ Most CPU are similar but all have peculiarities
  ▶ Some details depend on the specific executing (a.k.a. target) hardware architecture and software implementation
  ▶ Fortran provides facilities to translate between internal formats and human readable ones

Fortran allows programmers to:
  ▶ think in terms of data types and named containers
  ▶ disregard details on actual memory locations and data movements
Fortran is a Strongly Typed Language

- Each literal constant has a type
  - Dictates internal representation of the data value
- Each variable has a type
  - Dictates content internal representation and amount of memory
  - Type must be specified in a declaration before use
  - Unless you are so naive to rely on implicit declaration
- Each expression has a type
  - And subexpressions have too
  - Depends on operators and their arguments
- Each function has a type
  - That is the type of the returned value
  - Specified in function interface
- Procedure arguments have types
  - i.e. type of arguments to be passed in calls
  - Specified in procedure interface
  - If the compiler doesn’t know the interface, it will blindly pass whatever you provide
More Flow Control

Fortran Intrinsic Types, Variables and Math
  Integer Types
  Floating Types
  Expressions
  Arithmetic Conversions
  More Intrinsic Types

Arrays
## Integer Types (as on most CPUs)

<table>
<thead>
<tr>
<th>Type</th>
<th>Sign</th>
<th>Usual huge ()</th>
<th>Usual Width (bits)</th>
<th>Usual Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer(selected_int_kind(2))</td>
<td>+/-</td>
<td>127</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>integer(selected_int_kind(5))</td>
<td>+/-</td>
<td>32767</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>integer(kind(0))</td>
<td>+/-</td>
<td>2147483647</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>integer(selected_int_kind(9))</td>
<td>+/-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer(selected_int_kind(18))</td>
<td>+/-</td>
<td>9223372036854775807</td>
<td>64</td>
<td>8</td>
</tr>
</tbody>
</table>

- `selected_int_kind(n)` returns the least type able to host $10^n$
- `selected_int_kind(n)` returns -1 if no suitable type is available
- New platform/compiler? Always check maximum headroom with `huge()` or `range()`
- As we said, on most platforms `kind()` returns the byte size, but it’s not standard
Integer Literal Constants

- Integer literal constants have kinds too

By default, `kind(0)`

Unless you specify it:

In a non-portable way:

```
-123456_8
```

Or in a portable way:

```
integer, parameter :: i8=selected_int_kind(18)
-123456_i8
```

Rule of thumb:

- Write the number as is, if it is in the default integer kind range.
- Otherwise, specify kind.

Remember:

- Do not write:
  
```
spokes = bycicles*2*36
```

- Integer, parameter :: SpokesPerWheel = 36

- Code will be more readable, and you'll be ready for easy changes.
Integer Literal Constants

- Integer literal constants have kinds too
- By default, \texttt{kind(0)}
Integer Literal Constants

- Integer literal constants have kinds too
- By default, `kind(0)`
- Unless you specify it
  - In a non portable way:
    - `-123456_8`
  - Or in a portable way:
    ```
    integer, parameter :: i8=selected_int_kind(18)
    -123456_i8
    ```
- Rule of thumb:
  - write the number as is, if it is in default integer kind range
  - otherwise, specify kind
- Remember:
  - do not write `spokes = bycicles*2*36`
  - `integer, parameter :: SpokesPerWheel = 36`
  - code will be more readable, and you'll be ready for easy changes
Integer Literal Constants

- Integer literal constants have kinds too
- By default, \texttt{kind(0)}
- Unless you specify it
  - In a non portable way:
    \[\texttt{-123456_8}\]
  - Or in a portable way:
    \[
    \text{integer, parameter :: i8=selected_int_kind(18)}
    \]
    \[
    \texttt{-123456_i8}
    \]
- Rule of thumb:
  - write the number as is, if it is in default integer kind range
  - otherwise, specify kind
Integer Literal Constants

- Integer literal constants have kinds too
- By default, `kind(0)`
- Unless you specify it
  - In a non portable way:
    ```
    -123456_8
    ```
  - Or in a portable way:
    ```
    integer, parameter :: i8=selected_int_kind(18)
    -123456_i8
    ```
- Rule of thumb:
  - write the number as is, if it is in default integer kind range
  - otherwise, specify kind
- Remember:
  - do not write `spokes = bycicles*2*36`
  - `integer, parameter :: SpokesPerWheel = 36`
  - code will be more readable, and you’ll be ready for easy changes
### Integer Math

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(i)</td>
<td></td>
</tr>
<tr>
<td>sign(i, j)</td>
<td></td>
</tr>
<tr>
<td>dim(i, j)</td>
<td>if i &gt; j returns i − j else returns 0</td>
</tr>
<tr>
<td>mod(i, j)</td>
<td>Remainder function i − int(i/j) × j</td>
</tr>
<tr>
<td>modulo(i, j)</td>
<td>Modulo function i − floor(i/j) × j</td>
</tr>
<tr>
<td>min(i, j[, ...])</td>
<td>min{i, j[, ...]}</td>
</tr>
<tr>
<td>max(i, j[, ...])</td>
<td>max{i, j[, ...]}</td>
</tr>
</tbody>
</table>

- Use like: $a = \text{abs}(b+i) + c$
- More functions are available to manipulate values
  - E.g. for bit manipulations on binary computers
  - We’ll not cover them in this course, you can learn more about if you need to
- They can be found under different names (e.g. `iabs()`)—these are relics from the past
Outline

More Flow Control

Fortran Intrinsic Types, Variables and Math
  Integer Types
  Floating Types
  Expressions
  Arithmetic Conversions
  More Intrinsic Types

Arrays
## Floating Types (as on most CPUs)

<table>
<thead>
<tr>
<th>Type</th>
<th>Usual huge ()</th>
<th>Usual Width (bits)</th>
<th>Usual Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>3.40282347e38</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>real(kind(0.0))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>real(selected_real_kind(6))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>double precision real(kind(0.0d0))</td>
<td>1.79769313486231573e308</td>
<td>64</td>
<td>8</td>
</tr>
<tr>
<td>real(selected_real_kind(15))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>real(selected_real_kind(18))</td>
<td>&gt; 1.2e4932</td>
<td>80 or 128</td>
<td>10 or 16</td>
</tr>
<tr>
<td>complex</td>
<td>NA</td>
<td>NA</td>
<td>8</td>
</tr>
<tr>
<td>complex(kind(0.0))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>complex(selected_real_kind(6))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>complex(kind(0.0d0))</td>
<td>NA</td>
<td>NA</td>
<td>16</td>
</tr>
<tr>
<td>complex(selected_real_kind(15))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>complex(selected_real_kind(18))</td>
<td>NA</td>
<td>NA</td>
<td>20 or 32</td>
</tr>
</tbody>
</table>

- In practice, always in IEEE Standard binary format, but not a Standard requirement
- `selected_real_kind()` gets number of significant decimal digits, plus a second optional argument for exponent range, returns negative result if no suitable type is available
- `tiny()` returns smallest positive value
- New platform/compiler? Always check maximum headroom with `huge()` or `range()`
**real** Literal Constants

- Need something to distinguish them from integers
  - Decimal notation: 1.0, −17., .125, 0.22
  - Exponential decimal notation: $2e19$ ($2 \times 10^{19}$), $−123.4e9$ ($−1.234 \times 10^{11}$), $.72e−6$ ($7.2 \times 10^{-7}$)

By default, `kind(0.0)`

Unless you specify it

For double precision only:

```
-1.23456d5
```

For all kinds:

```
integer, parameter :: r8=selected_real_kind(15)
-123456.0_r8
```

Remember:

Do not write `charge = protons*1.602176487E-19`

Real, parameter:: `UnitCharge=1.602176487E-19`

It will come handier when more precise measurements will be available.
real Literal Constants

▶ Need something to distinguish them from integers
  ▶ Decimal notation: 1.0, −17., .125, 0.22
  ▶ Exponential decimal notation: 2e19 \( (2 \times 10^{19}) \), −123.4e9 \( (−1.234 \times 10^{11}) \), .72e−6 \( (7.2 \times 10^{-7}) \)

▶ By default, \texttt{kind(0.0)}
Real Literal Constants

- Need something to distinguish them from integers
  - Decimal notation: 1.0, -17., .125, 0.22
  - Exponential decimal notation: 2e19 (2 × 10^{19}), -123.4e9 (-1.234 × 10^{11}), .72e-6 (7.2 × 10^{-7})

- By default, `kind(0.0)`

- Unless you specify it
  - For double precision only:
    - -1.23456d5
  - For all kinds:
    ```
    integer, parameter :: r8=selected_real_kind(15)
    -123456.0_r8
    ```
real Literal Constants

- Need something to distinguish them from integers
  - Decimal notation: 1.0, -17., .125, 0.22
  - Exponential decimal notation: 2e19 \((2 \times 10^{19})\), -123.4e9 \((-1.234 \times 10^{11})\), .72e-6 \((7.2 \times 10^{-7})\)

- By default, `kind(0.0)`
- Unless you specify it
  - For double precision only:
    - -1.23456d5
  - For all kinds:
    
    integer, parameter :: r8=selected_real_kind(15)
    -123456.0_r8

- Remember:
  - do not write `charge = protons*1.602176487E-19`
  - real, parameter :: UnitCharge=1.602176487E-19
  - it will come handier when more precise measurements will be available
### Real Math

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(x)</td>
<td></td>
</tr>
<tr>
<td>sign(x, y)</td>
<td>x if y ≥ 0, −</td>
</tr>
<tr>
<td>dim(x, y)</td>
<td>if x &gt; y returns x − y else returns 0</td>
</tr>
<tr>
<td>mod(x, y)</td>
<td>Remainder function x − int(x/y) × y</td>
</tr>
<tr>
<td>modulo(x, y)</td>
<td>Modulo function x − floor(x/y) × y</td>
</tr>
<tr>
<td>aint(x)^1, int(x)^1,2</td>
<td>if x &gt; 0 returns</td>
</tr>
<tr>
<td>anint(x)^2, nint(x)^1,2</td>
<td>nearest integer to x</td>
</tr>
<tr>
<td>floor(x)^1,2, ceiling(x)^1,2</td>
<td>⌊x⌋, ⌈x⌉</td>
</tr>
<tr>
<td>fraction(x)</td>
<td>fractional part of x</td>
</tr>
<tr>
<td>nearest(x, s)</td>
<td>next representable value to x,</td>
</tr>
<tr>
<td></td>
<td>in direction given by the sign of s</td>
</tr>
<tr>
<td>spacing(x)</td>
<td>absolute spacing of numbers near x</td>
</tr>
<tr>
<td>max(x, y[, ...])</td>
<td>max{x, y[, ...]}</td>
</tr>
<tr>
<td>min(x, y[, ...])</td>
<td>min{x, y[, ...]}</td>
</tr>
</tbody>
</table>

1. Result is of integer type
2. Accept an optional argument for kind type of the result

- They can be found under different names (e.g. `dabs()`): these are relics from the past
- More functions are available to manipulate values
  - Mostly in the spirit of IEEE Floating Point Standard
  - We’ll not cover them in this course, but encourage you to learn more about
### Functions

<table>
<thead>
<tr>
<th>Functions</th>
<th>Compute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sqrt(x)</code></td>
<td>$\sqrt{x}$</td>
</tr>
<tr>
<td><code>sin(x)</code>, <code>cos(x)</code>, <code>tan(x)</code>, <code>asin(x)</code>, <code>acos(x)</code>, <code>atan(x)</code></td>
<td>Trigonometric functions</td>
</tr>
<tr>
<td><code>atan2(x, y)</code></td>
<td>Arc tangent in $(-\pi, \pi]$</td>
</tr>
<tr>
<td><code>exp(x)</code>, <code>log(x)</code>, <code>log10(x)</code></td>
<td>$e^x$, $\log_e x$, $\log_{10} x$</td>
</tr>
<tr>
<td><code>sinh(x)</code>, <code>cosh(x)</code>, <code>tanh(x)</code></td>
<td>Hyperbolic functions</td>
</tr>
</tbody>
</table>

- Again, they can be found under different names (e.g. `dcos( )`): these are relics from the past
<table>
<thead>
<tr>
<th>Functions</th>
<th>Compute</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(z),</td>
<td>$</td>
</tr>
<tr>
<td>aimag(z)</td>
<td>imaginary part of $z$,</td>
</tr>
<tr>
<td>real(z)↑</td>
<td>real part of $z$</td>
</tr>
<tr>
<td>cmplx(x, y)↑</td>
<td>converts from real to complex</td>
</tr>
<tr>
<td>conj(z)</td>
<td>Complex conjugate of $z$</td>
</tr>
<tr>
<td>sqrt(z)</td>
<td>$\sqrt{z}$</td>
</tr>
<tr>
<td>sin(z), cos(z)</td>
<td>sine and cosine</td>
</tr>
<tr>
<td>exp(z),</td>
<td>$e^z$,</td>
</tr>
<tr>
<td>log(z)</td>
<td>$\log_e z$</td>
</tr>
</tbody>
</table>

1. Accept an optional argument for kind type of the result

- Once again, they can be found under different names (e.g. `cabs()`): again, these are relics from the past
The intrinsic function `precision(x)` for real or complex `x` returns the number of significant decimal digits.

Write a `module` which defines the `kind` constant for single, double and quadruple real precision.
The intrinsic function `precision (x)` for real or complex `x` returns the number of significant decimal digits.

Write a `module` which defines the `kind` constant for single, double and quadruple real precision.
The intrinsic function `precision(x)` for real or complex `x` returns the number of significant decimal digits.

Write a `module` which defines the `kind` constant for single, double and quadruple real precision.

To gain confidence: write a small program to print out `range` and `huge` values for these kinds.
The intrinsic function `precision (x)` for real or complex `x` returns the number of significant decimal digits.

Write a `module` which defines the `kind` constant for single, double and quadruple real precision.

To gain confidence: write a small program to print out `range` and `huge` values for these kinds.

Something going wrong?
Hands-on Session #2

- The intrinsic function `precision (x)` for real or complex `x` returns the number of significant decimal digits.
- Write a module which defines the `kind` constant for single, double and quadruple real precision.

- To gain confidence: write a small program to print out `range` and `huge` values for these kinds.
- Something going wrong?
- GNU Fortran compiler, up to release 4.5, lacks support for the quad-precision.
- If you are using Linux, load the most recent GNU compiler version and try again:
  
  ```fortran
  module load gnu
  ```
Let’s Be *Generic*

- Use the `real_kinds` module to rewrite `dsp` module functions to support both single and double precision
- And make all of them generic procedures
- Modify your test program to see exercise the new `dsp` module
More Flow Control

Fortran Intrinsic Types, Variables and Math
  Integer Types
  Floating Types
  **Expressions**
  Arithmetic Conversions
  More Intrinsic Types

Arrays
Arithmetic Expressions and Assignment

- Binary operators +, −, ∗ (multiplication) and / have the usual meaning and behavior
- And so do unary operators − and +
- Precedence
  - −a ∗ b + c/d same as ((−a) ∗ b) + (c/d)
  - −a + b same as (−a) + b
- Associativity of binary ones is from left to right
  - a + b + c same as (a + b) + c
  - a ∗ b/c ∗ d same as ((a ∗ b) / c) ∗ d
- Explicit ( and ) override precedence and associativity
- ** is the exponentiation operator
- Assignment: =
  - Assigns the value of expression on right hand side to a variable on the left hand side
  - Prior to first assignment, a variable content is undefined
Hitting Limits

- All types are limited in range
- What about:
  - `huge(0) + 1`? (too big)
  - `−huge(0.0) * 3.0`? (too negative)
- Technically speaking, this is an arithmetic overflow
- And division by zero is a problem too
- For integer types, the Standard says:
  - behavior and results are unpredictable
  - i.e. up to the implementation
- For real types, it also depends on the floating point environment
  - i.e. how behavior is configured for those cases
  - you could get `-huge(0.0)`, or a `NaN`, or `-Inf`
- Best practice: NEVER rely on behaviors observed with a specific architecture and/or compiler
Order of Subexpressions Evaluation

- Just imagine both functions $\text{foo}(x, y)$ and $\text{bar}(x, y)$ modify their actual arguments, or do I/O
Order of Subexpressions Evaluation

▶ Just imagine both functions $\text{foo}(x, y)$ and $\text{bar}(x, y)$ modify their actual arguments, or do I/O
  ▶ As you’ll remember, these are known as side effects
Order of Subexpressions Evaluation

- Just imagine both functions $foo(x, y)$ and $bar(x, y)$ modify their actual arguments, or do I/O
  - As you’ll remember, these are known as *side effects*

- Now imagine you meet code like this:

  ```
  t = foo(a, b) - bar(b, a)
  q = mod(foo(a, b), bar(a, b))
  ```
Order of Subexpressions Evaluation

- Just imagine both functions `foo(x, y)` and `bar(x, y)` modify their actual arguments, or do I/O
  - As you’ll remember, these are known as *side effects*

- Now imagine you meet code like this:
  
  ```
  t = foo(a, b) - bar(b, a)
  q = mod(foo(a, b), bar(a, b))
  ```

- Code like this is evil!
Order of Subexpressions Evaluation

- Just imagine both functions $\text{foo}(x, y)$ and $\text{bar}(x, y)$ modify their actual arguments, or do I/O
  - As you’ll remember, these are known as side effects

- Now imagine you meet code like this:
  
  $t = \text{foo}(a, b) - \text{bar}(b, a)$
  
  $q = \text{mod} (\text{foo}(a, b), \text{bar}(a, b))$

- Code like this is evil!

- Order of subexpressions evaluation is implementation dependent!
Order of Subexpressions Evaluation

- Just imagine both functions $\text{foo}(x, y)$ and $\text{bar}(x, y)$ modify their actual arguments, or do I/O
  - As you’ll remember, these are known as side effects

- Now imagine you meet code like this:
  
  $t = \text{foo}(a,b) - \text{bar}(b,a)$
  $q = \text{mod}(\text{foo}(a,b), \text{bar}(a,b))$

- Code like this is evil!

- Order of subexpressions evaluation is implementation dependent!
  - You don’t know in advance the order in which $\text{foo}()$ and $\text{bar}()$ are called
Order of Subexpressions Evaluation

- Just imagine both functions \( \text{foo}(x, y) \) and \( \text{bar}(x, y) \) modify their actual arguments, or do I/O
  - As you’ll remember, these are known as \textit{side effects}
- Now imagine you meet code like this:
  \[
  t = \text{foo}(a, b) - \text{bar}(b, a)
  \]
  \[
  q = \text{mod}(\text{foo}(a, b), \text{bar}(a, b))
  \]
- Code like this is evil!
- Order of subexpressions evaluation is implementation dependent!
  - You don’t know in advance the order in which \( \text{foo}() \) and \( \text{bar}() \) are called
  - Thus program behavior could differ among different implementations, or even among different compilations by the same compiler!
Order of Subexpressions Evaluation

- Just imagine both functions $\text{foo}(x, y)$ and $\text{bar}(x, y)$ modify their actual arguments, or do I/O
  - As you’ll remember, these are known as *side effects*

- Now imagine you meet code like this:
  
  $$t = \text{foo}(a, b) - \text{bar}(b, a)$$
  $$q = \text{mod}(\text{foo}(a, b), \text{bar}(a, b))$$

- Code like this is evil!

- Order of subexpressions evaluation is implementation dependent!
  - You don’t know in advance the order in which $\text{foo}()$ and $\text{bar}()$ are called
  - Thus program behavior could differ among different implementations, or even among different compilations by the same compiler!

- Ditto for order of evaluation of function arguments!
Order of Subexpressions Evaluation

▶ Just imagine both functions $\text{foo}(x,y)$ and $\text{bar}(x,y)$ modify their actual arguments, or do I/O
  ▶ As you’ll remember, these are known as side effects

▶ Now imagine you meet code like this:
  
  $t = \text{foo}(a,b) - \text{bar}(b,a)$
  $q = \text{mod}(\text{foo}(a,b), \text{bar}(a,b))$

▶ Code like this is evil!

▶ Order of subexpressions evaluation is implementation dependent!
  ▶ You don’t know in advance the order in which $\text{foo}()$ and $\text{bar}()$ are called
  ▶ Thus program behavior could differ among different implementations, or even among different compilations by the same compiler!

▶ Ditto for order of evaluation of function arguments!

▶ NEVER! NEVER write code that relies on order of evaluation of subexpressions, or actual arguments!
More Flow Control

Fortran Intrinsic Types, Variables and Math
  Integer Types
  Floating Types
  Expressions
  Arithmetic Conversions
  More Intrinsic Types

Arrays
Mixing Types in Expressions

- Fortran allows for expressions mixing any arithmetic types
  - A result will always be produced
  - Whether this is the result you expect, it’s another story

- Broadly speaking, the base concept is clear
  - For each binary operator in the expression, in order of precedence and associativity:
    - If both operands have the same type, fine
    - Otherwise, operand with narrower range is converted to type of other operand

- OK when mixing floating types
  - The wider range includes the narrower one
- OK when mixing integer types
  - The wider range includes the narrower one
Mixing Types in Expressions

- Fortran allows for expressions mixing any arithmetic types
  - A result will always be produced
  - Whether this is the result you expect, it’s another story
- Broadly speaking, the base concept is clear
Mixing Types in Expressions

- Fortran allows for expressions mixing any arithmetic types
  - A result will always be produced
  - Whether this is the result you expect, it’s another story
- Broadly speaking, the base concept is clear
- For each binary operator in the expression, in order of precedence and associativity:
  - if both operands have the same type, fine
  - otherwise, operand with narrower range is converted to type of other operand
Mixing Types in Expressions

- Fortran allows for expressions mixing any arithmetic types
  - A result will always be produced
  - Whether this is the result you expect, it’s another story

- Broadly speaking, the base concept is clear

- For each binary operator in the expression, in order of precedence and associativity:
  - if both operands have the same type, fine
  - otherwise, operand with narrower range is converted to type of other operand

- OK when mixing floating types
  - The wider range includes the narrower one
Mixing Types in Expressions

- Fortran allows for expressions mixing any arithmetic types
  - A result will always be produced
  - Whether this is the result you expect, it’s another story
- Broadly speaking, the base concept is clear
- For each binary operator in the expression, in order of precedence and associativity:
  - if both operands have the same type, fine
  - otherwise, operand with narrower range is converted to type of other operand
- OK when mixing floating types
  - The wider range includes the narrower one
- OK when mixing integer types
  - The wider range includes the narrower one
Type Conversion Traps

- For the assignment statement:
  - if variable and expression have the same type, fine
  - otherwise, right operand is converted to left operand type
  - if the value cannot be represented in the destination type, it’s an overflow, and you are on your own
Type Conversion Traps

- For the assignment statement:
  - if variable and expression have the same type, fine
  - otherwise, right operand is converted to left operand type
  - if the value cannot be represented in the destination type, it’s an overflow, and you are on your own

- We said: in order of precedence and associativity
Type Conversion Traps

For the assignment statement:
- if variable and expression have the same type, fine
- otherwise, right operand is converted to left operand type
- if the value cannot be represented in the destination type, it's an overflow, and you are on your own

We said: in order of precedence and associativity
- if $a$ is a 64 bits wide integer variable, and $b$ is a 32 bits wide integer variable and contains value $\text{huge}(0)$, in:
  \[ a = b \times 2 \]
  multiplication will overflow
Type Conversion Traps

- For the assignment statement:
  - if variable and expression have the same type, fine
  - otherwise, right operand is converted to left operand type
  - if the value cannot be represented in the destination type, it’s an overflow, and you are on your own

- We said: in order of precedence and associativity
  - if \( a \) is a 64 bits wide integer variable, and \( b \) is a 32 bits wide integer variable and contains value \( \text{huge}(0) \), in:
    \[
    a = b \times 2
    \]
    multiplication will overflow
  - and in (\( i8 \) as in a previous example):
    \[
    a = b \times 2 + 1_{i8}
    \]
    multiplication will overflow too
Type Conversion Traps

- For the assignment statement:
  - if variable and expression have the same type, fine
  - otherwise, right operand is converted to left operand type
  - if the value cannot be represented in the destination type, it’s an overflow, and you are on your own

- We said: in order of precedence and associativity
  - if \( a \) is a 64 bits wide integer variable, and \( b \) is a 32 bits wide integer variable and contains value \( \text{huge}(0) \), in:
    \[
    a = b \times 2
    \]
    multiplication will overflow
  - and in (\( \text{i8} \) as in a previous example):
    \[
    a = b \times 2 + 1_{\text{i8}}
    \]
    multiplication will overflow too
  - while:
    \[
    a = b \times 2_{\text{i8}} + 1
    \]
    is OK
Subtle Type Conversion Traps

- Think of mixing floating and integer types

Floating types have wider dynamic range than integer ones

- Not necessarily more precision

- A 32 bits real has fewer digits of precision than a 32 bits integer

- A 64 bits real has fewer digits of precision than a 64 bits integer

- The result of a conversion could actually be smaller than expected!
Subtle Type Conversion Traps

- Think of mixing floating and integer types
- Floating types have wider dynamic range than integer ones
Subtle Type Conversion Traps

- Think of mixing floating and integer types
- Floating types have wider dynamic range than integer ones
- But not necessarily more precision
Subtle Type Conversion Traps

- Think of mixing floating and integer types
- Floating types have wider dynamic range than integer ones
- But not necessarily more precision
- A 32 bits real has fewer digits of precision than a 32 bits integer
Subtle Type Conversion Traps

- Think of mixing floating and integer types
- Floating types have wider dynamic range than integer ones
- But not necessarily more precision
- A 32 bits real has fewer digits of precision than a 32 bits integer
- And a 64 bits real has fewer digits of precision than a 64 bits integer
Subtle Type Conversion Traps

- Think of mixing floating and integer types
- Floating types have wider dynamic range than integer ones
- But not necessarily more precision
- A 32 bits real has fewer digits of precision than a 32 bits integer
- And a 64 bits real has fewer digits of precision than a 64 bits integer
- The result of a conversion could actually be smaller than expected!
Get in Control!

▶ Do not blindly rely on implementation dependent chance!

▶ Use explicit type conversion functions:
  - `int(x[,kind])`
  - `real(x[,kind])`
  - `cmplx(x[,y][,kind])`

▶ They let you override standard conversion rules
▶ In previous example, you could use it like this:
  ```
  a = int(b,i8)*2 + 1
  ```

▶ Type conversion functions are not magic
▶ Only convert values, not type of variables you assign to
▶ Do not abuse them
▶ Make codes unreadable
▶ Could be evidence of design mistakes
▶ Or that your Fortran needs a refresh
Get in Control!

- Do not blindly rely on implementation dependent chance!
- Use explicit type conversion functions:
  - `int(x[, kind])`
  - `real(x[, kind])`
  - `cmplx(x[, y][, kind])`

They let you override standard conversion rules

Type conversion functions are not magic

Only convert values, not type of variables you assign to

Do not abuse them

Make codes unreadable

Could be evidence of design mistakes

Or that your Fortran needs a refresh
Get in Control!

- Do not blindly rely on implementation dependent chance!
- Use explicit type conversion functions:
  - `int(x[, kind])`
  - `real(x[, kind])`
  - `cmplx(x[, y][, kind])`
- They let you override standard conversion rules
  - In previous example, you could use it like this:
    
    ```fortran
    a = int(b,i8)*2 + 1
    ```
Get in Control!

- Do not blindly rely on implementation dependent chance!
- Use explicit type conversion functions:
  - `int(x[, kind])`
  - `real(x[, kind])`
  - `cmplx(x[, y][, kind])`
- They let you override standard conversion rules
  - In previous example, you could use it like this:
    ```
a = int(b,i8)*2 + 1
    ```
- Type conversion functions are not magic
  - Only convert values, not type of variables you assign to
Get in Control!

- Do not blindly rely on implementation dependent chance!
- Use explicit type conversion functions:
  - `int(x[, kind])`
  - `real(x[, kind])`
  - `cmplx(x[, y][, kind])`
- They let you override standard conversion rules
  - In previous example, you could use it like this:
    
    ```fortran
    a = int(b,i8)*2 + 1
    ```
- Type conversion functions are not magic
  - Only convert values, not type of variables you assign to
- Do not abuse them
  - Make codes unreadable
  - Could be evidence of design mistakes
  - Or that your Fortran needs a refresh
More Flow Control

Fortran Intrinsic Types, Variables and Math
  Integer Types
  Floating Types
  Expressions
  Arithmetic Conversions
  More Intrinsic Types

Arrays
Being logical

» A type good at reasoning
  » May have `.false.` or `.true.` value
  » Kind only affects size in memory
Being logical

- A type good at reasoning
  - May have `.false.` or `.true.` value
  - Kind only affects size in memory

- Arithmetic comparison operators return logical values
  - `==` (equal), `=/=` (not equal), `>`, `<`, `>=`, `<=`
  - or, in ancient Fortran, `.eq.`, `.ne.`, `.gt.`, `.lt.`, `.ge.`, `.le.`
Being logical

- A type good at reasoning
  - May have `.false.` or `.true.` value
  - Kind only affects size in memory

- Arithmetic comparison operators return logical values
  - `==` (equal), `!=` (not equal), `>`, `<`, `>=`, `<=`
  - or, in ancient Fortran, `.eq., .ne., .gt., .lt., .ge., .le.

- Logical expressions
  - `.not.` is unary NOT, `.and.` and `.or.` are binary AND and OR respectively, `.eqv.` is logical equivalence (.true. if operands both .true. or both .false.)
  - `.not. a .and. b .or. a .and. .not. b` means
    - `((.not.a).and.b).or.(a.and(.not.b))`
  - In doubt, add parentheses, but be sober
More Logic

- Logical friends from `ieee_arithmetic` module (simply use it)
  - `ieee_is_finite(x)`: .true. if argument value is finite
  - `ieee_is_nan(x)`: .true. if argument value is NaN
  - `ieee_unordered(x, y)`: .true. if at least one among x and y is NaN

As usual, order of subexpressions evaluation is implementation dependent.

But it's worse:
- if `test()` is a function returning a logical type value
- and `a` is .true.
- and `b` is .false.
- implementation is free (but not forced!) to not call `test()` at all in `a.or.test(x)` and `b.and.test(x)`

Again, do not rely on expressions side effects.
More Logic

- **Logical friends from `ieee_arithmetic` module (simply use it)**
  - `ieee_is_finite(x)`: `.true.` if argument value is finite
  - `ieee_is_nan(x)`: `.true.` if argument value is NaN
  - `ieee_unordered(x, y)`: `.true.` if at least one among `x` and `y` is NaN

- **As usual, order of subexpressions evaluation is implementation dependent**
More Logic

- Logical friends from `ieee_arithmetic` module (simply use it)
  - `ieee_is_finite(x)`: .true. if argument value is finite
  - `ieee_is_nan(x)`: .true. if argument value is NaN
  - `ieee_unordered(x, y)`: .true. if at least one among `x` and `y` is NaN

- As usual, order of subexpressions evaluation is implementation dependent

- But it’s worse:
  - if `test()` is a function returning a logical type value
  - and `a` is .true.
  - and `b` is .false.
  - implementation is free (but not forced!) to not call `test()` at all in `a.or.test(x)` and `b.and.test(x)`
  - Again, do not rely on expressions side effects
Fortran is not that good at manipulating text

But it has some `character`:

- `character :: c` defines a variable holding a single character, like ‘f’
- `character(len=80) :: s1, s2, s3` defines three variables holding strings of up to 80 characters, like ‘Fortran 2003’

There are character expressions, like:

- `s3(1:40) = s1(1:20)//s2(21:40)` which assigns to first half of `s3` the first quarter of `s1` and second quarter of `s2`

On assignment of a character expression to a longer variable, blank filling will take place.

On assignment of a character expression to a shorter variable, truncation will happen.
## String Manipulation

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(s)</code></td>
<td>string length</td>
</tr>
<tr>
<td><code>len_trim(s)</code></td>
<td>string length with trailing blanks ignored</td>
</tr>
<tr>
<td><code>trim(s)</code></td>
<td>string with trailing blanks removed</td>
</tr>
<tr>
<td><code>repeat(s, n)</code></td>
<td>string made of n copies of s</td>
</tr>
<tr>
<td><code>adjustl(s)</code></td>
<td>move leading blanks to trailing position</td>
</tr>
<tr>
<td><code>adjustr(s)</code></td>
<td>move trailing blanks to leading position</td>
</tr>
<tr>
<td><code>lge(s1,s2), lgt(s1,s2), lle(s1,s2), llt(s1,s2)</code></td>
<td>string comparisons</td>
</tr>
<tr>
<td><code>index(s,subs)</code></td>
<td>starting position of subs in s, 0 if not found</td>
</tr>
<tr>
<td><code>scan(s,set)</code></td>
<td>first position in s of a character matching set, 0 if none found</td>
</tr>
<tr>
<td><code>verify(s,set)</code></td>
<td>first position in s of a character not matching set, 0 if all match</td>
</tr>
<tr>
<td><code>achar(i)</code></td>
<td>character with ASCII code i</td>
</tr>
<tr>
<td><code>iachar(c)</code></td>
<td>ASCII code of character c</td>
</tr>
</tbody>
</table>

### Our advice:

- For most practical purposes, use I/O statements to manipulate strings as internal files (more on this later)
- If you are really serious about textual data, learn more
- Or switch to a different language
Outline

More Flow Control

Fortran Intrinsic Types, Variables and Math

Arrays
  Smoothing Signals
  A More Compact Notation
Outline

More Flow Control

Fortran Intrinsic Types, Variables and Math

Arrays
  Smoothing Signals
  A More Compact Notation
module smoothing
    implicit none
contains
    subroutine smooth(v, k)
        real, intent(inout) :: v(:)
        integer, intent(in) :: k
        integer :: n, l, i, j
        real :: work(size(v))

        n=size(v)
        l = 2*k + 1
        work = 0.0
        do i=1,n
            do j=i-k,i+k
                work(i) = work(i) + v(1+mod(n-1+j, n))
            enddo
        enddo
        v = work/l
    end subroutine smooth
end module smoothing

program test_smooth
    use smoothing
    implicit none
    integer, parameter :: n=10
    integer :: i, k
    real :: x(n)

    k = 2
    x = (/ (real(mod(i,n/2)), i=1,n) /)
    if ( k > n) stop 'More smoothing points than array elements'
    call smooth(x,k)
    write(*,*) x
end program test_smooth
module smoothing  
  implicit none  
contains  
  subroutine smooth(v, k)  
    real, intent(inout) :: v(:)  
    integer, intent(in) :: k  
    integer :: n, l, i, j  
    real :: work(size(v))  

    n=size(v)  
    l = 2*k +1  
    work = 0.0  
    do i=1,n  
      do j=i-k,i+k  
        work(i) = work(i) + v(1+mod(n-1+j, n))  
      enddo  
    enddo  
    v = work/l  
  end subroutine smooth  
end module smoothing

program test_smooth  
  use smoothing  
  implicit none  
  integer, parameter :: n=10  
  integer :: i, k  
  real :: x(n)  

  k = 2  
  x = (/ (real(mod(i,n/2)), i=1,n) /)  
  if ( k > n) stop 'More smoothing points than array elements'  
  call smooth(x,k)  
  write(*,*) x  
end program test_smooth
Subroutines

- Subroutines are procedures, like functions, except they do not return any value.
- They are invoked by:
  ```call subroutine-name(argument-list)```
- Like functions, they have *dummy* arguments that will be associated to *actual* arguments at call time.
- Unlike functions, they can not be used inside expressions.
- Their use is to be preferred to functions when:
  - actual arguments must be modified.
  - more than one result needs to be returned.
In Place Smoothing of a Periodic Signal

```fortran
module smoothing
  implicit none
contains
  subroutine smooth(v, k)
    real, intent(inout) :: v(:)
    integer, intent(in) :: k
    integer :: n, l, i, j
    real :: work(size(v))

    n=size(v)
    l = 2*k +1
    work = 0.0
    do i=1,n
      do j=i-k,i+k
        work(i) = work(i) + v(1+mod(n-1+j, n))
      enddo
    enddo
    v = work/l
  end subroutine smooth
end module smoothing

program test_smooth
  use smoothing
  implicit none
  integer, parameter :: n=10
  integer :: i, k
  real :: x(n)
  k = 2
  x = (/ (real(mod(i,n/2)), i=1,n) /)
  if ( k > n) stop 'More smoothing points than array elements'
  call smooth(x,k)
  write(*,*) x
end program test_smooth
```
Arrays

- `real :: x(n)`
  - Declares an array named `x`
  - A collection of variables of the same type (elements), laid out contiguously in memory
  - $i$-th element can be accessed with `x(i)`
  - `n` must be an integer expression whose value must be known at declaration time

- `x = (/.../)` is an array constructor
  - i.e. a sequence of values forming an array
  - Assigned to array in a single statement

  $(expression, index=initial, final)$ evaluates `expression` for each value of `index` as in a do-loop (hence is termed implied do-loop)
In Place Smoothing of a Periodic Signal

module smoothing
  implicit none
contains
  subroutine smooth(v, k)
    real, intent(inout) :: v(:)
    integer, intent(in) :: k
    integer :: n, l, i, j
    real :: work(size(v))

    n = size(v)
    l = 2*k +1
    work = 0.0
    do i=1,n
      do j=i-k,i+k
        work(i) = work(i) + v(mod(n-1+j, n))
      enddo
    enddo
    v = work/l
  end subroutine smooth
end module smoothing

program test_smooth
  use smoothing
  implicit none
  integer, parameter :: n=10
  integer :: i, k
  real :: x(n)

  k = 2
  x = (/ (real(mod(i,n/2)), i=1,n) /)
  if ( k > n) stop 'More smoothing points than array elements'
  call smooth(x,k)
  write(*,*) x
end program test_smooth
Arrays

- **real :: x(n)**
  - Declares an array named `x`
  - A collection of variables of the same type (elements), laid out contiguously in memory
  - `i`-th element can be accessed with `x(i)`
  - `n` must be an integer expression whose value must be known at declaration time

- What’s that `x = (/ . . . /)`?
  - `/ . . . /` is an array constructor
  - i.e. a sequence of values forming an array
  - Assigned to array in a single statement
  - `(expression, index=initial, final)` evaluates `expression` for each value of `index` as in a do-loop (hence is termed *implied do-loop*)
In Place Smoothing of a Periodic Signal

module smoothing
  implicit none
contains
  subroutine smooth(v, k)
    real, intent(inout) :: v(:)
    integer, intent(in) :: k
    integer :: n, l, i, j
    real :: work(size(v))

    n = size(v)
    l = 2*k + 1
    work = 0.0
    do i=1,n
      do j=i-k,i+k
        work(i) = work(i) + v(1+mod(n-1+j, n))
      enddo
    enddo
    v = work/l
  end subroutine smooth
end module smoothing

program test_smooth
  use smoothing
  implicit none
  integer, parameter :: n=10
  integer :: i, k
  real :: x(n)

  k = 2
  x = (/ (real(mod(i,n/2)), i=1,n) /)
  if (k > n) stop 'More smoothing points than array elements'
call smooth(x,k)
write(*,*) x
end program test_smooth
Subroutines and Arrays

- Arrays can be passed as arguments to procedures
- How can subroutine *smooth* know the size of the actual argument passed as `v`?
  - *real* :: `v(:)` states that size of `v` will be that of the actual argument
  - `v` is termed an *assumed-shape* array
  - This only works if the subroutine has explicit interface
- Otherwise, you can still use the good ol’ way:

  ```fortran
  subroutine smooth(v,k,n)
    integer n
    real v(n)
    ...
  end subroutine smooth
  ```
module smoothing
    implicit none
contains
    subroutine smooth(v, k)
        real, intent(inout) :: v(:)
        integer, intent(in) :: k
        integer :: n, l, i, j
        real :: work(size(v))

        n = size(v)
        l = 2*k + 1
        work = 0.0
        do i=1,n
            do j=i-k,i+k
                work(i) = work(i) + v(1 + mod(n - 1 + j, n))
            enddo
        enddo
        v = work/l
    end subroutine smooth
end module smoothing

program test_smooth
    use smoothing
    implicit none
    integer, parameter :: n=10
    integer :: i, k
    real :: x(n)

    k = 2
    x = (/ (real(mod(i,n/2)), i=1,n) /)
    if (k > n) stop 'More smoothing points than array elements'
    call smooth(x, k)
    write(*,*) x
end program test_smooth
Subroutines and Arrays

- Arrays can be passed as arguments to procedures.

- How can subroutine *smooth* know the size of the actual argument passed as \( v \)?
  - `real :: v(:)` states that size of \( v \) will be that of the actual argument.
  - \( v \) is termed an *assumed-shape* array.
  - This only works if the subroutine has explicit interface.

- Otherwise, you can still use the good ol’ way:
  ```
  subroutine smooth(v,k,n)
      integer n
      real v(n)
      ...
  end subroutine smooth
  ```

- How can subroutine *smooth* declare a local array matching in size the actual argument?
  - `size(v)` returns the number of elements (size) of \( v \).
  - `real :: work(size(v))` gives \( work \) same size as \( v \).
  - \( work \) is termed an *automatic object*.
WARNING: NO BOUNDS CHECKING!

- In Fortran, there is no bounds checking on array access
- And it is possible for something like this to happen

```fortran
real :: a(10)
...
do i=-100,100
   a(i) = i
end do
```
- If you are lucky, you’ll get a runtime error, otherwise you’ll corrupt surrounding memory areas, with really puzzling behavior
- Once upon a long ago, it used to be a ‘feature’:

```fortran
subroutine smooth(v,k,n)
   integer n
   real v(1)
   ...
```
WARNING: NO BOUNDS CHECKING!

- In Fortran, there is no bounds checking on array access
- And it is possible for something like this to happen
  
  ```fortran
  real :: a(10)
  ...
  do i=-100,100
   a(i) = i
  end do
  ```

- If you are lucky, you’ll get a runtime error, otherwise you’ll corrupt surrounding memory areas, with really puzzling behavior

- Once upon a long ago, it used to be a ‘feature’:
  
  ```fortran
  subroutine smooth(v,k,n)
    integer n
    real v(1)
    ...
  ```

- Use compiler options to enable runtime detection of out of bounds accesses
  - But execution is incredibly slowed down
  - Just a debugging tool, do not use it in production
The intrinsic subroutine `cpu_time()` is used to time code regions.

```fortran
real :: t1, t2
... 
call cpu_time(t1)
... ! code to be timed

! code to be timed

call cpu_time(t2)
write(*,*) 'Execution time for section 1: ', t2-t1, 'seconds'
```

Takes a default real argument and returns the processor time consumed by the program in seconds. Use it to measure execution time of the `test_smooth` program. Can we use less operations to get the same results (within round-off errors)?
The intrinsic subroutine _cpu_time(_ is used to time code regions

```fortran
real :: t1, t2
...
call cpu_time(t1)
    ! code to be timed
call cpu_time(t2)
write(*,*) 'Execution time for section 1: ', t2-t1, 'seconds'
```

- Takes a default real argument
The intrinsic subroutine \texttt{cpu\_time()} is used to time code regions

```fortran
real :: t1, t2

... call cpu_time(t1) ...
    ! code to be timed
... call cpu_time(t2)
write(*,*) 'Execution time for section 1: ', t2-t1, 'seconds'
```

- Takes a default real argument
- And returns in it processor time consumed by the program in seconds
The intrinsic subroutine `cpu_time()` is used to time code regions.

```fortran
real :: t1, t2
...
call cpu_time(t1)  
    ! code to be timed

call cpu_time(t2)
write(*,*) 'Execution time for section 1: ', t2-t1, 'seconds'
```

- Takes a default real argument
- And returns in it processor time consumed by the program in seconds
- Use it to measure execution time of `test_smooth` program

Can we use less operations to get the same results (within round-off errors)?
The intrinsic subroutine `cpu_time()` is used to time code regions

```fortran
real :: t1, t2
...
call cpu_time(t1)                  ! code to be timed
...                                 
call cpu_time(t2)
write(*,*) 'Execution time for section 1: ', t2-t1, ' seconds'
```

- Takes a default real argument
- And returns in it processor time consumed by the program in seconds

- Use it to measure execution time of `test_smooth` program
- Can we use less operations to get the same results (within round-off errors)?
Outline

More Flow Control

Fortran Intrinsic Types, Variables and Math

Arrays
  Smoothing Signals
  A More Compact Notation
Same Smoothing in a Different Idiom

module smoothing
    implicit none
    contains

    subroutine smoothinplace(v, k)
        implicit none
        real, intent(inout) :: v(:)
        integer, intent(in) :: k
        real :: work(-k+1:size(v)+k)
        integer :: i, j, l, n

        n = size(v)
        l = 2*k + 1
        work(1:n) = v
        work(-k+1:0) = v(n-k+1:n)
        work(n+1:n+k) = v(1:k)

        do j=1, k
            v = v + work(1-j:n-j) + work(1+j:n+j)
        end do
        v = v/l

    end subroutine smoothinplace

end module smoothing
module smoothing
    implicit none
contains

subroutine smoothinplace(v, k)
    implicit none
    real, intent(inout) :: v(:)
    integer, intent(in) :: k
    real :: work(-k+1:size(v)+k)
    integer :: i, j, l, n

    n = size(v)
    l = 2*k +1
    work(1:n) = v
    work(-k+1:0) = v(n-k+1:n)
    work(n+1:n+k) = v(1:k)

    do j=1, k
        v = v + work(1-j:n-j) + work(1+j:n+j)
    end do
    v = v/l

end subroutine smoothinplace

end module smoothing
By default, first element of a Fortran array has index 1

- $work(-k+1:size(v)+k)$
  - If first element index > last element index, the number of elements will be zero.
  - `lbound()` and `ubound()` functions help to check.

Our `work` array is larger than `v`, to accommodate copies of values needed to smooth the first and last `k` elements.

`work` is initialized in steps, each corresponding to a different section.

- `work(-k+1:0)` selects the first `k` elements.
- `work(1:n)` selects the successive `n` elements.
- `work(n+1:n+k)` selects...

Arrays and array sections are assigned to by `=`, in a natural manner (more on this later).
Array Slices

- By default, first element of a Fortran array has index 1
- But you can pick one to your taste, as in
  \( \text{work}(-k+1: \text{size}(v)+k) \)
By default, first element of a Fortran array has index 1

But you can pick one to your taste, as in
\[
\text{work}(-k+1: \text{size(v)}+k)
\]

- If \textit{first element index} \textgreater \textit{last element index} than the number of elements will be zero

An array section is a subset of the elements, and is itself an array

 work(-k+1:0) selects the first \(k\) elements

 work(1:n) selects the successive \(n\) elements

 work(n+1:n+k) selects...
Array Slices

- By default, first element of a Fortran array has index 1
- But you can pick one to your taste, as in:
  \[ \text{work}(-k+1: \text{size}(v)+k) \]
  - If \textit{first element index} > \textit{last element index} than the number of elements will be zero
  - \texttt{lbound()} and \texttt{ubound()} functions help to check
Array Slices

- By default, first element of a Fortran array has index 1
- But you can pick one to your taste, as in \textit{work}(-k+1:size(v)+k)
  - If \textit{first} element index > \textit{last} element index than the number of elements will be zero
  - \textit{lboun}d() and \textit{uboun}d() functions help to check
- Our \textit{work} array is larger than \textit{v}, to accommodate copies of values needed to smooth the first and last \textit{k} elements
Array Slices

- By default, first element of a Fortran array has index 1
- But you can pick one to your taste, as in `work(-k+1:size(v)+k)`
  - If `first element index > last element index` than the number of elements will be zero
  - `lbound()` and `ubound()` functions help to check
- Our `work` array is larger than `v`, to accommodate copies of values needed to smooth the first and last `k` elements
- `work` is initialized in steps, each corresponding to a different section
Array Slices

- By default, first element of a Fortran array has index 1
- But you can pick one to your taste, as in:
  \[ \text{work}(-k+1: \text{size(v)}+k) \]
  - If first element index > last element index than the number of elements will be zero
  - \text{lbound()} and \text{ubound()} functions help to check
- Our \text{work} array is larger than \text{v}, to accommodate copies of values needed to smooth the first and last \text{k} elements
- \text{work} is initialized in steps, each corresponding to a different section
  - An array section is a subset of the elements, and is itself an array
Array Slices

- By default, first element of a Fortran array has index 1
- But you can pick one to your taste, as in
  \[ \text{work}(-k+1:\text{size}(v)+k) \]
  - If first element index > last element index than the number of elements will be zero
  - \text{lbound()} and \text{ubound()} functions help to check
- Our \text{work} array is larger than \( v \), to accommodate copies of values needed to smooth the first and last \( k \) elements
- \text{work} is initialized in steps, each corresponding to a different section
  - An array section is a subset of the elements, and is itself an array
  - \text{work}(-k+1:0) selects the first \( k \) elements
  - \text{work}(1:n) selects the successive \( n \) elements
  - \text{work}(n+1:n+k) selects...
Array Slices

- By default, first element of a Fortran array has index 1
- But you can pick one to your taste, as in
  \[ \text{work}(-k+1: \text{size}(v)+k) \]
    - If \textit{first element index} \(>\) \textit{last element index} than the number of elements will be zero
    - \texttt{lbound()} and \texttt{ubound()} functions help to check
- Our \texttt{work} array is larger than \texttt{v}, to accommodate copies of values needed to smooth the first and last \texttt{k} elements
- \texttt{work} is initialized in steps, each corresponding to a different section
  - An array section is a subset of the elements, and is itself an array
    - \texttt{work}(-k+1:0) selects the first \texttt{k} elements
    - \texttt{work}(1:n) selects the successive \texttt{n} elements
    - \texttt{work}(n+1:n+k) selects...
- Arrays and array sections are assigned to by \texttt{=} in a natural manner (more on this later)
module smoothing
  implicit none
contains

subroutine smoothinplace(v, k)
  implicit none
  real, intent(inout) :: v(:)
  integer, intent(in) :: k
  real :: work(-k+1:size(v)+k)
  integer :: i, j, l, n

  n = size(v)
  l = 2*k + 1
  work(1:n) = v
  work(-k+1:0) = v(n-k+1:n)
  work(n+1:n+k) = v(1:k)

  do j = 1, k
    v = v + work(1-j:n-j) + work(1+j:n+j)
  end do
  v = v/l

end subroutine smoothinplace

end module smoothing
Array Expressions

- Arrays and array sections may be
  - referenced and used in expressions
  - passed as arguments to procedures

```fortran
  do j=1, k
    v = v + work(1-j:n-j) + work(1+j:n+j)
  end do
```

- Without array expressions, this code would look like:

```fortran
  do j=1, k
    do i=1, n
      v(i) = v(i) + work(i-j) + work(i+j)
    end do
  end do
```

- In an array expression, result must not depend in any way on the order of evaluation of elements

- You should think of array expressions as if all elements were computed at the same time
In Good Shape

- The size of a one-dimensional array is its **shape**
- Arithmetic operators act on arrays element by element
- Binary operators combine pairs of corresponding elements from the operands
- With binary operators and assignments, you must use **conformable**, i.e. identically shaped, arrays
- Except for scalar values (not variables!), that match any shape, as if they were replicated

```fortran
real, dimension(4) :: u, v, w
real :: t(1), s

! it’s right
s = t

! it’s wrong
w = s*u+v+2.3

! it’s right
w = (u-v)**2

! it’s OK
w = u+v(1:2)

! it’s wrong
w = u+v(1:2)
```

- By the way, **dimension** attribute lets you specify bounds and dimensions for a list of identical arrays
Intrinsic subroutine `random_number(x)` returns pseudo-random numbers uniformly distributed in \([0, 1)\) interval

- Takes an argument of type `real`, that can be either a scalar or an array
- Returns one random number if \(x\) is a scalar
- Returns an array of random numbers if \(x\) is an array

Is `random_number()` as uniform as advertised? Let’s check...
Let’s Build An Histogram

Write a program that:

1. reads an integer \texttt{niter} from standard input
2. generates \texttt{niter} random numbers in interval [0, 10)
3. builds an histogram and computes their average
4. Prints out results

To build the histogram:

1. Initialize to 0s an array \texttt{hist} of 20 integers to hold the bin count, then, at each iteration:
2. generate a random number
3. find out the bin it belongs to (i.e. its index in the array \texttt{hist})
4. intrinsic \texttt{ceiling(x)} function helps: it returns \lceil x \rceil
5. increment the corresponding array element and compute the percentages
6. accumulate the sum of the random numbers to compute the average value
Hands-on Session #5

- A prime number is a natural number which has only two distinct natural divisors: 1 and itself
- Find all primes less than or equal to a given $n$ by Eratosthenes’ algorithm:
  1. create a list of consecutive integers from 2 to $n$
  2. let be $p \leftarrow 2$ the first prime
  3. strike from the list all multiples of $p$ up to $n$
  4. let $p \leftarrow$ next number still in the list after $p$
  5. if $2p < n$, get back to step 3
  6. all remaining numbers in the list are primes

Try it now!
A prime number is a natural number which has only two distinct natural divisors: 1 and itself.

Find all primes less than or equal to a given $n$ by Eratosthenes’ algorithm:

1. create a list of consecutive integers from 2 to $n$
2. let be $p \leftarrow 2$ the first prime
3. strike from the list all multiples of $p$ up to $n$
4. let $p \leftarrow$ next number still in the list after $p$
5. if $2p < n$, get back to step 3
6. all remaining numbers in the list are primes

Try it now!

How could you spare iterations?

How could you spare memory?
Part III

Array Syntax and I/O

Array Syntax
  More dimensions
  Not a Panacea
  Arrays of Constants
  Elemental Procedures
  More Array Syntax

Input/Output
Outline

Array Syntax
More dimensions
Not a Panacea
Arrays of Constants
Elemental Procedures
More Array Syntax

Input/Output
function avgk(v, k)

    implicit none

    real, intent(in) :: v(:,:)
    integer, intent(in) :: k
    real :: avgk(size(v,1)/k,size(v,2)/k)

    integer :: i, j, n, m

    n = (size(v,1)/k)*k
    m = (size(v,2)/k)*k

    avgk = 0.0

    do j=1, k
        do i=1, k
            avgk = avgk + v(i:n:k,j:m:k)
        end do
    end do

    avgk = avgk/k**2

end function avgk
function avgk(v, k)

    implicit none

    real, intent(in) :: v(:, :)
    integer, intent(in) :: k
    real :: avgk(size(v,1)/k,size(v,2)/k)

    integer :: i, j, n, m

    n = (size(v,1)/k)*k
    m = (size(v,2)/k)*k

    avgk = 0.0

    do j=1, k
        do i=1, k
            avgk = avgk + v(i:n:k,j:m:k)
        end do
    end do

    avgk = avgk/k**2

end function avgk
Arrays may have up to 7 dimensions

Lower bounds default to 1, but you can specify them as for one-dimensional arrays, like in \( q(-k:k, 11:20) \)

Elements are referenced by a list of indices: \( v(1, 1) \)

The sequence of extents of an array is termed its *shape*, e.g. if \( a \) is real :: \( a(3, 2:5) \) then:

- \( \text{shape}(a) \) returns the array of extents \( /3, 4/ \)
- whereas \( \text{size}(a) \) returns 12

Multidimensional (i.e. rank>1) arrays and array sections may be involved in array expressions

As in the case of rank 1 arrays, they must be conformable when needed:
\[
\text{avgk}(1:3, :) = \text{avgk}(5:9, :) \quad \text{is wrong}
\]
Arrays and memory

- Some statements treat the elements of an array one by one in a special order, the *array element order*
  - obtained by counting most rapidly in the early dimensions
  - in the natural matrix representation this corresponds to storing the elements by column

- Most implementations actually store arrays in contiguous storage following the array element order
  - not required by the Standard, though
  - but crucial wrt performances, a typical optimization topic

- When dealing with complex data structures, the contiguity issue arises
  - Fortran 2008 adds the *contiguous* keyword to somehow address it
function avgk(v, k)

    implicit none

    real, intent(in) :: v(:,:)
    integer, intent(in) :: k
    real :: avgk(size(v,1)/k,size(v,2)/k)

    integer :: i, j, n, m

    n = (size(v,1)/k)*k
    m = (size(v,2)/k)*k

    avgk = 0.0

    do j=1, k
        do i=1, k
            avgk = avgk + v(i:n:k,j:m:k)
        end do
    end do

    avgk = avgk/k**2

end function avgk
Yes, a function may return an array
  And can be used in array expressions
  Its type is defined like any automatic object
  It must be assigned values inside the function
  No array-sections of the result can be selected on invocation
Yes, a function may return an array
  ▶ And can be used in array expressions
  ▶ Its type is defined like any automatic object
  ▶ It must be assigned values inside the function
  ▶ No array-sections of the result can be selected on invocation

An explicit interface is mandatory in the calling program
Yes, a function may return an array
  ▶ And can be used in array expressions
  ▶ Its type is defined like any automatic object
  ▶ It must be assigned values inside the function
  ▶ No array-sections of the result can be selected on invocation

An explicit interface is mandatory in the calling program

`size(array, dim)` returns the integer extent of `array` along dimension `dim`
function avgk(v, k)

    implicit none

    real, intent(in) :: v(:, :)
    integer, intent(in) :: k
    real :: avgk(size(v,1)/k, size(v,2)/k)

    integer :: i, j, n, m

    n = (size(v,1)/k)*k
    m = (size(v,2)/k)*k

    avgk = 0.0

    do j=1, k
        do i=1, k
            avgk = avgk + v(i:n:k, j:m:k)
        end do
    end do

    avgk = avgk/k**2

end function avgk
Array-Valued Functions

► Yes, a function may return an array
  ► And can be used in array expressions
  ► Its type is defined like any automatic object
  ► It must be assigned values inside the function
  ► No array-sections of the result can be selected on invocation

► An explicit interface is mandatory in the calling program

► \texttt{size(array, dim)} returns the integer extent of \texttt{array} along dimension \texttt{dim}

► Number of dimensions (a.k.a. rank) is mandatory in assumed shape arrays
function avgk(v, k)

    implicit none

    real, intent(in) :: v(:, :)
    integer, intent(in) :: k
    real :: avgk(size(v,1)/k, size(v,2)/k)

    integer :: i, j, n, m

    n = (size(v,1)/k)*k
    m = (size(v,2)/k)*k

    avgk = 0.0

    do j=1, k
        do i=1, k
            avgk = avgk + v(i:n:k, j:m:k)
        end do
    end do

    avgk = avgk/k**2

end function avgk
Pay Attention to Conformability

- Why are $n$ and $m$ computed that way?

- To prevent a problem:
  - what if $v$ extents aren't multiple of $k$?
  - $v(i:n:k,j:m:k)$ and $avgk$ would not be conformable
  - This cannot be checked at compile time, when shape of $v$ and value of $k$ are still unknown
  - Runtime checking is too costly for a performance oriented language
  - And out of bounds access could happen
  - Compile time detection of non conformable operands only works in a few cases
  - Again, use compiler options for runtime bounds checking
  - Again, very slow, only tolerable in debugging
Pay Attention to Conformability

- Why are \( n \) and \( m \) computed that way?
- To prevent a problem:
Pay Attention to Conformability

- Why are \( n \) and \( m \) computed that way?
- To prevent a problem:
- what if \( v \) extents aren’t multiple of \( k \)?
Pay Attention to Conformability

- Why are \( n \) and \( m \) computed that way?
- To prevent a problem:
- what if \( v \) extents aren’t multiple of \( k \)?
  - \( v(i:n:k, j:m:k) \) and \( \text{avg}k \) would not be conformable
Pay Attention to Conformability

- Why are \( n \) and \( m \) computed that way?
- To prevent a problem:
- what if \( v \) extents aren’t multiple of \( k \)?
  - \( v(i:n:k, j:m:k) \) and \( \text{avgk} \) would not be conformable
  - This cannot be checked at compile time, when shape of \( v \) and value of \( k \) are still unknown
Pay Attention to Conformability

- Why are \( n \) and \( m \) computed that way?
- To prevent a problem:
- what if \( v \) extents aren’t multiple of \( k \)?
  - \( v(i:n:k, j:m:k) \) and \( \text{avg}k \) would not be conformable
  - This cannot be checked at compile time, when shape of \( v \) and value of \( k \) are still unknown
  - Runtime checking is too costly for a performance oriented language

Again, use compiler options for runtime bounds checking

Again, very slow, only tolerable in debugging
Pay Attention to Conformability

- Why are $n$ and $m$ computed that way?
- To prevent a problem:
- what if $v$ extents aren’t multiple of $k$?
  - $v(i:n:k, j:m:k)$ and $avgk$ would not be conformable
  - This cannot be checked at compile time, when shape of $v$ and value of $k$ are still unknown
  - Runtime checking is too costly for a performance oriented language
  - And out of bounds access could happen
Pay Attention to Conformability

- Why are \( n \) and \( m \) computed that way?
- To prevent a problem:
  - what if \( v \) extents aren't multiple of \( k \)?
    - \( v(i:n:k, j:m:k) \) and \( \text{avg}_k \) would not be conformable
    - This cannot be checked at compile time, when shape of \( v \) and value of \( k \) are still unknown
    - Runtime checking is too costly for a performance oriented language
    - And out of bounds access could happen

- Compile time detection of non conformable operands only works in a few cases
Pay Attention to Conformability

- Why are \( n \) and \( m \) computed that way?
- To prevent a problem:
- what if \( v \) extents aren’t multiple of \( k \)?
  - \( v(i:n:k, j:m:k) \) and \( \text{avg}_k \) would not be conformable
  - This cannot be checked at compile time, when shape of \( v \) and value of \( k \) are still unknown
  - Runtime checking is too costly for a performance oriented language
  - And out of bounds access could happen

- Compile time detection of non conformable operands only works in a few cases
- Again, use compiler options for runtime bounds checking
Pay Attention to Conformability

- Why are \( n \) and \( m \) computed that way?
- To prevent a problem:
- what if \( v \) extents aren't multiple of \( k \)?
  - \( v(i:n:k, j:m:k) \) and \( \text{avgk} \) would not be conformable
  - This cannot be checked at compile time, when shape of \( v \) and value of \( k \) are still unknown
  - Runtime checking is too costly for a performance oriented language
  - And out of bounds access could happen
- Compile time detection of non conformable operands only works in a few cases
- Again, use compiler options for runtime bounds checking
- Again, very slow, only tolerable in debugging
Good ol’ style:

```plaintext
do i=1,n
    x(i) = b(i) / a(i,i)
    do j=i+1,n
        b(j) = b(j) - A(j,i)*x(i)
    enddo
endo
endo
```
Lower-Triangular Linear System

Good ol’ style:

```fortran
    do i=1,n
        x(i) = b(i) / a(i,i)
        do j=i+1,n
            b(j) = b(j) - A(j,i)*x(i)
        enddo
    enddo
```

In modern idiom:

```fortran
    do i=1,n
        x(i) = b(i) / a(i,i)
        b(i+1:n) = b(i+1:n) - A(i+1:n,i)*x(i)
    enddo
```
Lower-Triangular Linear System

- **Good ol’ style:**
  
  ```
  do i=1,n
    x(i) = b(i) / a(i,i)
    do j=i+1,n
      b(j) = b(j) - A(j,i)*x(i)
    enddo
  enddo
  ```

- **In modern idiom:**
  
  ```
  do i=1,n
    x(i) = b(i) / a(i,i)
    b(i+1:n) = b(i+1:n) - A(i+1:n,i)*x(i)
  enddo
  ```

- **What happens for i==n?**
Lower-Triangular Linear System

- **Good ol’ style:**
  ```fortran
do i=1,n
  x(i) = b(i) / a(i,i)
do j=i+1,n
  b(j) = b(j) - A(j,i)*x(i)
enddo
enddo
```

- **In modern idiom:**
  ```fortran
do i=1,n
  x(i) = b(i) / a(i,i)
  b(i+1:n) = b(i+1:n) - A(i+1:n,i)*x(i)
enddo
```

- **What happens for i==n?**
  - the array section \( b(n+1:n) \) has zero size:
    lower bound > upper bound
Lower-Triangular Linear System

- **Good ol’ style:**
  
  ```
  do i=1,n
    x(i) = b(i) / a(i,i)
    do j=i+1,n
      b(j) = b(j) - A(j,i)*x(i)
    enddo
  enddo
  ```

- **In modern idiom:**
  
  ```
  do i=1,n
    x(i) = b(i) / a(i,i)
    b(i+1:n) = b(i+1:n) - A(i+1:n,i)*x(i)
  enddo
  ```

- **What happens for i==n?**
  
  - the array section `b(n+1:n)` has zero size: lower bound > upper bound
  - No operation is performed
Picking Up Array Elements

- \( a(11:20) \) specifies all elements from index 11 to index 20
Picking Up Array Elements

- \( a(11:20) \) specifies all elements from index 11 to index 20
- \( a(11:20:2) \) specifies all odd index elements from index 11 to index 19
Picking Up Array Elements

- \(a(11:20)\) specifies all elements from index 11 to index 20
- \(a(11:20:2)\) specifies all odd index elements from index 11 to index 19
- \(a(19:10:-2)\) specifies the same elements, but in reverse order

Thus \(b = a(11:20)\) takes elements 11th to 20th of \(a\) and assigns them to \(b\).

And \(b = a(20:11:-1)\) does the same, but elements order is reversed.

Remember: \(b\) and the right hand side expression must be conformable.

Which in this case implies:

- \(\text{size(shape}(b))\) returns 1
- \(\text{size}(b)\) returns 10
Picking Up Array Elements

- \( a(11:20) \) specifies all elements from index 11 to index 20
- \( a(11:20:2) \) specifies all odd index elements from index 11 to index 19
- \( a(19:10:-2) \) specifies the same elements, but in reverse order
- Thus \( b = a(11:20) \) takes elements 11th to 20th of \( a \) and assigns them to \( b \)

\[ size(shape(b)) \] returns 1

\[ size(b) \] returns 10
Picking Up Array Elements

- \(a(11:20)\) specifies all elements from index 11 to index 20
- \(a(11:20:2)\) specifies all odd index elements from index 11 to index 19
- \(a(19:10:-2)\) specifies the same elements, but in reverse order
- Thus \(b = a(11:20)\) takes elements 11th to 20th of \(a\) and assigns them to \(b\)
- And \(b = a(20:11:-1)\) does the same, but elements order is reversed
Picking Up Array Elements

- \( a(11:20) \) specifies all elements from index 11 to index 20
- \( a(11:20:2) \) specifies all odd index elements from index 11 to index 19
- \( a(19:10:-2) \) specifies the same elements, but in reverse order
- Thus \( b = a(11:20) \) takes elements 11th to 20th of \( a \) and assigns them to \( b \)
- And \( b = a(20:11:-1) \) does the same, but elements order is reversed
- Remember: \( b \) and the right hand side expression must be conformable
Picking Up Array Elements

- \( a(11:20) \) specifies all elements from index 11 to index 20
- \( a(11:20:2) \) specifies all odd index elements from index 11 to index 19
- \( a(19:10:-2) \) specifies the same elements, but in reverse order
- Thus \( b = a(11:20) \) takes elements 11th to 20th of \( a \) and assigns them to \( b \)
- And \( b = a(20:11:-1) \) does the same, but elements order is reversed
- Remember: \( b \) and the right hand side expression must be conformable
- Which in this case implies:
Picking Up Array Elements

- \( a(11:20) \) specifies all elements from index 11 to index 20
- \( a(11:20:2) \) specifies all odd index elements from index 11 to index 19
- \( a(19:10:-2) \) specifies the same elements, but in reverse order
- Thus \( b = a(11:20) \) takes elements 11th to 20th of \( a \) and assigns them to \( b \)
- And \( b = a(20:11:-1) \) does the same, but elements order is reversed
- Remember: \( b \) and the right hand side expression must be conformable
- Which in this case implies:
  - \( \text{size}(\text{shape}(b)) \) returns 1
Picking Up Array Elements

- \( a(11:20) \) specifies all elements from index 11 to index 20
- \( a(11:20:2) \) specifies all odd index elements from index 11 to index 19
- \( a(19:10:-2) \) specifies the same elements, but in reverse order
- Thus \( b = a(11:20) \) takes elements 11th to 20th of \( a \) and assigns them to \( b \)
- And \( b = a(20:11:-1) \) does the same, but elements order is reversed
- Remember: \( b \) and the right hand side expression must be conformable
- Which in this case implies:
  - \( \text{size(shape}(b)) \) returns 1
  - and \( \text{size}(b) \) returns 10
In array assignment everything must happen ‘as if’ the r.h.s. expression is evaluated before assignment.

But difficult ones exist, like:

- $x(2:10) = x(1:9)$
  - In which $x(2)$ may not be assigned $x(1)$ value until the existing $x(2)$ value is assigned to $x(3)$, which itself...

A prudent (lazy?) compiler could add intermediate copies to temporary arrays:

- $x(10:2:-1) = x(9:1:-1)$

Array syntax can be very compact and elegant.

But temporary copies may impact performance, use your compiler options to spot them.
A Closer Look To Array Expressions

- In array assignment everything must happen ‘as if’ the r.h.s. expression is evaluated before assignment.
- To the benefit of performances, this is in many cases unnecessary.

- A prudent (lazy?) compiler could add intermediate copies to temporary arrays.
- \( x(2:10) = x(1:9) \) is more easily understood by some compilers.
- Array syntax can be very compact and elegant.
- Temporary copies may impact performance, use your compiler options to spot them.
A Closer Look To Array Expressions

- In array assignment everything must happen ‘as if’ the r.h.s. expression is evaluated before assignment
- To the benefit of performances, this is in many cases unnecessary
- But difficult ones exist, like \( x(2:10) = x(1:9) \)
- A prudent (lazy?) compiler could add intermediate copies to temporary arrays
- \( x(10:2:-1) = x(9:1:-1) \) is more easily understood by some compilers
- Array syntax can be very compact and elegant
- But temporary copies may impact performance, use your compiler options to spot them
A Closer Look To Array Expressions

- In array assignment everything must happen ‘as if’ the r.h.s. expression is evaluated before assignment.
- To the benefit of performances, this is in many cases unnecessary.
- But difficult ones exist, like \( x(2:10) = x(1:9) \)
- In which \( x(2) \) may not be assigned \( x(1) \) value until the existing \( x(2) \) value is assigned to \( x(3) \), which itself...
In array assignment everything must happen ‘as if’ the r.h.s. expression is evaluated before assignment.

To the benefit of performances, this is in many cases unnecessary.

But difficult ones exist, like \( x(2:10) = x(1:9) \)

In which \( x(2) \) may not be assigned \( x(1) \) value until the existing \( x(2) \) value is assigned to \( x(3) \), which itself...

A prudent (lazy?) compiler could add intermediate copies to temporary arrays.
A Closer Look To Array Expressions

- In array assignment everything must happen ‘as if’ the r.h.s. expression is evaluated before assignment.
- To the benefit of performances, this is in many cases unnecessary.
- But difficult ones exist, like \( x(2:10) = x(1:9) \).
- In which \( x(2) \) may not be assigned \( x(1) \) value until the existing \( x(2) \) value is assigned to \( x(3) \), which itself...
- A prudent (lazy?) compiler could add intermediate copies to temporary arrays.
- \( x(10:2:-1) = x(9:1:-1) \) is more easily understood by some compilers.
In array assignment everything must happen ‘as if’ the r.h.s. expression is evaluated before assignment.

To the benefit of performances, this is in many cases unnecessary.

But difficult ones exist, like $x(2:10) = x(1:9)$

In which $x(2)$ may not be assigned $x(1)$ value until the existing $x(2)$ value is assigned to $x(3)$, which itself...

A prudent (lazy?) compiler could add intermediate copies to temporary arrays.

$x(10:2:-1) = x(9:1:-1)$ is more easily understood by some compilers.

Array syntax can be very compact and elegant.
A Closer Look To Array Expressions

- In array assignment everything must happen ‘as if’ the r.h.s. expression is evaluated before assignment.
- To the benefit of performances, this is in many cases unnecessary.
- But difficult ones exist, like \( x(2:10) = x(1:9) \)
  - In which \( x(2) \) may not be assigned \( x(1) \) value until the existing \( x(2) \) value is assigned to \( x(3) \), which itself...
- A prudent (lazy?) compiler could add intermediate copies to temporary arrays.
  - \( x(10:2:-1) = x(9:1:-1) \) is more easily understood by some compilers.
- Array syntax can be very compact and elegant.
- But temporary copies may impact performance, use your compiler options to spot them.
Array Syntax
  More dimensions
  Not a Panacea
  Arrays of Constants
  Elemental Procedures
  More Array Syntax

Input/Output
function trace(matrix)
    implicit none
    real, intent(in) :: matrix(:,:)
    real :: trace
    integer :: i
    integer :: dim(2)

    dim = shape(matrix)
    trace = 0.0
    if (dim(1) /= dim(2)) return

    do i=1,dim(1)
        trace = trace + matrix(i,i)
    enddo
end function trace

- Not all operations on arrays can easily be expressed in array syntax
- Do you remember shape()? It returns an array whose elements are the extents of its argument
subroutine smooth(v, k)
  implicit none
  real, intent(inout) :: v(:)
  integer, intent(in) :: k
  integer :: n, l, i, j
  real :: work(size(v))

  n=size(v)
  l = 2*k +1
  work(1) = 0.0
  do j=1-k,1+k
    work(1) = work(1) + v(1+mod(n-1+j, n))
  enddo
  do i=2,n
    work(i)=work(i-1)+v(1+mod(n-1+i+k, n))-v(1+mod(n-2+i-k, n))
  enddo
  v = work/l
end subroutine smooth

The above code does the smoothing with minimal operations count

And cannot be expressed at all in array syntax

This is a quite common situation: optimal algorithms operating on arrays often sports dependencies in elements evaluations and updates
Array Syntax
   More dimensions
   Not a Panacea
   Arrays of Constants
   Elemental Procedures
   More Array Syntax

Input/Output
Tables of Coefficients

! Polynomial approximation of J0(x) for -3<=x<=3
! See Abramowitz&Stegun for details

function j0(x)
    implicit none
    real :: j0
    real, intent(in) :: x
    integer, parameter :: order = 6
    real, parameter, dimension(0:order) :: coeff = &
    (/ 1.0000000, &
     -2.2499997, &
     1.2656208, &
     -0.3163866, &
     0.0444479, &
     -0.0039444, &
     0.0002100 /)
    real :: xo3sq
    integer :: i

    xo3sq = (x/3.0)**2
    j0 = coeff(order)

    ! Horner method
    do i=order, 1, -1
        j0 = j0*xo3sq + coeff(i-1)
    end do
end function j0
parameter Arrays

- parameter arrays are very good at storing tables of:
  - polynomial coefficients
  - physical measurements
  - function values at discrete points

- In the past, data statements were used:
  
  ```
  data coeff /1.0,-2.2499997,1.2656208,-0.3163866, &
  0.0444479,-0.0039444,0.0002100/
  ```

- data statements:
  - are very versatile
  - very difficult to decipher
  - and tend to float away from variable declaration

- Use initialization instead
Array Syntax

More dimensions
Not a Panacea
Arrays of Constants
Elemental Procedures
More Array Syntax

Input/Output
program array_swap

    implicit none
    integer :: i, j
    real    :: a(0:10,10), b(11,10)

a=reshape( (/ (i*0.1, i=1,110) /), (/11,10/) )
b=reshape( (/ ((i*j+i, i=1,11), j=1,10) /), (/11,10/) )
call swap(a,b)

end program array_swap

subroutine swap(a,b)

    implicit none

    real, intent(inout) :: a(:,,:),b(:,:)
    real, dimension(size(a,1),size(a,2)) :: tmp

    tmp = a
    a = b
    b = tmp

end subroutine swap
program array_swap

  implicit none
  integer :: i, j
  real    :: a(0:10,10), b(11,10)

  a=reshape( (/ (i*0.1, i=1,110) /), (/11,10/) )
  b=reshape( (/ ((i*j+i, i=1,11), j=1,10) /), (/11,10/) )
  call swap(a,b)

end program array_swap

subroutine swap(a,b)

  implicit none

  real, intent(inout) :: a(:,,:),b(:,:)
  real, dimension(size(a,1),size(a,2)) :: tmp

  tmp = a
  a = b
  b = tmp

end subroutine swap
Assumed-shape arrays & Automatic objects

- The scope of the implied do loop indices \( i \) and \( j \) is the loop itself
  - Other variables with same names are unaffected

\[ \text{reshape(source, new_shape)} \]
returns an array with shape given by the rank one integer array \( \text{new_shape} \), and elements taken from \( \text{source} \) in array element order

Interface is as always mandatory for assumed shape arguments, so the compiler knows that additional information must be passed in to the function

But life can be simpler...
program array_swap

    implicit none
    integer :: i, j
    real :: a(0:10,10), b(11,10)

    a=reshape( (/ (i*0.1, i=1,110) /), (/11,10/) )
    b=reshape( (/ ((i*j+i, i=1,11), j=1,10) /), (/11,10/) )
    call swap(a,b)

end program array_swap

subroutine swap(a,b)

    implicit none

    real, intent(inout) :: a(:,:), b(:,:)
    real, dimension(size(a,1),size(a,2)) :: tmp

    tmp = a
    a = b
    b = tmp

end subroutine swap
Assumed-shape arrays & Automatic objects

- The scope of the implied do loop indices \( i \) and \( j \) is the loop itself
  - Other variables with same names are unaffected

- \texttt{reshape(source, new\_shape)} returns an array with shape given by the rank one integer array \texttt{new\_shape}, and elements taken from \texttt{source} in array element order
program array_swap

  implicit none
  integer :: i, j
  real :: a(0:10,10), b(11,10)

  interface
    subroutine swap(a,b)
      real, intent(inout) :: a(:,,:), b(:, :)
      real, dimension(size(a,1),size(a,2)) :: tmp
    end subroutine swap
  end interface

  a=reshape( (/ (i*0.1, i=1,110) /), (/11,10/) )
  b=reshape( (/ ((i*j+i, i=1,11), j=1,10) /), (/11,10/) )
  call swap(a,b)

end program array_swap

subroutine swap(a,b)

  implicit none

  real, intent(inout) :: a(:,,:), b(:, :)
  real, dimension(size(a,1),size(a,2)) :: tmp

  tmp = a
  a = b
  b = tmp

end subroutine swap
Assumed-shape arrays & Automatic objects

- The scope of the implied do loop indices $i$ and $j$ is the loop itself
  - Other variables with same names are unaffected

- `reshape(source, new_shape)` returns an array with shape given by the rank one integer array `new_shape`, and elements taken from `source` in array element order

- Interface is as always mandatory for assumed shape arguments, so the compiler knows that additional information must be passed in to the function
The scope of the implied do loop indices $i$ and $j$ is the loop itself

- Other variables with same names are unaffected

`reshape(source, new_shape)` returns an array with shape given by the rank one integer array `new_shape`, and elements taken from `source` in array element order

- Interface is as always mandatory for assumed shape arguments, so the compiler knows that additional information must be passed in to the function

- But life can be simpler...
Elemental Arrays Swap

```
program array_swap
  implicit none
  integer :: i, j
  real :: a(0:10,10), b(11,10)

  interface
    elemental subroutine swap(a,b)
      real, intent(inout) :: a, b
      real :: tmp
    end subroutine swap
  end interface

  a = reshape( (/ (i*0.1, i=1,110) /), (/11,10/) )
  b = reshape( (/ ((i*j+i, i=1,11), j=1,10) /), (/11,10/) )

  call swap(a,b)

end program array_swap

elemental subroutine swap(a,b)
  implicit none

  real, intent(inout) :: a, b
  real :: tmp

  tmp = a
  a = b
  b = tmp

end subroutine swap
```
Elemental Procedures

- Elemental procedures are applied element-wise to arrays (like most intrinsic arithmetic operators and mathematical functions)
Elemental Procedures

- Elemental procedures are applied element-wise to arrays (like most intrinsic arithmetic operators and mathematical functions)
- To define one, it has to be pure
  - If a function, it shall not have side effects of sort (not even \texttt{stop}!)
  - If a subroutine, side effects shall be restricted to \texttt{intent(out)} and \texttt{intent(inout)} arguments
  - Of course, a procedure that appears to be pure, but calls a non pure procedure, is not pure at all!
  - And some more constraints ensure the different procedure calls can be safely executed in any order
Elemental procedures are applied element-wise to arrays (like most intrinsic arithmetic operators and mathematical functions)

To define one, it has to be pure

- If a function, it shall not have side effects of sort (not even `stop`!)
- If a subroutine, side effects shall be restricted to `intent(out)` and `intent(inout)` arguments
- Of course, a procedure that appears to be pure, but calls a non pure procedure, is not pure at all!
- And some more constraints ensure the different procedure calls can be safely executed in any order

An explicit interface is mandatory

- It must specify the procedure as `elemental`
- It must specify `intent()` attribute for all arguments
Outline

Array Syntax
  More dimensions
  Not a Panacea
  Arrays of Constants
  Elemental Procedures
  More Array Syntax

Input/Output
Masks and where

- Logical array expressions like \( a(:) > 0.0 \) are often termed "masks"
Masks and **where**

- Logical array expressions like `a(:)>0.0` are often termed **masks**
- They come useful to restrict computations to specific array elements, as in the **where** statement:
  ```plaintext
  where (abs(a) > abs(b)) a = b
  ```
  the elemental assignment is evaluated only on elements satisfying the condition
Masks and `where`

- Logical array expressions like `a(:) > 0.0` are often termed `masks`.

- They come useful to restrict computations to specific array elements, as in the `where` statement:
  ```
  where (abs(a) > abs(b)) a = b
  the elemental assignment is evaluated only on elements satisfying the condition.
  ```

- The general form is the `where` construct:
  ```
  where (abs(a) > abs(b))
    c=b
  elsewhere
    c=a
  end where
  ```
Masks and \textit{where}

- Logical array expressions like \texttt{a(:)>0.0} are often termed \textit{masks}.
- They come useful to restrict computations to specific array elements, as in the \texttt{where} statement:
  \begin{verbatim}
  where (abs(a) > abs(b)) a = b
  \end{verbatim}
  the elemental assignment is evaluated only on elements satisfying the condition.

- The general form is the \texttt{where} construct:
  \begin{verbatim}
  where (abs(a) > abs(b))
    c = b
  elsewhere
    c = a
  end where
  \end{verbatim}

- Pay attention if you use non elemental functions in a \texttt{where}, you could be in for a surprise!
Masks and \texttt{where}

- Logical array expressions like $a(:)>0.0$ are often termed \textit{masks}.
- They come useful to restrict computations to specific array elements, as in the \texttt{where} statement:
  \begin{verbatim}
  where (abs(a) > abs(b)) a = b
  \end{verbatim}
The elemental assignment is evaluated only on elements satisfying the condition.

- The general form is the \texttt{where} construct:
  \begin{verbatim}
  where (abs(a) > abs(b))
      c=b
  elsewhere
      c=a
  end where
  \end{verbatim}

- Pay attention if you use non elemental functions in a \texttt{where}, you could be in for a surprise!
- \texttt{where} constructs can be nested and given a name.
forall allows writing array assignments which cannot be expressed with array expressions:

```plaintext
forall(i = 1:n) a(i,i) = x(i)**2
```

forall also accepts masks:

```plaintext
forall(i = 1:n, j = 1:n, y(i,j)/=0.) x(j,i) = 1.0/y(i,j)
```

In its construct form, it looks like:

```plaintext
forall(i = 2:n-1, j = 2:n-1)
a(i,j) = a(i,j-1) + a(i,j+1) + a(i-1,j) + a(i+1,j)
b(i,j) = a(i,j)
end forall
```

It works like array assignments:

Unlike do, there is no ordering of iterations, and changes appear as they were deferred.

Thus, no conflicts between reads and writes to a.

Assignment to b(i,j) takes place after that to a(i,j).

Referenced procedures must be pure.

forall constructs can be nested and given a name.
Say it With **foralls**

- **forall** allows writing array assignments which cannot be expressed with array expressions:
  ```latex
defaul(i = 1:n) a(i,i) = x(i)**2
  ```

- **forall** also accepts masks:
  ```latex
  forall(i = 1:n, j = 1:n, y(i,j)/=0.) x(j,i) = 1.0/y(i,j)
  ```

- Referenced procedures must be pure
- **forall** constructs can be nested and given a name
forall allows writing array assignments which cannot be expressed with array expressions:
```
forall(i = 1:n) a(i,i) = x(i)**2
```

forall also accepts masks:
```
forall(i = 1:n, j = 1:n, y(i,j)/=0.) x(j,i) = 1.0/y(i,j)
```

In its construct form, it looks like:
```
forall(i = 2:n-1, j = 2:n-1)
a(i,j) = a(i,j-1) + a(i,j+1) + a(i-1,j) + a(i+1,j)
b(i,j) = a(i,j)
end forall
```

It works like array assignments:
Say it With **forall**s

- **forall** allows writing array assignments which cannot be expressed with array expressions:
  
  ```
  forall(i = 1:n) a(i,i) = x(i)**2
  ```

- **forall** also accepts masks:
  
  ```
  forall(i = 1:n, j = 1:n, y(i,j)/=0.) x(j,i) = 1.0/y(i,j)
  ```

- In its construct form, it looks like:
  
  ```
  forall(i = 2:n-1, j = 2:n-1)
  a(i,j) = a(i,j-1) + a(i,j+1) + a(i-1,j) + a(i+1,j)
  b(i,j) = a(i,j)
  end forall
  ```

  It works like array assignments:

  - Unlike **do**, there is no ordering of iterations, and changes appear as they were deferred
Say it With **forall**

- **forall** allows writing array assignments which cannot be expressed with array expressions:
  
  ```
  forall(i = 1:n) a(i,i) = x(i)**2
  ```

- **forall** also accepts masks:
  
  ```
  forall(i = 1:n, j = 1:n, y(i,j)/=0.) x(j,i) = 1.0/y(i,j)
  ```

- In its construct form, it looks like:
  
  ```
  forall(i = 2:n-1, j = 2:n-1)
    a(i,j) = a(i,j-1) + a(i,j+1) + a(i-1,j) + a(i+1,j)
    b(i,j) = a(i,j)
  end forall
  ```

  It works like array assignments:
  
  - Unlike **do**, there is no ordering of iterations, and changes appear as they were deferred
  - Thus, no conflicts between reads and writes to **a**
forall allows writing array assignments which cannot be expressed with array expressions:
forall(i = 1:n) a(i,i) = x(i)**2

forall also accepts masks:
forall(i = 1:n, j = 1:n, y(i,j)/=0.) x(j,i) = 1.0/y(i,j)

In its construct form, it looks like:
forall(i = 2:n-1, j = 2:n-1)
a(i,j) = a(i,j-1) + a(i,j+1) + a(i-1,j) + a(i+1,j)
b(i,j) = a(i,j)
end forall

It works like array assignments:

- Unlike do, there is no ordering of iterations, and changes appear as they were deferred
- Thus, no conflicts between reads and writes to a
- Assignment to b(i, j) takes place after that to a(i, j)
Say it With `forall`s

- `forall` allows writing array assignments which cannot be expressed with array expressions:
  ```latex
  forall(i = 1:n) a(i,i) = x(i)**2
  ```

- `forall` also accepts masks:
  ```latex
  forall(i = 1:n, j = 1:n, y(i,j)!=0.) x(j,i) = 1.0/y(i,j)
  ```

- In its construct form, it looks like:
  ```latex
  forall(i = 2:n-1, j = 2:n-1)
  a(i,j) = a(i,j-1) + a(i,j+1) + a(i-1,j) + a(i+1,j)
  b(i,j) = a(i,j)
  end forall
  ```

  It works like array assignments:
  - Unlike `do`, there is no ordering of iterations, and changes appear as they were deferred.
  - Thus, no conflicts between reads and writes to `a`.
  - Assignment to `b(i,j)` takes place after that to `a(i,j)`.

- Referenced procedures must be pure.
Say it With **forall**s

- **forall** allows writing array assignments which cannot be expressed with array expressions:
  \[
  \text{forall}(i = 1:n) \ a(i,i) = x(i)^2
  \]

- **forall** also accepts masks:
  \[
  \text{forall}(i = 1:n, \ j = 1:n, \ y(i,j)/=0.) \ x(j,i) = 1.0/y(i,j)
  \]

- In its construct form, it looks like:
  \[
  \text{forall}(i = 2:n-1, \ j = 2:n-1) \\
  \quad a(i,j) = a(i,j-1) + a(i,j+1) + a(i-1,j) + a(i+1,j) \\
  \quad b(i,j) = a(i,j) \\
  \text{end forall}
  \]

It works like array assignments:
- Unlike **do**, there is no ordering of iterations, and changes appear as they were deferred
- Thus, no conflicts between reads and writes to **a**
- Assignment to \(b(i,j)\) takes place after that to \(a(i,j)\)

- Referenced procedures must be pure
- **forall** constructs can be nested and given a name
Using `do` loops (dependencies! loop order is crucial)

```fortran
  do j=2,n-1
    do i=2,n-1
      T(i,j) = ( T(i-1,j) + T(i+1,j) + &
                 T(i,j-1) + T(i,j+1) )/4.0
    enddo
  enddo
```

Using array syntax (compiler enforces correct semantics)

```fortran
  T(2:n-1,2:n-1) = ( T(1:n-2,2:n-1) + T(3:n,2:n-1) &
                     T(2:n-1,1:n-2) + T(2:n-1,3:n) )/4.0
```

Using `forall` (ditto, but more readable)

```fortran
  forall (i=2:n-1, j=2:n-1)
    T(i,j) = ( T(i-1,j) + T(i+1,j) + &
              T(i,j-1) + T(i,j+1) )/4.0
  end forall
```
Laplace Equation in Three Idioms

- Using **do** loops (dependencies! loop order is crucial)
  ```fortran
  do j=2,n-1
    do i=2,n-1
      T(i,j) = ( T(i-1,j) + T(i+1,j) + &
                T(i,j-1) + T(i,j+1) )/4.0
    enddo
  enddo
  ```

- Using array syntax (compiler enforces correct semantics)
  ```fortran
  T(2:n-1,2:n-1) = ( T(1:n-2,2:n-1) + T(3:n,2:n-1) &
                   + T(2:n-1,1:n-2) + T(2:n-1,3:n) )/4.0
  ```
Laplace Equation in Three Idioms

- **Using `do` loops (dependencies! loop order is crucial)**
  
  ```
  do  j=2,n-1
    do  i=2,n-1
        T(i,j) = ( T(i-1,j) + T(i+1,j) + &
                   T(i,j-1) + T(i,j+1) )/4.0
    enddo
  enddo
  ```

- **Using array syntax (compiler enforces correct semantics)**
  
  ```
  T(2:n-1,2:n-1) = ( T(1:n-2,2:n-1) + T(3:n,2:n-1) &
                   + T(2:n-1,1:n-2) + T(2:n-1,3:n) )/4.0
  ```

- **Using `forall` (ditto, but more readable)**
  
  ```
  forall (i=2:n-1, j=2:n-1)
    T(i,j) = ( T(i-1,j) + T(i+1,j) + &
              T(i,j-1) + T(i,j+1) )/4.0
  end forall
  ```
integer, parameter :: maxn=768, maxm=939, R=3
real, parameter :: sd=10.0, sr=10.0
real, parameter :: sd22=2.0*sd**2, sr22=2.0*sr**2
integer :: i,j,m,n
real :: B(maxn,maxm), A(maxn,maxm)
real :: z(-R:R,-R:R), aw(-R:R,-R:R)
real, dimension(-R:R,-R:R), parameter :: z0=&
    reshape((/ ((exp(-(m**2 + n**2)/sr22), m=-R, R), n=-R,R) /), (/ 2*R+1, 2*R+1 /))
...
do i=1,maxn ! These two cannot be changed into forall
    do j=1,maxm ! Why?
        z = 0.0
        forall (m=max(1,i-R):min(maxn,i+R))
            forall (n=max(1,j-R):min(maxm,j+R))
                aw(m-i,n-j) = A(m,n)
                z(m-i,n-j) = exp(-(aw(m-i,n-j)-A(i,j))**2/sd22)*z0(m-i,n-j)
            end forall
        end forall
        B(i,j) = sum(z*aw)/sum(z)
    end do
end do
integer, parameter :: maxn=768, maxm=939, R=3
real, parameter :: sd=10.0, sr=10.0
real, parameter :: sd22=2.0*sd**2, sr22=2.0*sr**2
integer :: i,j,m,n
real :: B(maxn,maxm), A(maxn,maxm)
real :: z(-R:R,-R:R), aw(-R:R,-R:R)
real, dimension(-R:R,-R:R), parameter :: z0=&
    reshape((/ ((exp(-(m**2 + n**2)/sr22), m=-R, R), n=-R,R) /), (/ 2*R+1, 2*R+1 /))
... 
do i=1,maxn    ! These two cannot be changed into forall 
do j=1,maxm    ! Why?
z = 0.0        ! Because this happens at every iteration, it’s a dependency!
  forall (m=max(1,i-R):min(maxn,i+R))
    forall (n=max(1,j-R):min(maxm,j+R))
      aw(m-i,n-j) = A(m,n)
      z(m-i,n-j) = exp(-(aw(m-i,n-j)-A(i,j))**2/sd22)*z0(m-i,n-j)
    end forall
  end forall
  B(i,j) = sum(z*aw)/sum(z)
end do 
end do
Array Reductions

- Reductions squeeze an array to a scalar
  - `all(mask)` returns true if all the elements of mask are true
  - `any(mask)` returns true if any of the elements of mask are true
  - `count(mask)` returns the number of `.true.` elements in mask
  - `maxval(array)` returns the maximum value of array
  - `minval(array)` returns the minimum value of array
  - `sum(array)` returns the sum of the elements of array
  - `product(array)` returns the product of the elements of array

- Or to an array of rank reduced by one, if you specify an optional dimension to perform reduction along, like in `sum(a(:,:,:), dim=2)`
More functions, good to know:

- `maxloc()` and `minloc()` return locations of maximum and minimum value respectively
- `cshift()` performs a *circular* shift along an array dimension
- `eoshift()` perform a *end-off* shift along an array dimension
- `spread()` increases by one the rank of an array expression
- `pack()` selects elements from an array according to a mask and packs them in a rank-1 array
- And `unpack()` does the reverse

But too much detail to cover in this introduction, look for them on your compiler documentation, and experiment
Matrix Algebra

- Vector and matrix multiplication functions
  - `dot_product(vector_a, vector_b)`
  - `matmul(matrix_a, matrix_b)`

- But the BLAS libraries are around
  - Widely used
  - Highly optimized implementations available

- Outstanding compilers include special purpose, optimized BLAS version for those calls

- Good compilers do not include BLAS, but give option to link them for those calls

- Average compilers do not shine for those calls

- Our advice: install a reputably good BLAS version and use it

- There is more to matrix algebra than matrix multiplies and vector products
Hands-on Session #1

- Re-write the Sieve of Eratosthenes algorithm using array syntax
Array Syntax

Input/Output
  Formatted I/O
  File I/O
  Namelist
  Internal Files
  Unformatted I/O
  Robust I/O
Array Syntax

Input/Output
   Formatted I/O
   File I/O
   Namelist
   Internal Files
   Unformatted I/O
   Robust I/O
Formatted I/O

- Data are manipulated in internal (usually binary) format
- Fortran Standard leaves internal format details up to the implementation
- Formatted I/O translates internal representation of variables into human readable format

Best practices:
- Use formatted I/O just for small amount of data meant to be read by humans
- Beware: human readable representation may cause problems because of rounding or not enough digits
- Do not use I/O inside heavy computations: inhibits some code optimizations, and significantly affects performance
program golden_ratio
  ! experiments with the golden ratio iterative relation
  implicit none
  integer, parameter :: rk = kind(1.0d0)
  real(rk) :: phi, phi_old
  real(rk) :: phi_start, tol
  integer :: i, max_iter

  write(*,*) 'Enter start value, tol, max iterations'
  read(*,*) phi_start, tol, max_iter

  phi_old = phi_start
  do i=1,max_iter
    phi = 1.0d0/phi_old + 1.0d0
    if (abs(phi - phi_old) < tol) exit
    phi_old = phi
  end do

  write(*,100) 'Start value:',phi_start
  write(*,100) 'Tolerance:',tol
  write(*,'(2(A," ",I11," "))') 'Ended at iteration:', i, 'of', max_iter
  write(*,100) 'Final value:',phi

100 format(A," ",F13.10)
end program golden_ratio
Iterative search for the Golden Ratio

program golden_ratio
  ! experiments with the golden ratio iterative relation
  implicit none
  integer, parameter :: rk = kind(1.0d0)
  real(rk) :: phi, phi_old
  real(rk) :: phi_start, tol
  integer :: i, max_iter

  write(*,*) 'Enter start value, tol, max iterations'
  read(*,*) phi_start, tol, max_iter

  phi_old = phi_start
  do i=1,max_iter
    phi = 1.0d0/phi_old + 1.0d0
    if (abs(phi - phi_old) < tol) exit
    phi_old = phi
  end do

  write(*,100) 'Start value:',phi_start
  write(*,100) 'Tolerance:',tol
  write(*,'(2(A," ",I11," "))') 'Ended at iteration:', i, 'of', max_iter
  write(*,100) 'Final value:',phi

100 format(A," ",F13.10)
end program golden_ratio
List Directed I/O

- The easiest way to do formatted I/O
- Specified using *
- Values are translated according to their types
- In the order they are listed on I/O statements
- No-nonsense, implementation dependent format
- Often outputs more digits than you actually care of

Best practices:
- Use it for terminal input
- Use it for input of white-space separated values
- Use it for quick output
- Not suitable for rigid tabular formats
Explicit formats

- Put you in total control of what is read/written

- Where format-list is a comma separated list of items, which can be:
  - string literals, usually in double quotes, emitted as-is
  - or proper edit descriptors, which dictate how a corresponding element on the I/O list should be converted

- Repeat counts can be used like in 5I3, which will convert 5 integer values
- Like in 2(I3,F7.4), which will convert 2 pairs, each made of an integer and a real value

- Formats must be specified on I/O statements
  - As a literal string, usually in single quotes
  - As a character expression
  - As a numeric label of a format statement in the same program unit (traditionally, before its end), reusable in many statements
Iterative search for the Golden Ratio

program golden_ratio
! experiments with the golden ratio iterative relation
  implicit none
  integer, parameter :: rk = kind(1.0d0)
  real(rk) :: phi, phi_old
  real(rk) :: phi_start, tol
  integer :: i, max_iter

  write(*,*) 'Enter start value, tol, max iterations'
  read(*,*) phi_start, tol, max_iter

  phi_old = phi_start
  do i=1,max_iter
    phi = 1.0d0/phi_old + 1.0d0
    if (abs(phi - phi_old) < tol) exit
    phi_old = phi
  end do

  write(*,100) 'Start value:',phi_start
  write(*,100) 'Tolerance:',tol
  write(*,'(2(A," ",I11," "))') 'Ended at iteration:',i,' of ',max_iter
  write(*,100) 'Final value:',phi

100 format(A," ",F13.10)
end program golden_ratio
Explicit formats

- Put you in total control of what is read/written
- Specified by (format-list)
Explicit formats

- Put you in total control of what is read/written
- Specified by \((\text{format-list})\)
- Where \(\text{format-list}\) is a comma separated list of items, which can be:
  - string literals, usually in double quotes, emitted \textit{as-is}
  - or proper \textit{edit descriptors}, which dictate how a corresponding element on the I/O list should be converted

\(\text{Repeat counts can be used:}\)
\(\text{Like in } 5I3\) which will convert 5 integer values
\(\text{Like in } 2(I3,F7.4)\) which will convert 2 pairs, each made of an integer and a real value

\(\text{Formats must be specified on I/O statements:}\)
\(\text{As a literal string, usually in single quotes}\)
\(\text{As a character expression}\)
\(\text{As a numeric label of a format statement in the same program unit (traditionally, before its end), reusable in many statements}\)
program golden_ratio
! experiments with the golden ratio iterative relation
implicit none
integer, parameter :: rk = kind(1.0d0)
real(rk) :: phi, phi_old
real(rk) :: phi_start, tol
integer :: i, max_iter

write(*,*) 'Enter start value, tol, max iterations'
read(*,*) phi_start, tol, max_iter

phi_old = phi_start
do i=1,max_iter
   phi = 1.0d0/phi_old + 1.0d0
   if (abs(phi - phi_old) < tol) exit
   phi_old = phi
end do

write(*,100) 'Start value:',phi_start
write(*,100) 'Tolerance:',tol
write(*,'(2(A," ",I11," "))') 'Ended at iteration:', i, 'of', max_iter
write(*,100) 'Final value:',phi

100 format(A," ",F13.10)
end program golden_ratio
Explicit formats

- Put you in total control of what is read/written
- Specified by (format-list)
- Where format-list is a comma separated list of items, which can be:
  - string literals, usually in double quotes, emitted as-is
  - or proper edit descriptors, which dictate how a corresponding element on the I/O list should be converted
- Repeat counts can be used
  - Like in 5I3, which will convert 5 integer values
  - Like in 2(I3,F7.4), which will convert 2 pairs, each made of an integer and a real value
program golden_ratio
! experiments with the golden ratio iterative relation
    implicit none
    integer, parameter :: rk = kind(1.0d0)
    real(rk) :: phi, phi_old
    real(rk) :: phi_start, tol
    integer :: i, max_iter

    write(*,*) 'Enter start value, tol, max iterations'
    read(*,*) phi_start, tol, max_iter

    phi_old = phi_start
    do i=1,max_iter
        phi = 1.0d0/phi_old + 1.0d0
        if (abs(phi - phi_old) < tol) exit
        phi_old = phi
    end do

    write(*,100) 'Start value:',phi_start
    write(*,100) 'Tolerance:',tol
    write(*,'(2(A," ",I11," "))') 'Ended at iteration:',i,'of',max_iter
    write(*,100) 'Final value:',phi

100 format(A," ",F13.10)
end program golden_ratio
Explicit formats

- Put you in total control of what is read/written
- Specified by \textit{(format-list)}
- Where \textit{format-list} is a comma separated list of items, which can be:
  - string literals, usually in double quotes, emitted \textit{as-is}
  - or proper \textit{edit descriptors}, which dictate how a corresponding element on the I/O list should be converted
- \textit{Repeat counts} can be used
  - Like in 5I3, which will convert 5 \texttt{integer} values
  - Like in 2(I3,F7.4), which will convert 2 pairs, each made of an \texttt{integer} and a \texttt{real} value
- Formats must be specified on I/O statements
  - As a literal string, usually in single quotes
  - As a character expression
  - As a numeric label of a \texttt{format} statement in the same program unit (traditionally, before its end), reusable in many statements
Edit Descriptors: **characters** and **integers**

- **A** is used to translate **character** values
  - **A** will emit the value as is
  - **A10** will emit 10 characters, truncating the value if longer, right justifying it if shorter
  - Beware: leading white-space skipped on input
  - Beware: **A10** and **10A** mean very different things!
Edit Descriptors: characters and integers

- **A** is used to translate character values
  - ▶ **A** will emit the value as is
  - ▶ **A10** will emit 10 characters, truncating the value if longer, right justifying it if shorter
  - ▶ Beware: leading white-space skipped on input
  - ▶ Beware: **A10** and **10A** mean very different things!

- **I** is used to translate integer values
  - ▶ **I6** will emit up to 6 characters (sign included!), right justified with blanks
  - ▶ **I6.3** will emit 6 characters (sign included!), containing at least 3 (possibly zero) digits, right justified with blanks
  - ▶ Beware: again, **I10** and **10I** mean very different things!
Edit Descriptors: reals

- **F** can be used to translate real values
  - **F8.3** will emit up to 8 characters (sign and decimal point included!) in total, with 3 decimal digits (possibly zero), right justified with blanks
  - Beware: if **F6.2** is specified in input, and \(-12345\) is met, the value \(-123.45\) will be read in!
  - Beware: if **F6.2** is specified in input, and \(-1.234\) is met, the value \(-1.234\) will be read in anyhow!
- Beware of rounding: internal representation could have more precision than specified in edit descriptors
More Edit Descriptors for real values

- E (or D) can also be used to translate real values
  - Exponential form is used (mantissa in the [0,1) range)
  - Values $|x| < 10^{99}$, as $-1.5372 \times 10^{98}$, will be converted like: $-.15372E+99$
  - Values $|x| \geq 10^{99}$, as $-1.5372 \times 10^{99}$, will be converted like: $-.15372+100$
  - E15.7 will emit up to 15 characters (sign, decimal point, and exponent field included!), with 7 decimal mantissa digits (possibly zero), right justified with blanks
  - Ditto for E15.7E4, except that 4 digits will be used for exponent
  - Again, input is more liberal
More Edit Descriptors for reals

- **E** (or **D**) can also be used to translate real values
  - Exponential form is used (mantissa in the [0,1) range)
  - Values $|x| < 10^{99}$, as $-1.5372 \times 10^{98}$, will be converted like: $-.15372E+99$
  - Values $|x| \geq 10^{99}$, as $-1.5372 \times 10^{99}$, will be converted like: $-.15372+100$
  - **E15.7** will emit up to 15 characters (sign, decimal point, and exponent field included!), with 7 decimal mantissa digits (possibly zero), right justified with blanks
  - Ditto for **E15.7E4**, except that 4 digits will be used for exponent
  - Again, input is more liberal

- And more can be used to the same purpose
  - Like **E**N (engineering notation), same as **E**, with exponent always multiple of 3
  - Like **G**, which uses the most suitable between **F** and **E**, depending on the value magnitude
Even More Edit Descriptors

- /
  - Forces a new line on output
  - Skips to next line on input

And more... browse your compiler manuals
Even More Edit Descriptors

▶ /
  ▶ Forces a new line on output
  ▶ Skips to next line on input

▶ Leading sign of numeric values
  ▶ SP forces following numeric conversions to emit a leading +
    character for positive values
  ▶ SS restores the default (sign is suppressed for positive values)

▶ And more... browse your compiler manuals
Even More Edit Descriptors

- /
  - Forces a new line on output
  - Skips to next line on input

- Leading sign of numeric values
  - _SP_ forces following numeric conversions to emit a leading + character for positive values
  - _SS_ restores the default (sign is suppressed for positive values)

- Embedded blanks in numeric input fields
  - _BZ_ forces embedded blanks to be treated as 0 digits
  - _BN_ restores the default (blanks are skipped)

And more... browse your compiler manuals
Even More Edit Descriptors

- /
  - Forces a new line on output
  - Skips to next line on input

- Leading sign of numeric values
  - $SP$ forces following numeric conversions to emit a leading + character for positive values
  - $SS$ restores the default (sign is suppressed for positive values)

- Embedded blanks in numeric input fields
  - $BZ$ forces embedded blanks to be treated as 0 digits
  - $BN$ restores the default (blanks are skipped)

- And more... browse your compiler manuals
complexes and Arrays

- **complex** values are made of two reals
  - Thus two edit descriptors must be provided
  - First one for real part, second one for imaginary part
complexes and Arrays

- **complex** values are made of two reals
  - Thus two edit descriptors must be provided
  - First one for real part, second one for imaginary part

- Arrays are indexed collections of elements
  - Thus a proper edit descriptor must be provided for each element
  - And if elements are of **complex**, or derived types, see above
Fortran I/O is Robustly Designed

- What if more characters than needed are present on an input line?
- After `read`, remaining ones are ignored up to end of line.
- What if the list of items to read/write is exhausted before end of edit descriptors in a format?
  - Following edit descriptors are ignored.
- What if the list of edit descriptors in a format is exhausted before end of items to read/write?
  - Easy answer: I/O continues on a new line, reapplying the format list from its beginning, quite handy for arrays.
  - Could be more complex, look for reversion to know more.
- What if a numeric value is too big to fit the characters you specified on its corresponding edit descriptor?
  - The field is filled with asterisks (i.e. `*`).
- What if a type mismatch happens between an item to read/write and its corresponding edit descriptor?
  - Your fault, you are in for a runtime, implementation defined surprise!
Fortran I/O is Robustly Designed

- What if more characters than needed are present on an input line?
  - After `read`, remaining ones are ignored up to end of line
Fortran I/O is Robustly Designed

- What if more characters than needed are present on an input line?
  - After `read`, remaining ones are ignored up to end of line
- What if the list of items to read/write is exhausted before end of edit descriptors in a format?

  - Easy answer: I/O continues on a new line, reapplying the format list from its beginning, quite handy for arrays
  - Could be more complex, look for reversion to know more
- What if a numeric value is too big to fit the characters you specified on its corresponding edit descriptor?
  - The field is filled with asterisks (i.e. `*`)
- What if a type mismatch happens between an item to read/write and its corresponding edit descriptor?
  - Your fault, you are in for a runtime, implementation defined surprise!
Fortran I/O is Robustly Designed

- What if more characters than needed are present on an input line?
  - After `read`, remaining ones are ignored up to end of line
- What if the list of items to read/write is exhausted before end of edit descriptors in a format?
  - Following edit descriptors are ignored
Fortran I/O is Robustly Designed

- What if more characters than needed are present on an input line?
  - After `read`, remaining ones are ignored up to end of line

- What if the list of items to read/write is exhausted before end of edit descriptors in a format?
  - Following edit descriptors are ignored

- What if the list of edit descriptors in a format is exhausted before end of items to read/write?
program iterative_inversion
  ! experiments with matrix iterative inversion
  implicit none
  real, dimension(4,4) :: a, x, x_old, x_start
  real :: tol, err
  integer :: i, max_iter

  write(*,*) 'Enter 4x4 matrix to invert'
  read(*,*) a
  write(*,*) 'Enter 4x4 start matrix'
  read(*,*) x_start
  write(*,*) 'Enter tol, max iterations'
  read(*,*) tol, max_iter

  x_old = x_start
  do i=1,max_iter
    x = 2.0*x_old - matmul(x_old,matmul(a,x_old))
    err = maxval(abs(x - x_old))
    if (err < tol) exit
    x_old = x
  end do

  write(*,'("Matrix to invert:")')
  write(*,100) a
  write(*,'(/,"Start matrix:")')
  write(*,100) x_start
  write(*,'(/,A,"Tolerance:" ,E15.7)') 'Tolerance:',tol
  write(*,'(/,2(A,"I11," ))') 'Ended at iteration:', i, 'of', max_iter
  write(*,'("Final matrix:")')
  write(*,100) x

  100 format(4(E15.7," "))
end program iterative_inversion
Fortran I/O is Robustly Designed

- What if more characters than needed are present on an input line?
  - After `read`, remaining ones are ignored up to end of line
- What if the list of items to read/write is exhausted before end of edit descriptors in a format?
  - Following edit descriptors are ignored
- What if the list of edit descriptors in a format is exhausted before end of items to read/write?
  - Easy answer: I/O continues on a new line, reapplying the format list from its beginning, quite handy for arrays

What if a numeric value is too big to fit the characters you specified on its corresponding edit descriptor?

- The field is filled with asterisks (i.e. `*`)

What if a type mismatch happens between an item to read/write and its corresponding edit descriptor?

- Your fault, you are in for a runtime, implementation defined surprise!
Fortran I/O is Robustly Designed

- What if more characters than needed are present on an input line?
  - After `read`, remaining ones are ignored up to end of line

- What if the list of items to read/write is exhausted before end of edit descriptors in a format?
  - Following edit descriptors are ignored

- What if the list of edit descriptors in a format is exhausted before end of items to read/write?
  - Easy answer: I/O continues on a new line, reapplying the format list from its beginning, quite handy for arrays
  - Could be more complex, look for `reversion` to know more
Fortran I/O is Robustly Designed

- What if more characters than needed are present on an input line?
  - After `read`, remaining ones are ignored up to end of line
- What if the list of items to read/write is exhausted before end of edit descriptors in a format?
  - Following edit descriptors are ignored
- What if the list of edit descriptors in a format is exhausted before end of items to read/write?
  - Easy answer: I/O continues on a new line, reapplying the format list from its beginning, quite handy for arrays
  - Could be more complex, look for `reversion` to know more
- What if a numeric value is too big to fit the characters you specified on its corresponding edit descriptor?
Fortran I/O is Robustly Designed

- What if more characters than needed are present on an input line?
  - After `read`, remaining ones are ignored up to end of line
- What if the list of items to read/write is exhausted before end of edit descriptors in a format?
  - Following edit descriptors are ignored
- What if the list of edit descriptors in a format is exhausted before end of items to read/write?
  - Easy answer: I/O continues on a new line, reapplying the format list from its beginning, quite handy for arrays
  - Could be more complex, look for `reversion` to know more
- What if a numeric value is too big to fit the characters you specified on its corresponding edit descriptor?
  - The field is filled with asterisks (i.e. `*`)
Fortran I/O is Robustly Designed

- What if more characters than needed are present on an input line?
  - After `read`, remaining ones are ignored up to end of line

- What if the list of items to read/write is exhausted before end of edit descriptors in a format?
  - Following edit descriptors are ignored

- What if the list of edit descriptors in a format is exhausted before end of items to read/write?
  - Easy answer: I/O continues on a new line, reapplying the format list from its beginning, quite handy for arrays
  - Could be more complex, look for `reversion` to know more

- What if a numeric value is too big to fit the characters you specified on its corresponding edit descriptor?
  - The field is filled with asterisks (i.e. `*`)

- What if a type mismatch happens between an item to read/write and its corresponding edit descriptor?
Fortran I/O is Robustly Designed

- What if more characters than needed are present on an input line?
  - After `read`, remaining ones are ignored up to end of line
- What if the list of items to read/write is exhausted before end of edit descriptors in a format?
  - Following edit descriptors are ignored
- What if the list of edit descriptors in a format is exhausted before end of items to read/write?
  - Easy answer: I/O continues on a new line, reapplying the format list from its beginning, quite handy for arrays
  - Could be more complex, look for `reversion` to know more
- What if a numeric value is too big to fit the characters you specified on its corresponding edit descriptor?
  - The field is filled with asterisks (i.e. `*`)
- What if a type mismatch happens between an item to read/write and its corresponding edit descriptor?
  - Your fault, you are in for a runtime, implementation defined surprise!
Hands-on Session #2

- Play with `golden.f90` and `itinv.f90`:
  - trying good and bad inputs
  - giving less or more inputs than needed
  - changing format descriptors
Array Syntax

Input/Output
  Formatted I/O
  File I/O
  Namelist
  Internal Files
  Unformatted I/O
  Robust I/O
Iterative search for the Golden Ratio

program golden_ratio
! experiments with the golden ratio iterative relation
  implicit none
  integer, parameter :: rk = kind(1.0d0)
  real(rk) :: phi, phi_old
  real(rk) :: phi_start, tol
  integer :: i, max_iter

  open(11,FILE='golden.in',STATUS='old')
  read(11,*) phi_start, tol, max_iter
  close(11)

  phi_old = phi_start
  do i=1,max_iter
    phi = 1.0d0/phi_old + 1.0d0
    if (abs(phi - phi_old) < tol) exit
    phi_old = phi
  end do

  open(12,FILE='golden.out')
  write(12,100) 'Start value:',phi_start
  write(12,100) 'Tolerance:',tol
  write(12,100) 'Ended at iteration:', i, 'of', max_iter
  write(12,100) 'Final value:',phi
  close(12)

  100 format(A," ",F13.10)
end program golden_ratio
Iterative search for the Golden Ratio

```fortran
program golden_ratio
    ! experiments with the golden ratio iterative relation
    implicit none
    integer, parameter :: rk = kind(1.0d0)
    real(rk) :: phi, phi_old
    real(rk) :: phi_start, tol
    integer :: i, max_iter

    open(11, FILE='golden.in', STATUS='old')
    read(11, *) phi_start, tol, max_iter
    close(11)

    phi_old = phi_start
    do i=1, max_iter
        phi = 1.0d0/phi_old + 1.0d0
        if (abs(phi - phi_old) < tol) exit
        phi_old = phi
    end do

    open(12, FILE='golden.out')
    write(12, 100) 'Start value:', phi_start
    write(12, 100) 'Tolerance:', tol
    write(12, '2(A,",",I11")') 'Ended at iteration:', i, 'of', max_iter
    write(12, 100) 'Final value:', phi
    close(12)

100 format(A,",",F13.10)
end program golden_ratio
```
opening a File for I/O

open (u, FILE=\textit{file\_name}[,,\textit{option}][,,\textit{option}][...])

- \textit{u} is an integer, positive expression specifying a \textit{file handle}
opening a File for I/O

```fortran
open (u, FILE= file_name[, option][, option][...])
```

- `u` is an integer, positive expression specifying a *file handle*
- `file_name` is a string specifying file name (and possibly path) in your file system
opening a File for I/O

open (u, FILE=file_name[, option][, option][…])

- u is an integer, positive expression specifying a *file handle*
- `file_name` is a string specifying file name (and possibly path) in your file system
- *file handle* is then used as first argument to *read* and *write*
**opening a File for I/O**

```fortran
open (u, FILE=\texttt{file\_name}[, \texttt{option}][, \texttt{option}][...])
```

- \texttt{u} is an integer, positive expression specifying a *file handle*
- \texttt{file\_name} is a string specifying file name (and possibly path) in your file system
- *file handle* is then used as first argument to \texttt{read} and \texttt{write}
  - When you pass a \texttt{*} instead, you are using pre-opened units mapping to user terminal

For error messages, \texttt{0} is commonly used, but \texttt{error\_unit} from \texttt{iso\_fortran\_env} module is portable.
opening a File for I/O

open (\(u, \text{FILE}=\text{file\_name}[,, \text{option}][,, \text{option}][...])

- \(u\) is an integer, positive expression specifying a \textit{file handle}
- \textit{file\_name} is a string specifying file name (and possibly path) in your file system

- \textit{file handle} is then used as first argument to \texttt{read} and \texttt{write}
  - When you pass a \(\ast\) instead, you are using pre-opened units mapping to user terminal
  - Which usually means 5 for \texttt{read} and 6 for \texttt{write}, but \(\ast\), or \texttt{input\_unit} and \texttt{output\_unit} from \texttt{iso\_fortran\_env} Fortran 2003 module are more portable
opening a File for I/O

```
open(u,FILE=\texttt{file\_name}[,\ \texttt{option}][,\ \texttt{option}][...])
```

- \texttt{u} is an integer, positive expression specifying a \textit{file handle}
- \texttt{file\_name} is a string specifying file name (and possibly path) in your file system

- \textit{file handle} is then used as first argument to \texttt{read} and \texttt{write}
  - When you pass a * instead, you are using pre-opened units mapping to user terminal
  - Which usually means 5 for \texttt{read} and 6 for \texttt{write}, but *, or \texttt{input\_unit} and \texttt{output\_unit} from \texttt{iso\_fortran\_env} Fortran 2003 module are more portable
  - For error messages, 0 is commonly used, but \texttt{error\_unit} from \texttt{iso\_fortran\_env} module is portable
Some open Options

- **ACTION=act** specifies allowed actions
  - use 'read' to only read
  - use 'write' to only write
  - use 'readwrite' (the default) to allow both

- **STATUS=st** tells how to behave wrt file existence:
  - use 'old' to open a file that must already exist
  - use 'new' to open a file that must not exist
  - use 'replace' to open a new file, even if one already exists
  - use 'unknown' (the default) to leave it up to the implementation (in all cases we know of, this means 'replace')

- **POSITION=pos** tells where to start I/O on an existing file:
  - use 'rewind' (the default) to start at beginning of file
  - use 'append' to start at end of file
Some `open` Options

- **ACTION**=`act` specifies allowed actions
  - use `read` to only read
  - use `write` to only write
  - use `readwrite` (the default) to allow both

- **STATUS**=`st` tells how to behave wrt file existence:
  - use `old` to open a file that must already exist
  - use `new` to open a file that must not exist
  - use `replace` to open a new file, even if one already exists
  - use `unknown` (the default) to leave it up to the implementation (in all cases we know of, this means `replace`
Some open Options

- **ACTION=act** specifies allowed actions
  - use ‘read’ to only read
  - use ‘write’ to only write
  - use ‘readwrite’ (the default) to allow both

- **STATUS=st** tells how to behave wrt file existence:
  - use ‘old’ to open a file that must already exist
  - use ‘new’ to open a file that must not exist
  - use ‘replace’ to open a new file, even if one already exists
  - use ‘unknown’ (the default) to leave it up to the implementation (in all cases we know of, this means ‘replace’)

- **POSITION=pos** tells where to start I/O on an existing file
  - use ‘rewind’ (the default) to start at beginning of file
  - use ‘append’ to start at end of file
program golden_ratio
  ! experiments with the golden ratio iterative relation
  implicit none
  integer, parameter :: rk = kind(1.0d0)
  real(rk) :: phi, phi_old
  real(rk) :: phi_start, tol
  integer :: i, max_iter

  open(11,FILE='golden.in',STATUS='old')
  read(11,*) phi_start, tol, max_iter
  close(11)

  phi_old = phi_start
  do i=1,max_iter
    phi = 1.0d0/phi_old + 1.0d0
    if (abs(phi - phi_old) < tol) exit
    phi_old = phi
  end do

  open(12,FILE='golden.out')
  write(12,100) 'Start value:',phi_start
  write(12,100) 'Tolerance:',tol
  write(12,'(2(A,",",I11," "))') 'Ended at iteration:', i, 'of', max_iter
  write(12,100) 'Final value:',phi
  close(12)

  100 format(A," ",F13.10)
end program golden_ratio
How to close a File

```close(u[, STATUS=st])

- `close` completes all pending I/O operations and disassociates the file from the unit
```
How to close a File

```plaintext
close(u[, STATUS=st])
```

- `close` completes all pending I/O operations and disassociates the file from the unit
- `close` is automatically executed on all open files at program end, but closing a file explicitly when you are done with it is a good practice

- use `'keep'` to preserve the file (it's the default)
- use `'delete'` to remove it (good for files used for temporary storage)
How to close a File

\texttt{close(u[, STATUS=st])}

- \texttt{close} completes all pending I/O operations and disassociates the file from the unit
- \texttt{close} is automatically executed on all open files at program end, but closing a file explicitly when you are done with it is a good practice
- \texttt{st} tells what to do with the file after closing it
  - use ‘\texttt{keep}’ to preserve the file (it’s the default)
  - use ‘\texttt{delete}’ to remove it (good for files used for temporary storage)
Outline

Array Syntax

Input/Output
  Formatted I/O
  File I/O
  Namelist
  Internal Files
  Unformatted I/O
  Robust I/O
program golden_ratio
  ! experiments with the golden ratio iterative relation
  implicit none
  integer, parameter :: rk = kind(1.0d0)
  real(rk) :: phi, phi_old
  real(rk) :: phi_start, tol
  integer :: i, max_iter

  namelist /golden_inputs/ phi_start, tol, max_iter

  open(11, FILE='golden.in', STATUS='old')
  read(11, golden_inputs)
  close(11)

  phi_old = phi_start
  do i=1, max_iter
    phi = 1.0d0/phi_old + 1.0d0
    if (abs(phi - phi_old) < tol) exit
    phi_old = phi
  end do

  open(12, FILE='golden.out')
  write(12, 100) 'Start value:', phi_start
  write(12, 100) 'Tolerance:', tol
  write(12, '(2(A, " ",I11," "))') 'Ended at iteration:', i, 'of', max_iter
  write(12, 100) 'Final value:', phi
  close(12)

  100 format(A," ",F13.10)
end program golden_ratio
Iterative search for the Golden Ratio

program golden_ratio
! experiments with the golden ratio iterative relation
  implicit none
  integer, parameter :: rk = kind(1.0d0)
  real(rk) :: phi, phi_old
  real(rk) :: phi_start, tol
  integer :: i, max_iter

  namelist /golden_inputs/ phi_start, tol, max_iter

  open(11,FILE='golden.in',STATUS='old')
  read(11,golden_inputs)
  close(11)

  phi_old = phi_start
  do i=1,max_iter
    phi = 1.0d0/phi_old + 1.0d0
    if (abs(phi - phi_old) < tol) exit
    phi_old = phi
  end do

  open(12,FILE='golden.out')
  write(12,100) 'Start value:',phi_start
  write(12,100) 'Tolerance:',tol
  write(12,100) 'Ended at iteration:', i, 'of', max_iter
  write(12,100) 'Final value:',phi
  close(12)

  100 format(A," ",F13.10)
end program golden_ratio
namelists allow input/output of annotated lists of values
namelists are input/output of annotated lists of values

- Performed by `read` or `write` statements that do not have an I/O list and in which format is replaced by a namelist name.

```plaintext
&golden_inputs
tol=1.e-4 ! tolerance
phi_start=5.0 ! 0th iteration
max_iter=10000000 /
```

- Items missing in the input will retain previous value.
- Items can be added to a namelist in different statements, but a code like this easily misleads readers (and you read your own codes, don't you?)

- Use them to make input robust, in output mostly good for debugging.
namelists allow input/output of annotated lists of values

Performed by `read` or `write` statements that do not have an I/O list and in which format is replaced by a namelist name

File content is structured, self-describing, order independent, comments are allowed:

```plaintext
&golden_inputs
tol=1.e-4     ! tolerance
phi_start=5.0  ! 0th iteration
max_iter=10000000 /
```
namelists allow input/output of annotated lists of values

- Performed by `read` or `write` statements that do not have an I/O list and in which format is replaced by a namelist name
- File content is structured, self-describing, order independent, comments are allowed:

```
&golden_inputs
  tol=1.e-4       ! tolerance
  phi_start=5.0   ! 0th iteration
  max_iter=10000000 /
```

- Items missing in the input will retain previous value
namelists allow input/output of annotated lists of values

- Performed by **read** or **write** statements that do not have an I/O list and in which format is replaced by a namelist name

- File content is structured, self-describing, order independent, comments are allowed:

```
&golden_inputs
  tol=1.e-4   ! tolerance
  phi_start=5.0  ! 0th iteration
  max_iter=10000000 /
```

- Items missing in the input will retain previous value

- Items can be added to a namelist in different statements, but a code like this easily misleads readers (and you read your own codes, don’t you?)
NamelistS allow input/output of annotated lists of values

Performed by read or write statements that do not have an I/O list and in which format is replaced by a namelist name

File content is structured, self-describing, order independent, comments are allowed:

```
&golden_inputs
tol=1.e-4 ! tolerance
phi_start=5.0 ! 0th iteration
max_iter=10000000 /
```

Items missing in the input will retain previous value

Items can be added to a namelist in different statements, but a code like this easily misleads readers (and you read your own codes, don’t you?)

Use them to make input robust, in output mostly good for debugging
Outline

Array Syntax

Input/Output
  Formatted I/O
  File I/O
  Namelist
  Internal Files
  Unformatted I/O
  Robust I/O
program golden_ratio
! experiments with the golden ratio iterative relation
  implicit none
  integer, parameter :: rk = kind(1.0d0)
  real(rk) :: phi, phi_old
  real(rk) :: phi_start, tol
  integer :: i, max_iter, test_no
  character(15) :: outfilename

  namelist /golden_inputs/ phi_start, tol, max_iter, test_no

  test_no = 1
  open(11,FILE='golden.in',STATUS='old')
  read(11,golden_inputs)
  close(11)

  phi_old = phi_start
  do i=1,max_iter
    phi = 1.0d0/phi_old + 1.0d0
    if (abs(phi - phi_old) < tol) exit
    phi_old = phi
  end do

  write(outfilename,'("golden",I5.5,".out")') test_no
  open(12,FILE=outfilename)
  write(12,100) 'Start value:',phi_start
  write(12,100) 'Tolerance:',tol
  write(12,'(2(A," ",I11," "))') 'Ended at iteration:', i, 'of', max_iter
  write(12,100) 'Final value:',phi
  close(12)

  100 format(A," ",F13.10)
end program golden_ratio
Iterative search for the Golden Ratio

```fortran
program golden_ratio
! experiments with the golden ratio iterative relation
  implicit none
  integer, parameter :: rk = kind(1.0d0)
  real(rk) :: phi, phi_old
  real(rk) :: phi_start, tol
  integer :: i, max_iter, test_no
  character(15) :: outfilename

  namelist /golden_inputs/ phi_start, tol, max_iter, test_no

  test_no = 1
  open(11,FILE='golden.in',STATUS='old')
  read(11,golden_inputs)
  close(11)

  phi_old = phi_start
  do i=1,max_iter
    phi = 1.0d0/phi_old + 1.0d0
    if (abs(phi - phi_old) < tol) exit
    phi_old = phi
  end do

  write(outfilename,'("golden",I5.5,".out")') test_no
  open(12,FILE=outfilename)
  write(12,100) 'Start value:',phi_start
  write(12,100) 'Tolerance:',tol
  write(12,'(2(A,"",I11," "))') 'Ended at iteration:', i, 'of', max_iter
  write(12,100) 'Final value:',phi
  close(12)

  100 format(A," ",F13.10)
end program golden_ratio
```
Internal Files

- `character` variables of default kind can be specified in place of units in `read` and `write` statements

- Writing to internal files is good to:
  - Dynamically build file names according to a pattern (like number of iterations)
  - Dynamically assemble complex I/O formats, depending on actual data
  - Prepare complex labels for plot data formats
  - Build commands to be sent to hardware devices

- Reading from internal files can be useful to read complex inputs:
  - You have a textual input file sporting different formats
  - And the right format depends on actual data in the file
  - Just read each line in a `character` variable, suitably sized
  - Pick the suitable format
  - And use it to read from the variable itself
Internal Files

- **character** variables of default kind can be specified in place of units in `read` and `write` statements

- Writing to internal files is good to:
  - dynamically build file names according to a pattern (like number of iterations)
  - dynamically assemble complex I/O formats, depending on actual data
  - prepare complex labels for plot data formats
  - build commands to be sent to hardware devices
  - ...

- Reading from internal files can be useful to read complex inputs
  - You have a textual input file sporting different formats
  - And the right format depends on actual data in the file
  - Just read each line in a `character` variable, suitably sized
  - Pick the suitable format
  - And use it to read from the variable itself
Internal Files

- **character** variables of default kind can be specified in place of units in **read** and **write** statements.

- Writing to internal files is good to:
  - dynamically build file names according to a pattern (like number of iterations)
  - dynamically assemble complex I/O formats, depending on actual data
  - prepare complex labels for plot data formats
  - build commands to be sent to hardware devices
  - ...

- Reading from internal files can be useful to read complex inputs:
  - You have a textual input file sporting different formats
  - And the right format depends on actual data in the file
  - Just read each line in a **character** variable, suitably sized
  - Pick the suitable format
  - And use it to read from the variable itself
Play with *goldenfile.f90*, *goldenfnl.f90*, and *goldeniio.f90*:

- writing input files
- writing good and bad data in input files
- giving input files wrong file names
Array Syntax

Input/Output
- Formatted I/O
- File I/O
- Namelist
- Internal Files
- Unformatted I/O
- Robust I/O
Unformatted I/O

- Formatted I/O is good, but:
  - internal data format is much more compact
  - and roundoff may happen, making recovery of original values impossible
  - and conversion takes time

- Unformatted I/O is used to store and recover data in internal representation
- Just give FORM='unformatted' option when opening the file
- And omit format in read and write statements

- Unformatted I/O is performed on a record basis
- As we'll see, this allows walking your files backward and forward
- But has interesting consequences, as more than your data is written to your file...
program iterative_inversion
  ! experiments with matrix iterative inversion
  implicit none
  real, dimension(4,4) :: a, x, x_old, x_start
  real :: tol, err
  integer :: i, max_iter

  open(21,FILE='input.dat',FORM='unformatted',STATUS='old')
  read(21) a
  read(21) x_start
  read(21) tol, max_iter
  close(21)

  x_old = x_start
  do i=1,max_iter
    x = 2.0*x_old - matmul(x_old,matmul(a,x_old))
    err = maxval(abs(x - x_old))
    if (err < tol) exit
    x_old = x
  end do

  open(22,FILE='itinv.dat',FORM='unformatted')
  write(22) a
  write(22) x_start
  write(22) tol, max_iter
  write(22) i
  write(22) x
  close(22)

  end program iterative_inversion
program iterative_inversion
  ! experiments with matrix iterative inversion
  implicit none
  real, dimension(4,4) :: a, x, x_old, x_start
  real :: tol, err
  integer :: i, max_iter

  open(21,FILE='input.dat',FORM='unformatted',STATUS='old')
  read(21) a
  read(21) x_start
  read(21) tol,max_iter
  close(21)

  x_old = x_start
  do i=1,max_iter
    x = 2.0*x_old - matmul(x_old,matmul(a,x_old))
    err = maxval(abs(x - x_old))
    if (err < tol) exit
    x_old = x
  end do

  open(22,FILE='itinv.dat',FORM='unformatted')
  write(22) a
  write(22) x_start
  write(22) tol,max_iter
  write(22) i
  write(22) x
  close(22)

end program iterative_inversion
Unformatted I/O

- Formatted I/O is good, but:
  - internal data format is much more compact
  - and roundoff may happen, making recovery of original values impossible
  - and conversion takes time

- Unformatted I/O is used to store and recover data in internal representation
  - Just give `FORM=’unformatted’` option when opening the file
  - And omit format in `read` and `write` statements
Unformattted I/O

- Formatted I/O is good, but:
  - internal data format is much more compact
  - and roundoff may happen, making recovery of original values impossible
  - and conversion takes time

- Unformatted I/O is used to store and recover data in internal representation
  - Just give `FORM='unformatted'` option when opening the file
  - And omit format in `read` and `write` statements

- Unformatted I/O is performed on a record basis
  - In unformatted mode, each `write` writes a record
  - As we’ll see, this allows walking your files backward and forward
  - But has interesting consequences, as more than your data is written to your file...
Modify `itinv.f90` to perform unformatted I/O

To test it, you’ll need an additional program:

- taking text input from keyboard or initializing all needed data
- to write a good unformatted input file for the new version of `itinv.f90`
Try different ways to output the results:

- **element-wise**
  ```
  do j=1,n
    do i=1,n
      write(79) a(i,j)
    end do
  end do
  ```

- **column-wise, using an implied do-loop:**
  ```
  do j=1,n
    write(79) (a(i,j), i=1,n) ! a(:,j) will also do
  end do
  ```

- **with two implied do-loops:**
  ```
  write(79) ((a(i,j), i=1,n), j=1,n)
  ```

Can you spot the difference?
Try different ways to output the results:

- **element-wise**
  
  ```
  do j=1,n
    do i=1,n
      write(79) a(i,j)
    end do
  end do
  ```

- **column-wise, using an implied do-loop:**
  
  ```
  do j=1,n
    write(79) (a(i,j), i=1,n) ! a(:,j) will also do
  end do
  ```

- **with two implied do-loops:**
  
  ```
  write(79) ((a(i,j), i=1,n), j=1,n)
  ```

Can you spot the difference?

Not a big issue for $4 \times 4$ matrices, but think of a $256 \times 256 \times 1024$ grid!
File Positioning

- read always advance to next record, even if you read only part of the record (or possibly nothing)
- backspace$(u)$ moves position for subsequent I/Os to the record preceding the current one
- rewind$(u)$ moves position for subsequent I/Os to file beginning
- To allow positioning back and forth, a four bytes record marker is added in 32 bit mode (eight bytes in 64 bit mode) before and after each record
- Best practice: write data in whole blocks
Record markers added in unformatted I/O make exchanging data with other programs (notably C ones) troublesome.

`open(unit,...,ACCESS=’stream’,...)` is a new method to access external files.

No record markers are written before or after a `write`

- Thus, advancing or backspacing over records is not possible.
- But required position may be specified by:
  ```fortran
  write(unit,POS=position) x
  read(unit,POS=position) y
  ```

Best practice: if you are really serious about data exchanges, across different programs and systems, use libraries like HDF5, VTK, CGNS.
Array Syntax

Input/Output
  Formatted I/O
  File I/O
  Namelist
  Internal Files
  Unformatted I/O
  Robust I/O
I/O Errors and Mishaps

- You may happen to:
  - Try to open a new file, when one with same name already exists
  - Look for an existing file, which is missing
  - Encounter an unexpected end of record in a read
  - Encounter an unexpected end of file while reading
  - Run out of disk space while writing
  - Try writing to a read-only file
  - ...

...
You may happen to:

- Try to open a new file, when one with same name already exists
- Look for an existing file, which is missing
- Encounter an unexpected end of record in a read
- Encounter an unexpected end of file while reading
- Run out of disk space while writing
- Try writing to a read-only file
- ...

And get an unfriendly runtime error
I/O Errors and Mishaps

- You may happen to:
  - Try to open a new file, when one with same name already exists
  - Look for an existing file, which is missing
  - Encounter an unexpected end of record in a read
  - Encounter an unexpected end of file while reading
  - Run out of disk space while writing
  - Try writing to a read-only file
  - ...

- And get an unfriendly runtime error

- Or you may need to open a file in a library you are writing
  - And use a unit already opened in a calling program
  - The previously opened unit is automatically closed
  - With surprising consequences on program behavior
Managing I/O Errors

- All I/O statements accept an `IOSTAT=ios` option
  - `ios` must be an integer variable of default kind
  - Set to zero on success
  - Set to negative values on end of file or record
    (in Fortran 2003, `iostat_end` and `iostat_eor` respectively, from `iso_fortran_env` module)
  - Set to positive values on error
  - Execution will not stop
- Use it to identify the issue, and recover or fail gracefully
Managing I/O Errors

- All I/O statements accept an `IOMAT=ios` option
  - `ios` must be an integer variable of default kind
  - Set to zero on success
  - Set to negative values on end of file or record
    (in Fortran 2003, `iostat_end` and `iostat_eor` respectively, from `iso_fortran_env` module)
  - Set to positive values on error
  - Execution will not stop
- Use it to identify the issue, and recover or fail gracefully

- All I/O statements accept an `ERR=err-label` option
  - `err-label` is a statement label in the same program unit
  - Flow control jumps to `err-label` in case of error
- Use it to centralize error management and recovery
- Together with `iostat`, of course
Let’s assume \texttt{ans} is a logical variable, \texttt{k} is an integer variable, and \texttt{s} is a character variable of suitable length.
In Doubt? inquire!

- Let’s assume `ans` is a logical variable, `k` is an integer variable, and `s` is a character variable of suitable length.

- `inquire(FILE='input.dat',EXIST=ans)` will set `ans` to `.true.` if file `input.dat` exists.

- `inquire(FILE='input.dat',OPENED=ans)` will set `ans` to `.true.` if file `input.dat` is already opened.

- `inquire(15,OPENED=ans)` will set `ans` to `.true.` if a file is already opened on unit 15.

- `inquire(FILE='input.dat',NUMBER=k)` will set `k` to `-1` if file `input.dat` is not opened, to connected unit otherwise.
Let’s assume \texttt{ans} is a logical variable, \texttt{k} is an integer variable, and \texttt{s} is a character variable of suitable length

- \texttt{inquire(FILE='input.dat',EXIST=ans)} will set \texttt{ans} to \texttt{.true.} if file \texttt{input.dat} exists
- \texttt{inquire(FILE='input.dat',OPENED=ans)} will set \texttt{ans} to \texttt{.true.} if file \texttt{input.dat} is already opened
Let’s assume \texttt{ans} is a logical variable, \texttt{k} is an integer variable, and \texttt{s} is a character variable of suitable length.

- \texttt{inquire(FILE=’input.dat’,EXIST=ans)} will set \texttt{ans} to \texttt{.true.} if file \texttt{input.dat} exists.
- \texttt{inquire(FILE=’input.dat’,OPENED=ans)} will set \texttt{ans} to \texttt{.true.} if file \texttt{input.dat} is already opened.
- \texttt{inquire(15,OPENED=ans)} will set \texttt{ans} to \texttt{.true.} if a file is already opened on unit 15.
Let’s assume \texttt{ans} is a logical variable, \texttt{k} is an integer variable, and \texttt{s} is a character variable of suitable length.

- \texttt{inquire(FILE='input.dat',EXIST=ans)} will set \texttt{ans} to \texttt{.true.} if file \texttt{input.dat} exists
- \texttt{inquire(FILE='input.dat',OPENED=ans)} will set \texttt{ans} to \texttt{.true.} if file \texttt{input.dat} is already opened
- \texttt{inquire(15,OPENED=ans)} will set \texttt{ans} to \texttt{.true.} if a file is already opened on unit 15
- \texttt{inquire(FILE='input.dat',NUMBER=k)} will set \texttt{k} to \texttt{-1} if file \texttt{input.dat} is not opened, to connected unit otherwise.
More Doubts? \textit{inquire} More!

\textbf{\texttt{inquire}(15,\texttt{FORM}=s)} will set \texttt{s} to \texttt{’FORMATTED’} or \texttt{’UNFORMATTED’} if unit 15 is connected for formatted or unformatted I/O respectively, to \texttt{’UNDEFINED’} otherwise.
More Doubts? inquire More!

- **inquire(15, FORM=s)** will set `s` to ‘FORMATTED’ or ‘UNFORMATTED’ if unit 15 is connected for formatted or unformatted I/O respectively, to ‘UNDEFINED’ otherwise.

- **inquire(15, ACTION=s)** will set `s` to ‘READ’ or ‘WRITE’ or ‘READWRITE’, depending on what actions are allowed on unit 15, to ‘UNDEFINED’ if unconnected.
More Doubts? inquire More!

- **inquire(15, FORM=s)** will set s to ‘FORMATTED’ or ‘UNFORMATTED’ if unit 15 is connected for formatted or unformatted I/O respectively, to ‘UNDEFINED’ otherwise.

- **inquire(15, ACTION=s)** will set s to ‘READ’ or ‘WRITE’ or ‘READWRITE’, depending on what actions are allowed on unit 15, to ‘UNDEFINED’ if unconnected.

- **inquire(IOLENGTH=k)** `output-list` will set k to the number of processor dependent units (bytes, in practice) occupied by an unformatted write of `output-list`
More Doubts? *inquire* More!

- `inquire(15, FORM=s)` will set `s` to ‘FORMATTED’ or ‘UNFORMATTED’ if unit 15 is connected for formatted or unformatted I/O respectively, to ‘UNDEFINED’ otherwise.

- `inquire(15, ACTION=s)` will set `s` to ‘READ’ or ‘WRITE’ or ‘READWRITE’, depending on what actions are allowed on unit 15, to ‘UNDEFINED’ if unconnected.

- `inquire(IOLENGTH=k)` *output-list* will set `k` to the number of processor dependent units (bytes, in practice) occupied by an unformatted write of *output-list*.

- And many more variations, look to manuals.
More Doubts? inquire More!

- **inquire**(15, FORM=s) will set s to 'FORMATTED' or 'UNFORMATTED' if unit 15 is connected for formatted or unformatted I/O respectively, to 'UNDEFINED' otherwise

- **inquire**(15, ACTION=s) will set s to 'READ' or 'WRITE' or 'READWRITE', depending on what actions are allowed on unit 15, to 'UNDEFINED' if unconnected

- **inquire**(IOLength=k) output-list will set k to the number of processor dependent units (bytes, in practice) occupied by an unformatted write of output-list

- And many more variations, look to manuals

- Of course, IOSTAT and ERR can be useful on inquire too
Hands-on Session #5

- Write a program that:
  - reads an ‘arbitrarily’ long column of real numbers from an ASCII file
  - prints maximum, minimum, average of the numbers
  - and prints the \([n/2]\)-th row where \(n\) is the length of the column
Part IV

Derived Types and Memory Management

Derived types, operators overloading, parametric types and inheritance. Memory management, dynamic allocation and memory heap. Pointers. C and Fortran interoperability.
Outline

Extending the Language
Derived Types
Operators Overloading
Parameterized Types
Extending Types, and Objects

Managing Memory

Conclusions
Outline

Extending the Language
Derived Types
  Operators Overloading
Parameterized Types
Extending Types, and Objects

Managing Memory

Conclusions
User Defined Types

- Fortran allows programmers to add new types, built as assemblies of existing ones

```fortran
type position
  real :: x, y, z
end type position

type velocity
  real :: x, y, z
end type velocity
```

- Components in different derived types may have the same name (not a surprise!)

- `type(position) :: r` declares a variable of type `position`

- Components of a derived type can be accessed like this: `r%y = 0.0`
Growing Types from Types

- Derived types are not second class citizens
- Thus derived types (also termed *structures*) can be assembled from other derived types too

```fortran
type particle
  type(position) :: r
  type(velocity) :: v
  real :: mass
end type particle

type atom
  type(position) :: r
  type(velocity) :: v
  real :: mass ! In atomic units
  integer :: an ! Atomic number
end type atom
```

- `type(particle) :: p` declares a variable of type `particle`
- Components of a component of a variable can be accessed like this: `p%v%z = 0.0`
type(atom) :: h1, h2, he

h1%r = position(0.0, 0.0, 0.0)
h1%v = velocity(1.0, -1.0, 0.0)
h1%mass = 1.00794
h1%an = 1 ! Assigns atomic number

h2 = h1 ! Intrinsic assignment

he = atom(position(1.0, 0.0, -1.0), h2%v, 4.002602, 2)

- Derived type name can be used to construct values of the type
- Unsurprisingly, \texttt{velocity()} is termed a \textit{constructor}
- Values passed as argument to constructors must be ordered as in type definition
- Assignment is intrinsically available
Formatted I/O of Derived Types

- Derived types boil down (possibly recursively) to collections of intrinsic types
- And behavior is coherent with I/O of complex values and arrays
- All single intrinsic type (sub)components will be processed in sequence
- If you want control of the conversion:
  - a proper edit descriptor must be provided for each component
  - in same order as components are declared in type declaration
- Fortran 2003 introduces the DT edit descriptor to give users total control
Extending the Language
  Derived Types
  Operators Overloading
  Parameterized Types
  Extending Types, and Objects

Managing Memory

Conclusions
Same Name, Different Personality

- Binary operator + can be used to add:
  - a pair of integer values
  - a pair of real values
  - a pair of complex values
  - two integer values of different kinds
  - two real values of different kinds
  - two complex values of different kinds
  - an integer and a real value
  - an integer and a complex value
  - a real and a complex value

- It's like the meaning of + is 'overloaded'

- Different machine code is generated depending on operand types

- And ditto for -, \*, /, >, >=, ...
Same Name, Different Personality

- Binary operator + can be used to add:
  - a pair of integer values
Binary operator + can be used to add:
- a pair of integer values
- a pair of real values
Same Name, Different Personality

- Binary operator + can be used to add:
  - a pair of integer values
  - a pair of real values
  - a pair of complex values

It's like the meaning of + is 'overloaded'
Different machine code is generated depending on operand types
And ditto for -, *, /, >=, ...
Same Name, Different Personality

- Binary operator + can be used to add:
  - a pair of integer values
  - a pair of real values
  - a pair of complex values
  - two integer values of different kinds

It's like the meaning of + is 'overloaded'

Different machine code is generated depending on operand types

And ditto for -, *, /, \geq, ...
Binary operator + can be used to add:

- a pair of integer values
- a pair of real values
- a pair of complex values
- two integer values of different kinds
- two real values of different kinds
Same Name, Different Personality

- Binary operator + can be used to add:
  - a pair of integer values
  - a pair of real values
  - a pair of complex values
  - two integer values of different kinds
  - two real values of different kinds
  - two complex values of different kinds
Same Name, Different Personality

- Binary operator + can be used to add:
  - a pair of integer values
  - a pair of real values
  - a pair of complex values
  - two integer values of different kinds
  - two real values of different kinds
  - two complex values of different kinds
  - an integer and a real value
  - an integer and a complex value
  - a real and a complex value

It's like the meaning of + is 'overloaded'

Different machine code is generated depending on operand types

And ditto for -, *, /, >=, ...
Binary operator + can be used to add:

- a pair of integer values
- a pair of real values
- a pair of complex values
- two integer values of different kinds
- two real values of different kinds
- two complex values of different kinds
- an integer and a real value
- an integer and a complex value

It's like the meaning of + is 'overloaded'. Different machine code is generated depending on operand types. And ditto for -,*,/ etc.
Binary operator + can be used to add:

- a pair of integer values
- a pair of real values
- a pair of complex values
- two integer values of different kinds
- two real values of different kinds
- two complex values of different kinds
- an integer and a real value
- an integer and a complex value
- a real and a complex value
Binary operator + can be used to add:

- a pair of integer values
- a pair of real values
- a pair of complex values
- two integer values of different kinds
- two real values of different kinds
- two complex values of different kinds
- an integer and a real value
- an integer and a complex value
- a real and a complex value

It’s like the meaning of + is ‘overloaded’
Same Name, Different Personality

- Binary operator + can be used to add:
  - a pair of integer values
  - a pair of real values
  - a pair of complex values
  - two integer values of different kinds
  - two real values of different kinds
  - two complex values of different kinds
  - an integer and a real value
  - an integer and a complex value
  - a real and a complex value

- It’s like the meaning of + is ‘overloaded’
  - Different machine code is generated depending on operand types
Binary operator + can be used to add:

- a pair of integer values
- a pair of real values
- a pair of complex values
- two integer values of different kinds
- two real values of different kinds
- two complex values of different kinds
- an integer and a real value
- an integer and a complex value
- a real and a complex value

It’s like the meaning of + is ‘overloaded’
- Different machine code is generated depending on operand types
- And ditto for -, *, /, >, >=, ...
Bringing Abstractions Further

Wouldn’t it be nice to have arithmetic operators work on structures?

interface operator(-)
  function subvel(p1, p2)
    type(velocity), intent(in) :: p1, p2
    type(velocity) :: subvel
  end function
end interface operator(-)

interface operator(-)
  function chsvel(p)
    type(velocity), intent(in) :: p
    type(velocity) :: chsvel
  end function
end interface operator(-)

function subvel(p1, p2)
  implicit none
  type(velocity), intent(in) :: p1, p2
  type(velocity) :: subvel
  subvel%x = p1%x-p2%x; subvel%y = p1%y-p2%y; subvel%z = p1%z-p2%z
end function subvel

function chsvel(p)
  implicit none
  type(velocity), intent(in) :: p
  type(velocity) :: chsvel
  chsvel%x = -p%x; chsvel%y = -p%y; chsvel%z = -p%z
end function chsvel
Changing Rules as We Need

- We are fitting an infinite space into a finite box with periodic boundary conditions
- Wouldn’t it be nice to define our operators with custom functionality?

```plaintext
interface operator(+)
  function addpos(p1, p2)
    type(position), intent(in) :: p1, p2
    type(position) :: addpos
  end function
end interface operator(+)

function addpos(p1, p2) ! Adds positions with periodic boundary conditions
  implicit none
  type(position), intent(in) :: p1, p2
  type(position) :: addpos
  real, parameter :: boxwidth = 128.0

  addpos%x = modulo(p1%x+p2%x, boxwidth)
  addpos%y = modulo(p1%y+p2%y, boxwidth)
  addpos%z = modulo(p1%z+p2%z, boxwidth)
end function addpos
```


Operator Overloading

- **interface operator** *(op-name)* lets you overload *(op-name)* with a generic procedure
  - Arguments must be *intent*(in) and can be either one or two
  - *(op-name)* may be an intrinsic operator, or a *new_name*.
Operator Overloading

- `interface operator(op-name)` lets you overload `op-name` with a generic procedure
  - Arguments must be `intent(in)` and can be either one or two
  - `op-name` may be an intrinsic operator, or a `.new_name`.

- Precedence:
  - same for existing operators
  - highest for new unary operators
  - lowest for new binary operators

- Now velocities may be added as intrinsic arithmetic types
- And defining subtraction is an easy job
- Positions may be added as usual intrinsic variables and boundary conditions are automatically imposed
- Time for a module
Operator Overloading

- **interface operator**(op-name) lets you overload op-name with a generic procedure
  - Arguments must be **intent**(in) and can be either one or two
  - op-name may be an intrinsic operator, or a .new_name.

- Precedence:
  - same for existing operators
  - highest for new unary operators
  - lowest for new binary operators

- Now velocities may be added as intrinsic arithmetic types
- And defining subtraction is an easy job
- Positions may be added as usual intrinsic variables and boundary conditions are automatically imposed
Operator Overloading

- **interface operator**(op-name) lets you overload op-name with a generic procedure
  - Arguments must be **intent**(in) and can be either one or two
  - op-name may be an intrinsic operator, or a .new_name.

- Precedence:
  - same for existing operators
  - highest for new unary operators
  - lowest for new binary operators

- Now velocities may be added as intrinsic arithmetic types
- And defining subtraction is an easy job
- Positions may be added as usual intrinsic variables and boundary conditions are automatically imposed

- Time for a module
module periodic_box
    implicit none
    real, private, parameter :: boxwidth = 128.0
    private addpos, addvel, chsvel, subvel, subpos

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

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

    interface operator(+)
        module procedure addpos
        module procedure addvel
    end interface operator(+)

    contains
    function addpos(p1, p2) ! Adds positions with periodic boundary conditions on x
        type(position), intent(in) :: p1, p2
        type(position) :: addpos
        addpos%x = modulo(p1%x+p2%x, boxwidth)
        addpos%y = modulo(p1%y+p2%y, boxwidth)
        addpos%z = modulo(p1%z+p2%z, boxwidth)
    end function addpos

    function addvel
        !...
    end function addvel

    end module periodic_box
Again, modules are the best way of grouping related stuff

Again, with modules and module procedures we don’t need to write interface blocks

Modules let us hide implementation details

Best practice: put structure definitions and related functions and operators in modules

Anyway, they will be used together

When dealing with nested types with many related functions, a hierarchy of modules would probably help

Because, of course, you can use modules in a module
Hands-on Session #1

- Write a module that defines:
  - A new type `vector` made up of three real components
  - Operator `.cross.` for cross product
  - Operator `+` to sum two `vectors`

- Write a program to test your module

```
program test_class_vector
  use class_vector

  implicit none

  type(vector) :: v, w, z

  v=vector(1.d0,0.d0,0.d0)
  w=vector(0.d0,1.d0,0.d0)
  z=vector(0.d0,0.d0,1.d0)

  write(*,*) v+w.cross.z

end program test_class_vector
```

- Definition of cross product:

\[
a \times b = (a_2 b_3 - a_3 b_2)\hat{i} + (a_3 b_1 - a_1 b_3)\hat{j} + (a_1 b_2 - a_2 b_1)\hat{k}
\]

- Then extend operators to have them work with array of vectors: it’s elementary!
Extending the Language
   Derived Types
   Operators Overloading
   Parameterized Types
   Extending Types, and Objects

Managing Memory

Conclusions
What if we wanted different kinds of points?

This is a possibility:

```fortran
type point
    real( selected_real_kind(5) ) :: x, y, z
end type point

type widepoint
    real( selected_real_kind(12) ) :: x, y, z
end type widepoint
```

But not very elegant, nor easy to manage
In Fortran 2003, types may have kind type parameters:

```fortran
type point(point_kind)
    integer, kind :: point_kind = kind(0.0)
    real(point_kind) :: x, y, z
end type point

type(point(point_kind=kind(0.0))) :: apoint

type(point) :: anotherpoint

type(point(selected_real_kind(12))) :: awiderpoint
```

- **kind** states that this type parameter behaves as a kind
- And it works as **kind** does for intrinsic types
More Derived Type Parameters

- Structures may have array components
  
  ```fortran
  type segments(point_kind)
      integer, kind :: point_kind = kind(0.0)
      type(point(point_kind)), dimension(100) :: start_point
      type(point(point_kind)), dimension(100) :: end_point
  end type segments
  ```

- Our `segments` type looks a bit rigid, doesn’t it?

- Derived type parameters come to rescue:
  
  ```fortran
  type segments(point_kind, n)
      integer, kind :: point_kind = kind(0.0)
      integer, len :: n
      type(point(point_kind)), dimension(n) :: start_point
      type(point(point_kind)), dimension(n) :: end_point
  end type segments
  ```

  ```fortran
  type(segments(n=100)) :: ahundredsegments
  type(segments(n=1000)) :: athousandsegments
  ```
Outline

Extending the Language
  Derived Types
  Operators Overloading
  Parameterized Types

Extending Types, and Objects

Managing Memory

Conclusions
Objects

- So, we are able to define new types, and specialized procedures and operators to use them.
So, we are able to define new types, and specialized procedures and operators to use them.

This is what Computer Science priests term *Object-Based* programming.
Objects

- So, we are able to define new types, and specialized procedures and operators to use them
- This is what Computer Science priests term *Object-Based* programming
- But *point*, *position*, and *velocity* have the same components
  - And that’s always true, whatever the space dimensions
  - But they are conceptually (and dimensionally!) different things
Objects

- So, we are able to define new types, and specialized procedures and operators to use them.
- This is what Computer Science priests term *Object-Based* programming.
- But *point*, *position*, and *velocity* have the same components:
  - And that’s always true, whatever the space dimensions.
  - But they are conceptually (and dimensionally!) different things.
- And *particle*, and *atom* share identical components:
  - And a *ion* would simply add a *charge* component.
So, we are able to define new types, and specialized procedures and operators to use them

This is what Computer Science priests term *Object-Based* programming

But *point, position, and velocity* have the same components

- And that’s always true, whatever the space dimensions
- But they are conceptually (and dimensionally!) different things

And *particle, and atom* share identical components

- And a *ion* would simply add a *charge* component

Wouldn’t it be nice to ‘inherit’ from one type to another?

- Yeah, and easier to manage, too!
- And this is what CS priests call *Object-Oriented* programming, and is so trendy!
Fortran 2003 *extends* Derived Types

```fortran
type point
  real :: x, y, z
end type point

type, extends(point) :: position
end type position

type, extends(point) :: velocity
end type velocity

type particle
  type(position) :: r
  type(velocity) :: v
  real :: mass
end type particle

type, extends(particle) :: atom
  integer :: an ! atomic number
end type atom

type, extends(atom) :: ion
  integer :: charge ! in units of elementary charge
end type ion
```

- *extends* means that the new type has the same components, and possibly more
- Now we still have to write procedures and operators, don’t we?
Handling inheritance

▶ A type extension includes an implicit component with the same name and type as its parent type
  ▶ this can come in handy when the programmer wants to operate on components specific to a parent type

```fortran
  type(ion) :: p ! declare p as a ion object
  p%mass ! access mass component for p
  p%atom%mass ! another way
  p%atom%particle%mass ! ...
```

▶ We often say the child and parent types have a “is a” relationship
  ▶ an atom “is” a particle
  ▶ but a particle is not an atom because the atomic component may be found in atom but not in particle
Consider the case you have to evolve the position of a particle according to a given velocity field

- atoms or ions may behave in the (nearly) same way wrt this evolution
- and you do not want to write two (nearly) identical procedures for the two types

Polymorphic procedures are the right way

- i.e., procedures which can take one or more polymorphic variables as arguments
- “polymorphic variable” = variable whose data type is dynamic at runtime
- the *class* keyword allows F2003 programmers to create polymorphic variables
- use it for dummy arguments (the simplest usage, not the only one)
subroutine setMass(p, m)
  class(particle) :: p
  real, intent(in) :: m
  p%mass = m
end subroutine setMass

- The \( p \) dummy argument is polymorphic, based on the usage of \texttt{class(particle)}
- The subroutine can operate on objects that satisfy the "is a" particle relationship
  - \texttt{setMass} can be called passing a particle, atom, ion, or any future type extension of particle

\begin{verbatim}
type(particle) :: pa ! declare an instance of particle
type(atom)    :: at ! declare an instance of atom
type(ion)     :: io ! declare an instance of ion

  call setMass(pa, mm) ! set the mass for a particle
  call setMass(at, mm) ! set the mass for an atom
  call setMass(io, mm) ! set the mass for a ion
\end{verbatim}
Selecting type

- By default, only those components found in the declared type of an object are accessible
Selecting type

- By default, only those components found in the declared type of an object are accessible
  - e.g., only mass, r, v are accessible for p declared as class(particle)

- To access the components of the dynamic type, the select type construct is required
  - and optional arguments come in handy

- There are two styles of type checks that we can perform
  - type is: satisfied if the dynamic type of the object is the same as the type specified in parentheses
  - class is: satisfied if the dynamic type of the object is the same or an extension of the specified type in parentheses

- Best practice: add a class default branch and print error when p is not an extension of particle type
  - an empty type is (particle) branch may be required to avoid getting error when p is only a particle
By default, only those components found in the declared type of an object are accessible

- e.g., only mass, r, v are accessible for p declared as `class(particle)`

To access the components of the dynamic type, the `select type` construct is required
Selecting type

- By default, only those components found in the declared type of an object are accessible
  - e.g., only mass, r, v are accessible for p declared as `class(particle)`
- To access the components of the dynamic type, the `select type` construct is required
  - and optional arguments come in handy
subroutine initialize(p, mm, rr, vv, aan, ccharge)
class(particle) :: p
real :: mm
type(position) :: rr
type(velocity) :: vv
integer, optional :: aan, ccharge
p%mass = mm
p%r = rr
p%v = vv
select type (p)
type is (particle)
   ! no further initialization required
class is (atom)
   ! atom or ion specific initializations
   if (present(aan)) then
      p%an = aan
   else
      p%an = 1
   endif
class is (ion)
   if (present(ccharge)) then
      p%charge = ccharge
   else
      p%charge = 0
   endif
class default
   ! give error for unexpected/unsupported type
   stop 'initialize: unexpected type for p object!'
end select
end subroutine initialize
By default, only those components found in the declared type of an object are accessible

- e.g., only mass, r, v are accessible for p declared as `class(particle)`

To access the components of the dynamic type, the `select type` construct is required

- and optional arguments come in handy

There are two styles of type checks that we can perform

- type is satisfied if the dynamic type of the object is the same as the type specified in parentheses
- class is satisfied if the dynamic type of the object is the same or an extension of the specified type in parentheses

Best practice: add a `class default` branch and print error when p is not an extension of `particle`
Selecting type

- By default, only those components found in the declared type of an object are accessible
  - e.g., only mass, r, v are accessible for p declared as class(particle)
- To access the components of the dynamic type, the select type construct is required
  - and optional arguments come in handy
- There are two styles of type checks that we can perform
  - type is: satisfied if the dynamic type of the object is the same as the type specified in parentheses
Selecting type

- By default, only those components found in the declared type of an object are accessible
  - e.g., only mass, r, v are accessible for p declared as `class(particle)`
- To access the components of the dynamic type, the `select type` construct is required
  - and optional arguments come in handy
- There are two styles of type checks that we can perform
  - `type is`: satisfied if the dynamic type of the object is the same as the type specified in parentheses
  - `class is`: satisfied if the dynamic type of the object is the same or an extension of the specified type in parentheses
subroutine initialize(p, mm, rr, vv, aan, ccharge)
  class(particle) :: p
  real :: mm
  type(position) :: rr
  type(velocity) :: vv
  integer, optional :: aan, ccharge
  p%mass = mm
  p%r = rr
  p%v = vv
  select type (p)
    type is (particle)
      ! no further initialization required
    class is (atom)
      ! atom or ion specific initializations
      if (present(aan)) then
        p%an = aan
      else
        p%an = 1
      endif
    class is (ion)
      if (present(ccharge)) then
        p%charge = ccharge
      else
        p%charge = 0
      endif
    class default
      ! give error for unexpected/unsupported type
      stop 'initialize: unexpected type for p object,'
  end select
end subroutine initialize
Selecting type

- By default, only those components found in the declared type of an object are accessible
  - e.g., only mass, \( r \), \( v \) are accessible for \( p \) declared as `class(particle)`
- To access the components of the dynamic type, the `select type` construct is required
  - and optional arguments come in handy
- There are two styles of type checks that we can perform
  - `type is`: satisfied if the dynamic type of the object is the same as the type specified in parentheses
  - `class is`: satisfied if the dynamic type of the object is the same or an extension of the specified type in parentheses
- Best practice: add a `class default` branch and print error when \( p \) is not an extension of `particle` type
Selecting type

- By default, only those components found in the declared type of an object are accessible
  - e.g., only mass, r, v are accessible for p declared as class(particle)
- To access the components of the dynamic type, the select type construct is required
  - and optional arguments come in handy
- There are two styles of type checks that we can perform
  - type is: satisfied if the dynamic type of the object is the same as the type specified in parentheses
  - class is: satisfied if the dynamic type of the object is the same or an extension of the specified type in parentheses
- Best practice: add a class default branch and print error when p is not an extension of particle type
  - an empty type is (particle) branch may be required to avoid getting error when p is only a particle
subroutine initialize(p, mm, rr, vv, aan, ccharge)
class(particle) :: p
real :: mm
type(position) :: rr
type(velocity) :: vv
integer, optional :: aan, ccharge
p%m = mm
p%r = rr
p%v = vv
select type (p)
type is (particle)
  ! no further initialization required
class is (atom)
  ! atom or ion specific initializations
  if (present(aan)) then
    p%an = aan
  else
    p%an = 1
  endif
class is (ion)
  if (present(ccharge)) then
    p%charge = ccharge
  else
    p%charge = 0
  endif
class default
  ! give error for unexpected/unsupported type
  stop 'initialize: unexpected type for p object!'
end select
end subroutine initialize
Type-bound procedures

Objects in Fortran 2003

- A Fortran 90/95 module can be viewed as an object because it can encapsulate both data and procedures
- But, derived types in F2003 are considered objects because they now can encapsulate data as well as procedures
- Modules and types work together...

Procedures encapsulated in a derived type are called type-bound procedures ("methods" in OO jargon)

type particle
  type(position) :: r
  type(velocity) :: v
  real :: mass
contains
  procedure :: initialize => initialize_particle
end type particle

initialize_particle is the name of the underlying procedure to be implemented

Explicit interface is required: wrap in a module!
Employing modules and types to design objects

```fortran
module particle_mod
  type particle
    type(position) :: r
    type(velocity) :: v
    real :: mass
  contains
    procedure :: initialize => initialize_particle
  end type particle
  type, extends(particle) :: atom
    ...
  end type atom
  type, extends(atom) :: ion
    ...
  end type ion
  contains
    ! insert the implementation or at least the interface of initialize_particle
    subroutine initialize_particle(p, mm, rr, vv, aan, ccharge)
      class(particle) :: p
    ...
  end subroutine initialize_particle
end module particle_mod
```
Using **class**

**initialize** is the name to be used to invoke the type bound procedure

```fortran
use particle_mod

type(particle) :: p ! declare an instance of particle
call p%initialize(mas, pos, vel) ! initialize particle
```

**What about the first dummy argument of initialize?**

- it is known as the *passed-object* dummy argument
- must be declared **class** and of the same type as the derived type that defined the type-bound procedure
- by default, it is the first dummy argument in the type-bound procedure: it receives the object that invoked the type-bound procedure

**It is possible to pass another argument in place of the first one**

```fortran
procedure, pass(p) :: initialize
```

**...or to avoid passing it at all**

```fortran
procedure, nopass :: initialize
```
Inheritance and TBP

- A child type inherits or reuses components from their parent or ancestor types: this applies to both data and procedures

```fortran
type(particle) :: pa ! declare an instance of particle
type(atom) :: at ! declare an instance of atom
type(ion) :: io ! declare an instance of ion
call pa%initialize(mas, pos, vel) ! initialize a particle
call at%initialize(mas, pos, vel, anu) ! initialize an atom
call io%initialize(mas, pos, vel, anu, cha) ! initialize a ion
```

- `initialize` behaves accordingly to the passed arguments, i.e. using `optional` and `select type` features

- Sometimes, another approach may be more appropriate: overriding!
module particle_mod

  type particle
    type(position) :: r
    type(velocity) :: v
    real :: mass
  contains
    procedure :: initialize => initialize_particle
  end type particle

  type, extends(particle) :: atom
  ...
  contains
    procedure :: initialize => initialize_atom
  end type atom

  type, extends(atom) :: ion
  ...
  end type ion

contains
! insert the implementation or at least the interface of initialize
subroutine initialize_particle(p, mm, rr, vv, aan, cch)
class(particle) :: p
...
end subroutine initialize_particle

subroutine initialize_atom(p, mm, rr, vv, aan, cch)
class(atom) :: p
...
end subroutine initialize_atom

end module particle_mod
Override with care

Beware: an overriding type-bound procedure must have exactly the same interface as the overridden procedure except for the passed-object dummy argument which must be `class(new-type)`

- optional arguments may hide useless arguments

```fortran
type(particle) :: pa
! declare an instance of particle

! declare an instance of atom

type(atom) :: at

type(ion) :: io
! declare an instance of ion

! calls initialize_particle

call pa%initialize(mas, pos, vel)

call at%initialize(mas, pos, vel, anu)
! calls initialize_atom

call io%initialize(mas, pos, vel, anu, cha)
! calls initialize_atom
```
module particle_mod

type particle
    type(position) :: r
    type(velocity) :: v
    real :: mass
contains
    procedure :: initialize => initialize_particle
end type particle

type, extends(particle) :: atom
contains
    procedure :: initialize => initialize_atom
end type atom

type, extends(atom) :: ion
contains
    ! insert the implementation or at least the interface of initialize
end type ion

subroutine initialize_particle(p, mm, rr, vv, aan, cch)
class(particle) :: p
...
end subroutine initialize_particle

subroutine initialize_atom(p, mm, rr, vv, aan, cch)
class(atom) :: p
...
end subroutine initialize_atom
end module particle_mod
Override with care

```fortran

! declare an instance of particle
type(particle) :: pa

! declare an instance of atom
type(atom) :: at

! declare an instance of ion
type(ion) :: io

! calls initialize_particle
call pa%initialize(mas, pos, vel)

! calls initialize_atom
call at%initialize(mas, pos, vel, anu)

call io%initialize(mas, pos, vel, anu, cha)

! calls initialize_atom
```

- Beware: an overriding type-bound procedure must have exactly the same interface as the overridden procedure except for the passed-object dummy argument which must be `class(new-type)`
  - optional arguments may hide useless arguments

- Of course, it is still possible to explicitly invoke the version defined by a parent type instead of the overrided one
Override with care

```
module my_module

  type(particle) :: pa ! declare an instance of particle
  type(atom)    :: at ! declare an instance of atom
  type(ion)     :: io ! declare an instance of ion

  call pa%initialize(mas, pos, vel) ! calls initialize_particle
  call at%initialize(mas, pos, vel, anu) ! calls initialize_atom
  call io%initialize(mas, pos, vel, anu, cha) ! calls initialize_atom

end module my_module
```

- Beware: an overriding type-bound procedure must have exactly the same interface as the overridden procedure except for the passed-object dummy argument which must be `class(new-type)`
  - optional arguments may hide useless arguments
- Of course, it is still possible to explicitly invoke the version defined by a parent type instead of the overridden one
- And it is possible to prevent any type extensions from overriding a particular type-bound procedure

```
procedure, non_overridable :: initialize
```
Information hiding allows the programmer to view an object and its procedures as a “black box”

- procedure overriding is a first example of information hiding, `initialize` has different “hidden” implementations depending on the calling object
Information hiding allows the programmer to view an object and its procedures as a “black box”

- procedure overriding is a first example of information hiding, `initialize` has different “hidden” implementations depending on the calling object

Hiding data:

- safer against data corruption: the user may modify data only through adequate procedures
- changes to the data structure will not affect codes using our class provided that we don’t change interfaces
Information hiding allows the programmer to view an object and its procedures as a “black box”
  - procedure overriding is a first example of information hiding, `initialize` has different “hidden” implementations depending on the calling object

Hiding data:
  - safer against data corruption: the user may modify data only through adequate procedures
  - changes to the data structure will not affect codes using our class provided that we don’t change interfaces

Hiding procedures: e.g., prevent users from calling low-level procedures
Fortran 2003 adds “private” and “public” keywords for derived types

beware of the placement of the keywords, in modules and/or in types: confused?

```fortran
module particle_mod
private ! hide the implementation of type-bound procedures
public :: average_position_particle ! allow access to particle averaging position
type, public :: particle
private ! hide the data underlying details
type(position) :: r
type(velocity) :: v
real :: mass
contains
private ! hide the type bound procedures by default
    procedure :: check_init => check_init_particle ! private type-bound procedure
    procedure, public :: initialize => initialize_particle ! allow access to TBP
end type particle
contains
! implementation of type-bound procedures
subroutine initialize_particle(p, mm, rr, vv, aan, cch)
...
subroutine check_init_particle(p)
...
subroutine average_position_particle(p1,p2)
class(particle) :: p1, p2
...
end subroutine average_position_particle
end module particle_mod
```
Data Polymorphism:

- as how polymorphic dummy arguments form the basis to procedure polymorphism...
- ...polymorphic non-dummy variables form the basis to data polymorphism
Data Polymorphism:
- as how polymorphic dummy arguments form the basis to procedure polymorphism...
- ...polymorphic non-dummy variables form the basis to data polymorphism

Typed allocation
Data Polymorphism:
- as how polymorphic dummy arguments form the basis to procedure polymorphism...
- ...polymorphic non-dummy variables form the basis to data polymorphism

Typed allocation

Unlimited Polymorphic Objects
- you may encounter `class(*)`
Data Polymorphism:
- as how polymorphic dummy arguments form the basis to procedure polymorphism...
- ...polymorphic non-dummy variables form the basis to data polymorphism

Typed allocation

Unlimited Polymorphic Objects
- you may encounter `class(*)`

Generic-type bound procedures
- like generic interfaces, but for type-bound procedures
Data Polymorphism:
- as how polymorphic dummy arguments form the basis to procedure polymorphism...
- ...polymorphic non-dummy variables form the basis to data polymorphism

Typed allocation

Unlimited Polymorphic Objects
- you may encounter `class(*)`

Generic-type bound procedures
- like generic interfaces, but for type-bound procedures

Abstract types and deferred bindings
Data Polymorphism:
- as how polymorphic dummy arguments form the basis to procedure polymorphism...
- ...polymorphic non-dummy variables form the basis to data polymorphism

Typed allocation

Unlimited Polymorphic Objects
- you may encounter `class(*)`

Generic-type bound procedures
- like generic interfaces, but for type-bound procedures

Abstract types and deferred bindings

Finalization
Extending the Language

Managing Memory
  Dynamic Memory Allocation
  Fortran Pointers
  Sketchy Ideas on Data Structures
  Bridging the Gap with C

Conclusions
Let’s imagine we have to solve a PDE
Let’s imagine we have to solve a PDE

On a dense, Cartesian, uniform grid

- Mesh axes are parallel to coordinate ones
- Steps along each direction have the same size
- And we have some discretization schemes in time and space to solve for variables at each point
integer, parameter :: NX = 200
integer, parameter :: NY = 450
integer, parameter :: NZ = 320

integer, parameter :: rk = selected_real_kind(12)

real(rk) :: deltax ! Grid steps
real(rk) :: deltay
real(rk) :: deltaz
real(rk) :: u(NX,NY,NZ)
real(rk) :: v(NX,NY,NZ)
real(rk) :: w(NX,NY,NZ)
real(rk) :: p(NX,NY,NZ)

We could write something like that in a module, and use it everywhere
We could write something like that in a module, and use it everywhere

But it has annoying consequences

- Recompile each time grid resolution changes
- A slow process, for big programs
- And error prone, as we may forget about
A Rigid Solution

We could write something like that in a module, and use it everywhere.

But it has annoying consequences:
- Recompile each time grid resolution changes
- A slow process, for big programs
- And error prone, as we may forget about

Could we size data structures according to user input?
A Recurrent Issue: SoA or AoS

Which one is best?

Both have merits

The choice strongly depends on the computer architecture

for cache-based CPUs the choice is difficult (it depends on the order of the accesses of your numerical scheme)

but using GPUs or MICs the first one is usually better!
A Recurrent Issue: SoA or AoS

- **type flow**
  
  real(rk) :: u(NX,NY,NZ)
  real(rk) :: v(NX,NY,NZ)
  real(rk) :: w(NX,NY,NZ)
  real(rk) :: p(NX,NY,NZ)

  end type

  type(flow) :: f

  Or

- **type flow**
  
  real(rk) :: u,v,w,p

  end type

  type(flow) :: f(NX,NY,NZ)

Which one is best?

- Both have merits
A Recurrent Issue: SoA or AoS

- type flow
  - real(rk) :: u(NX, NY, NZ)
  - real(rk) :: v(NX, NY, NZ)
  - real(rk) :: w(NX, NY, NZ)
  - real(rk) :: p(NX, NY, NZ)

end type

type(flow) :: f

Or

type flow
  - real(rk) :: u,v,w,p
end type

type(flow) :: f(NX, NY, NZ)

Which one is best?

- Both have merits
- The choice strongly depends on the computer architecture
A Recurrent Issue: SoA or AoS

- **type flow**
  - real(rk) :: u(NX,NY,NZ)
  - real(rk) :: v(NX,NY,NZ)
  - real(rk) :: w(NX,NY,NZ)
  - real(rk) :: p(NX,NY,NZ)

end type

- type(flow) :: f

Or

- type flow
  - real(rk) :: u,v,w,p

end type

- type(flow) :: f(NX,NY,NZ)

Which one is best?

- Both have merits
- The choice strongly depends on the computer architecture
  - for cache-based CPUs the choice is difficult (it depends on the order of the accesses of your numerical scheme)
A Recurrent Issue: SoA or AoS

- **type flow**
  - real(rk) :: u(NX, NY, NZ)
  - real(rk) :: v(NX, NY, NZ)
  - real(rk) :: w(NX, NY, NZ)
  - real(rk) :: p(NX, NY, NZ)

- end type

- type(flow) :: f

Or

- type flow
  - real(rk) :: u, v, w, p

- end type

- type(flow) :: f(NX, NY, NZ)

Which one is best?

- Both have merits
- The choice strongly depends on the computer architecture
  - for cache-based CPUs the choice is difficult (it depends on the order of the accesses of your numerical scheme)
  - but using GPUs or MICs the first one is usually better!
Looking for Flexibility

```fortran
subroutine my_pde_solver(nx, ny, nz)
    integer, intent(in) :: nx, ny, nz
    integer, parameter :: rk = selected_real_kind(12)
    real(rk):: deltax, deltay, deltaz ! Grid steps
    real(rk) :: u(nx,ny,nz)
    real(rk) :: v(nx,ny,nz)
    real(rk) :: w(nx,ny,nz)
    real(rk) :: p(nx,ny,nz)

    We could think of declaring automatic arrays inside a subroutine
```
subroutine my_pde_solver(nx, ny, nz)
  integer, intent(in) :: nx, ny, nz

  integer, parameter :: rk = selected_real_kind(12)
  real(rk):: deltax, deltay, deltaz ! Grid steps

  real(rk) :: u(nx,ny,nz)
  real(rk) :: v(nx,ny,nz)
  real(rk) :: w(nx,ny,nz)
  real(rk) :: p(nx,ny,nz)

▶ We could think of declaring automatic arrays inside a subroutine
▶ This is unwise
  ▶ Automatic arrays are usually allocated on the process stack
  ▶ Which is a precious resource
  ▶ And limited in most system configurations
A Bad, Old, Common approach

given for a different shape to dummy arguments

We could give a different shape to dummy arguments
A Bad, Old, Common approach

program pde_solve
    parameter (MAXNX=400, MAXNY=400, MAXNZ=400)
    parameter (MAXSIZE=MAXNX*MAXNX*MAXNZ)

    real*8 u(MAXSIZE), v(MAXSIZE), w(MAXSIZE), p(MAXSIZE)

    common u, v, w, p

    ! ... call my_pde_solver(nx, ny, nz, u, v, w, p)
    ! ... end

subroutine my_pde_solver(nx, ny, nz, u, v, w, p)
    real*8 u(nx, ny, nz), v(nx, ny, nz), w(nx, ny, nz), p(nx, ny, nz)

    ! ...

➤ We could give a different shape to dummy arguments
➤ But this only works if interface is implicit
    ➤ Which is dangerous
A Bad, Old, Common approach

```fortran
program pde_solve
  parameter (MAXNX=400, MAXNY=400, MAXNZ=400)
  parameter (MAXSIZE=MAXNX*MAXNX*MAXNZ)

  real*8 u(MAXSIZE), v(MAXSIZE), w(MAXSIZE), p(MAXSIZE)

  common u, v, w, p
  ! ... 
  call my_pde_solver(nx, ny, nz, u, v, w, p)
  ! ... 
end

subroutine my_pde_solver(nx, ny, nz, u, v, w, p)
  real*8 u(nx, ny, nz), v(nx, ny, nz), w(nx, ny, nz), p(nx, ny, nz)
  ! ...

  ! We could give a different shape to dummy arguments
  ! But this only works if interface is implicit
  ! Which is dangerous

  ! Maximum problem size still program limited: nx*ny*nz must be less than MAXSIZE
```
Removing Limitations

- Being program limited is annoying

- It’s much better to accommodate to any user specified problem size
  - Right, as long as there is enough memory
  - But if memory is not enough, not our fault
  - It’s computer or user’s fault

- And there are many complex kinds of computations
  - Those in which memory need cannot be foreseen in advance
  - Those in which arrays do not fit
  - Those in which very complex data structures are needed
Outline

Extending the Language

Managing Memory

*Dynamic Memory Allocation*

Fortran Pointers

Sketchy Ideas on Data Structures

Bridging the Gap with C

Conclusions
integer, parameter :: rk = selected_real_kind(12)

real(rk), dimension(:, :, :), allocatable :: u, v, w, p

allocate (u(nx, ny, nz), v(nx, ny, nz), w(nx, ny, nz), p(nx, ny, nz))

- When allocatable arrays are declared, only their rank is specified (dimension(:, :, :))
Enter Allocatable Arrays

```fortran
integer, parameter :: rk = selected_real_kind(12)
real(rk), dimension(:, :, :), allocatable :: u, v, w, p
allocate(u(nx, ny, nz), v(nx, ny, nz), w(nx, ny, nz), p(nx, ny, nz))
```

- When allocatable arrays are declared, only their rank is specified (`dimension(:, :, :)`)  
- `allocate` statement performs actual memory allocation and defines extents

▶ Best practice: use `STAT=` and, on failure, provide information to users before terminating execution
integer, parameter :: rk = selected_real_kind(12)

real(rk), dimension(:,:,:,:), allocatable :: u,v,w,p

allocate (u(nx,ny,nz), v(nx,ny,nz), w(nx,ny,nz), p(nx,ny,nz))

- When allocatable arrays are declared, only their rank is specified (\texttt{dimension(:,:,:,:)}).
- \texttt{allocate} statement performs actual memory allocation and defines extents.
  - On failure, program \texttt{stops}.

Best practice: use \texttt{STAT=} and, on failure, provide information to users before terminating execution.
integer, parameter :: rk = selected_real_kind(12)

real(rk), dimension(:,:,:,:), allocatable :: u,v,w,p

allocate(u(nx,ny,nz),v(nx,ny,nz),w(nx,ny,nz),p(nx,ny,nz))

- When allocatable arrays are declared, only their rank is specified (dimension(:,:,:,:))
- allocate statement performs actual memory allocation and defines extents
  - On failure, program stops
  - But if STAT=integer_var is specified, integer_var is set to zero on success and to a positive value on failure, and execution doesn’t stop
Enter Allocatable Arrays

```fortran
integer, parameter :: rk = selected_real_kind(12)

real(rk), dimension(:,:,:,:), allocatable :: u,v,w,p

allocate(u(nx,ny,nz),v(nx,ny,nz),w(nx,ny,nz),p(nx,ny,nz))
```

- When allocatable arrays are declared, only their rank is specified (`dimension(:,:,:,:)`)
- `allocate` statement performs actual memory allocation and defines extents
  - On failure, program `stop`
  - But if `STAT=integer_var` is specified, `integer_var` is set to zero on success and to a positive value on failure, and execution doesn’t stop
- Best practice: use `STAT=` and, on failure, provide information to users before terminating execution
Freeing Memory

- Where all these ‘dynamic allocated memory’ comes from?
  - From an internal area, often termed “memory heap”
  - When that is exhausted, OS is asked to give the process more memory
  - And if OS is short of memory, or some configuration limit is exhausted...
Freeing Memory

Where all these ‘dynamic allocated memory’ comes from?

- From an internal area, often termed “memory heap”
- When that is exhausted, OS is asked to give the process more memory
- And if OS is short of memory, or some configuration limit is exhausted...

When you are done with an allocatable, use `deallocate` to claim memory back

- Allocatable which are local to a procedure are automatically deallocated on return
- But it’s implementation defined what happens to allocatable private to a module
Freeing Memory

- Where all these ‘dynamic allocated memory’ comes from?
  - From an internal area, often termed “memory heap”
  - When that is exhausted, OS is asked to give the process more memory
  - And if OS is short of memory, or some configuration limit is exhausted...

- When you are done with an allocatable, use deallocate to claim memory back
  - Allocatable which are local to a procedure are automatically deallocated on return
  - But it’s implementation defined what happens to allocatable private to a module

- Best practice: always deallocate when you are done with an allocatable array
Three Common Mistakes

▶ Trying to allocate or deallocate an array that was not allocatable

▶ Compiler will catch it

▶ Trying to allocate or deallocate an array that was not deallocated or allocated respectively

▶ Compiler can't catch it, runtime error

▶ In some cases (error recovery) use logical allocated() function to check

▶ Mistaking allocatables for a substitute to procedure automatic arrays

▶ Dynamic allocation incurs costs

▶ Only worth for big arrays that would not fit program stack
Three Common Mistakes

- Trying to allocate or deallocate an array that was not allocatable
  - Compiler will catch it
- Trying to allocate or deallocate an array that was not deallocated or allocated respectively
  - Compiler can’t catch it, runtime error
  - In some cases (error recovery) use logical allocated() function to check
- Mistaking allocatables for a substitute to procedure automatic arrays
  - Dynamic allocation incurs costs
  - Only worth for big arrays that would not fit program stack
Three Common Mistakes

- Trying to allocate or deallocate an array that was not allocatable
  - Compiler will catch it

- Trying to allocate or deallocate an array that was not deallocated or allocated respectively
Three Common Mistakes

- Trying to allocate or deallocate an array that was not allocatable
  - Compiler will catch it

- Trying to allocate or deallocate an array that was not deallocated or allocated respectively
  - Compiler can’t catch it, runtime error
Three Common Mistakes

- Trying to allocate or deallocate an array that was not allocatable
  - Compiler will catch it

- Trying to allocate or deallocate an array that was not deallocated or allocated respectively
  - Compiler can’t catch it, runtime error
  - In some cases (error recovery) use logical \texttt{allocated()} function to check

Dynamic allocation incurs costs
Only worth for big arrays that would not fit program stack
Three Common Mistakes

- Trying to allocate or deallocate an array that was not allocatable
  - Compiler will catch it

- Trying to allocate or deallocate an array that was not deallocated or allocated respectively
  - Compiler can’t catch it, runtime error
  - In some cases (error recovery) use logical allocated() function to check

- Mistaking allocatables for a substitute to procedure automatic arrays
Three Common Mistakes

- Trying to allocate or deallocate an array that was not allocatable
  - Compiler will catch it

- Trying to allocate or deallocate an array that was not deallocated or allocated respectively
  - Compiler can’t catch it, runtime error
  - In some cases (error recovery) use logical `allocated()` function to check

- Mistaking allocatables for a substitute to procedure automatic arrays
  - Dynamic allocation incurs costs
Three Common Mistakes

- Trying to allocate or deallocate an array that was not allocatable
  - Compiler will catch it

- Trying to allocate or deallocate an array that was not deallocated or allocated respectively
  - Compiler can’t catch it, runtime error
  - In some cases (error recovery) use logical `allocated()` function to check

- Mistaking allocatables for a substitute to procedure automatic arrays
  - Dynamic allocation incurs costs
  - Only worth for big arrays that would not fit program stack
When assigning an array value to a not allocated allocatable array, the allocatable array gets automatically allocated.

This simplifies the use of array functions which return a variable-sized result.

```fortran
real, dimension(100) :: x
real, allocatable, dimension(:) :: all_values, nonzero_values

! size is 100, small benefit wrt explicit allocation
all_values = x

! size depends on x values, AA is a great benefit now
nonzero_values = pack(x, x/=0)
```

Also useful when dealing with allocatable components in a derived type.
Automatic allocation (F2003)

- When assigning an array value to a not allocated allocatable array, the allocatable array gets automatically allocated.
- This simplifies the use of array functions which return a variable-sized result.

```fortran
real, dimension(100) :: x
real, allocatable, dimension(::) :: all_values, nonzero_values

! size is 100, small benefit wrt explicit allocation
all_values = x

! size depends on x values, AA is a great benefit now
nonzero_values = pack(x, x/=0)
```

- Also useful when dealing with allocatable components in a derived type:
  - Avoids separate coding for each allocatable component.
Automatic allocation (F2003) / 2

- Automatic re-allocation is performed when the shape of the assignment does not fit, e.g.
  \[ a = (/ a, 5, 6 /) \]

- Beware: it may dramatically affect performances!
  - if you don’t need it, disable it using compiler options

- AA naturally extends to `characters` strongly increasing their adaptability
  - when declaring `characters`, the `len` value declaration may be postponed (deferred type parameter)
  - during assignment the Right Hand Side passes its `len` on the deferred-length string (under the hood, automatic re-allocation may occur)
  - explicit allocation is possible but often worthless, required when reading from input, though

```fortran
character(len=:), allocatable :: str
character(len=50) :: fixed_str
allocate(character(80) :: str) ! allocates str using len=80
str = fixed_str ! re-allocates str using len=50
```
Outline

Extending the Language

Managing Memory
- Dynamic Memory Allocation
- Fortran Pointers
- Sketchy Ideas on Data Structures
- Bridging the Gap with C

Conclusions
Enter Fortran Pointers

- Fortran pointers are aliases to other objects
- Declared like regular variables, with attribute `pointer`
- Associated to actual objects with pointer assignment `=>`
- To be associated with a pointer, variables must have the `target` attribute
  - But compilers are often liberal (sloppy?) on this
- Disassociated by actual objects with `nullify` statement or by pointer assignment of `null()`

```fortran
real, dimension(:, :, :), pointer :: r
real, target :: a(5, 15, 6), b(3, 22, 7)

r => a         ! pointer assignment
! now r is an alias of a
r(1, 1, 1) = 2. ! usual assignment
! now both r(1, 1, 1) and a(1, 1, 1) values are 2.
nullify(r)     ! a is still alive

r => b         ! now r is an alias of b
r => null()```
Let us clarify

real, dimension(:), pointer :: p

does not declare an array of pointers, but a pointer capable of aliasing an array
Let us clarify

```fortran
real, dimension(:), pointer :: p
```

does not declare an array of pointers, but a pointer capable of aliasing an array

What about array of pointers?

```fortran
type row
  real, dimension(:), pointer :: r
end type row

type(row), dimension(n) :: t

do i=1,n
  allocate(t(i)%r(1:i)) ! Allocate row i of length i
enddo
```
Let us clarify

```fortran
real, dimension(:), pointer :: p
```

does not declare an array of pointers, but a pointer capable of aliasing an array

What about array of pointers?

- as such are not allowed in Fortran, but the equivalent effect can be achieved by creating a type containing a pointer component and building an array of this type
Let us clarify

```fortran
real, dimension(:), pointer :: p
```
does not declare an array of pointers, but a pointer capable of aliasing an array

What about array of pointers?

- as such are not allowed in Fortran, but the equivalent effect can be achieved by creating a type containing a pointer component and building an array of this type

For example, a lower-triangular matrix may be held using a pointer for each row

```fortran
type row
  real, dimension(:), pointer :: r
end type row

type(row), dimension(n) :: t

do i=1,n
  allocate(t(i)%r(1:i)) ! Allocate row i of length i
endo`
```
Pointers may also alias subobjects

real, dimension(:,:,:), pointer :: r

```fortran
r => a(2:4,1:10,3:6) ! r(3,10,4) aliases a(4,10,6)
```

type(velocity), pointer :: v

```fortran
v => oneatom%velocity
```

real, target :: a(5,15,6)

type(atom), target :: oneatom

```fortran
r => a(2:4,1:10,3:6) ! r(1,1,1) aliases a(2,1,3)
    ! r(3,10,4) aliases a(4,10,6)
```

```fortran
v => oneatom%velocity
```
More Fortran Pointers

- Pointers may also alias subobjects

```fortran
real, dimension(:,:,:,:), pointer :: r
type(velocity), pointer :: v
real, target :: a(5,15,6)
type(atom), target :: oneatom

r => a(2:4,1:10,3:6) ! r(1,1,1) aliases a(2,1,3)
   ! r(3,10,4) aliases a(4,10,6)
v => oneatom%velocity
```

- The reverse is not true: it is not possible to explicitly associate sections of pointers

```fortran
s(2:,:,:,:) => a(2:4,1:10,3:6) ! s(2,1,1) aliases a(2,1,3)
```
More Fortran Pointers

▶ Pointers may also alias subobjects

```fortran
real, dimension(:,,:,:), pointer :: r
type(velocity), pointer :: v
real, target :: a(5,15,6)
type(atom), target :: oneatom

r => a(2:4,1:10,3:6) ! r(1,1,1) aliases a(2,1,3)
    ! r(3,10,4) aliases a(4,10,6)
v => oneatom%velocity
```

▶ The reverse in not true: it is not possible to explicitly associate sections of pointers

▶ But lower bounds may be specified (from F2003)

```fortran
s(2,:,:,:) => a(2:4,1:10,3:6) ! s(2,1,1) aliases a(2,1,3)
```
Pointers may also alias subobjects

```fortran
real, dimension(:,,:,:), pointer :: r
type(velocity), pointer :: v
real, target :: a(5,15,6)
type(atom), target :: oneatom
```

```fortran
r => a(2:4,1:10,3:6) ! r(1,1,1) aliases a(2,1,3)
    ! r(3,10,4) aliases a(4,10,6)

v => oneatom%velocity
```

The reverse in not true: it is not possible to explicitly associate sections of pointers

But lower bounds may be specified (from F2003)

```fortran
s(2:,:,::) => a(2:4,1:10,3:6) ! s(2,1,1) aliases a(2,1,3)
```

A target of a multidimensional array pointer may be one-dimensional

```fortran
a(1:n,1:n) => a_linear(1:n*n)
```
Allocating Pointers

- If you allocate a pointer, an unnamed object of the pointee type is created, and associated with the pointer itself

```fortran
real, dimension(:, :, :), pointer :: r
type(atom_list), pointer :: first
```

```fortran
allocate(r(5, 15, 6))
! now r refers an unnamed array allocated on the heap
```

```fortran
allocate(first)
! now first refers to an unnamed type(atom_list) variable, ! allocated on the heap
```
Allocating Pointers

▶ If you allocate a pointer, an unnamed object of the pointee type is created, and associated with the pointer itself

```fortran
type(atom_list), pointer :: first
real, dimension(:,:,:), pointer :: r
allocate(r(5,15,6))
! now r refers an unnamed array allocated on the heap
allocate(first)
! now first refers to an unnamed type(atom_list) variable, ! allocated on the heap
```

▶ Unlike `allocatables`, once allocated the pointers may be migrated to other targets

```fortran```
Allocating Pointers

- If you allocate a pointer, an unnamed object of the pointee type is created, and associated with the pointer itself

```fortran
real, dimension(:,:,:,:), pointer :: r
type(atom_list), pointer :: first

allocate(r(5,15,6))
! now r refers an unnamed array allocated on the heap

allocate(first)
! now first refers to an unnamed type(atom_list) variable, ! allocated on the heap
```

- Unlike `allocatables`, once allocated the `pointers` may be migrated to other targets

- You can use pointer allocation in place of allocatable, but, unless necessary, prefer `allocatable`: the compiler usually optimizes better
Allocating Pointers

- If you allocate a pointer, an unnamed object of the pointee type is created, and associated with the pointer itself.

```fortran
real, dimension(:,:,,:), pointer :: r
type(atom_list), pointer :: first

allocate(r(5,15,6))
! now r refers an unnamed array allocated on the heap

allocate(first)
! now first refers to an unnamed type(atom_list) variable, allocated on the heap
```

- Unlike allocatables, once allocated the pointers may be migrated to other targets.

- You can use pointer allocation in place of allocatable, but, unless necessary, prefer allocatable: the compiler usually optimizes better.

- You can deallocate the pointee by specifying the pointer in a deallocate statement.
In alternative to target variables, allocated pointers may be practically used as targets.

```fortran
type(atom_list), pointer :: next
```

which comes in handy to define complex data structures, like lists.
In alternative to target variables, allocated pointers may be practically used as targets.

Structure components can be pointers.
In alternative to target variables, allocated pointers may be practically used as targets.

Structure components can be pointers.

And a pointer in a structure can point to a structure of the same type:

```fortran
  type atom_list
    type(atom) :: a
    type(atom_list), pointer :: next
  end type
```

which comes in handy to define complex data structures, like lists:

```
FIRST      CURRENT
  ↓          ↓
STUFF     STUFF     STUFF     ...............     STUFF
  NEXT     NEXT     NEXT     NEXT
```

null()
BIG Mistakes with Pointers

- Referencing an undefined pointer (strange things may happen, it may also seem to work)
  - Good practice: initialize pointers to `null()`
BIG Mistakes with Pointers

- Referencing an undefined pointer (strange things may happen, it may also seem to work)
  - Good practice: initialize pointers to `null()`

- Referencing a nullified pointer
  - Your program will fail
  - Which is better than messing up with memory
BIG Mistakes with Pointers

- Referencing an undefined pointer (strange things may happen, it may also seem to work)
  - Good practice: initialize pointers to `null()`

- Referencing a nullified pointer
  - Your program will fail
  - Which is better than messing up with memory

- Changing association of an allocated pointer
  - This is a memory leak, and programmers causing memory leaks have really bad reputation
BIG Mistakes with Pointers

- Referencing an undefined pointer (strange things may happen, it may also seem to work)
  - Good practice: initialize pointers to null()

- Referencing a nullified pointer
  - Your program will fail
  - Which is better than messing up with memory

- Changing association of an allocated pointer
  - This is a memory leak, and programmers causing memory leaks have really bad reputation

```fortran
real, dimension(:,:), pointer :: r, p
!...
allocate(r(n,m))
p => r
! ...
deallocate(r)
p(k,l) = p(k,l)+1
```

Now you’ll be in troubles with `p`, with really strange behavior
Laplace Equation

- Discretization on Cartesian 2D grid with Dirichlet Boundary Conditions

\[
\begin{cases}
  f(x_{i+1,j}) + f(x_{i-1,j}) - 2f(x_{i,j}) + \\
  f(x_{i,j+1}) + f(x_{i,j-1}) - 2f(x_{i,j}) = 0 & \forall x_{i,j} \in (a, b)^2 \\
  f(x_{i,j}) = \alpha(x_{i,j}) & \forall x_{i,j} \in \partial[a, b]^2
\end{cases}
\]

- Iterative advancement using Jacobi method

\[
\begin{cases}
  f_{n+1}(x_{i,j}) = \frac{1}{4} \left[ f_n(x_{i+1,j}) + f_n(x_{i-1,j}) + \\
  f_n(x_{i,j+1}) + f_n(x_{i,j-1}) \right] & \forall n > 0 \\
  f_0(x_{i,j}) = 0 & \forall x_{i,j} \in (a, b)^2 \\
  f_n(x_{i,j}) = \alpha(x_{i,j}) & \forall x_{i,j} \in \partial[a, b]^2, \quad \forall n > 0
\end{cases}
\]
program laplace
  implicit none
  integer, parameter :: dp=kind(1.d0), n = 100
  integer :: maxIter = 100000, i, j, iter = 0
  real(dp), dimension(0:n+1,0:n+1) :: T, Tnew
  real(dp) :: tol = 1.d-4, var = 1.d0, top = 100.d0
  T(0:n,0:n) = 0.d0
  T(n+1,1:n) = (/ (i, i=1,n) /) * (top / (n+1))
  T(1:n,n+1) = (/ (i, i=1,n) /) * (top / (n+1))
  do while (var > tol .and. iter <= maxIter)
    iter = iter + 1; var = 0.d0
    do j = 1, n
      do i = 1, n
        Tnew(i,j) = 0.25d0*( T(i-1,j) + T(i+1,j) + &
        T(i,j-1) + T(i,j+1) )
        var = max(var, abs( Tnew(i,j) - T(i,j) ))
      end do
    end do
    if (mod(iter,100)==0) &
      write(*,"(a,i8,e12.4)") ' iter, variation: ', iter, var
    T(1:n,1:n) = Tnew(1:n,1:n)
  end do
end program laplace
Laplace: static implementation
program laplace
implicit none
integer, parameter :: dp=kind(1.d0), n = 100
integer
:: maxIter = 100000, i, j, iter = 0
real(dp), dimension(0:n+1,0:n+1) :: T, Tnew
real(dp)
:: tol = 1.d-4, var = 1.d0, top = 100.d0
T(0:n,0:n) = 0.d0
T(n+1,1:n) = (/ (i, i=1,n) /) * (top / (n+1))
T(1:n,n+1) = (/ (i, i=1,n) /) * (top / (n+1))
do while (var > tol .and. iter <= maxIter)
iter = iter + 1;
var = 0.d0
do j = 1, n
do i = 1, n
Tnew(i,j) = 0.25d0*( T(i-1,j) + T(i+1,j) + &
T(i,j-1) + T(i,j+1) )
var = max(var, abs( Tnew(i,j) - T(i,j) ))
end do
end do
if (mod(iter,100)==0) &
write(*,"(a,i8,e12.4)") ’ iter, variation:’, iter, var
T(1:n,1:n) = Tnew(1:n,1:n)
end do
end program laplace


Hands-on Session #2

- Modify the code using advanced Fortran features:
  - array syntax
  - allocatable arrays
  - pointer arrays

- Try to list pros and cons of each approach
Very basic lists

- Declare two pointers to list elements (typically head and current elements)
- Allocate the head and let the current pointer alias the head, too
- Fill the inner content of the list element
- To add an element to the end allocate the **next** component
- Let the current pointer be associated to this new element

```fortran
  type(atom_list), pointer :: first, current
  allocate(first) ; first%next => null() ; current => first
  current%a = 2
  allocate(current%next)
  current => current%next ; current%next => null()
```

- And if you want to access to an existing list, use **associated**

```fortran
  current => first
  do while (associated(current))
    print*,'List Element: ',current%a
    current => current%next
  end do
```
Write a program that:

- reads an 'arbitrarily' long column of real numbers from an ASCII file
- stores the values in a double-linked list type line_list

```fortran
real :: a
type(line_list), pointer :: next, previous
endtype line_list

type(line_list), pointer :: first => null(), current => null()
allocate(first) ; first%next => null(); first%previous => null()
current => first
```

- Start by declaring the first and current pointers
- Next, allocate and initialize the first pointer
- Current loop over the lines of the file until a invalid read occurs
- For each valid read, add an element to the list and advance...
Write a program that:

- reads an ‘arbitrarily’ long column of real numbers from an ASCII file
Hands-on Session #3

- Write a program that:
  - reads an ‘arbitrarily’ long column of real numbers from an ASCII file
  - store the values in a double-linked list

```plaintext
type line_list
  real :: a
  type(line_list), pointer :: next
  type(line_list), pointer :: previous
endtype line_list
```
Hands-on Session #3

- Write a program that:
  - reads an ‘arbitrarily’ long column of real numbers from an ASCII file
  - store the values in a double-linked list
    ```fortran
    type line_list
      real :: a
      type(line_list), pointer :: next
      type(line_list), pointer :: previous
    endtype line_list
    ```
  - Start by declaring the first and current pointers
    ```fortran
    type(line_list), pointer :: first=>null(), current=>null()
    ```
Write a program that:

- reads an ‘arbitrarily’ long column of real numbers from an ASCII file
- store the values in a double-linked list

```fortran
  type line_list
    real :: a
    type(line_list), pointer :: next
    type(line_list), pointer :: previous
  endtype line_list
```

- Start by declaring the first and current pointers

  ```fortran
  type(line_list), pointer :: first=>null(), current=>null()
  ```

- Next, allocate and initialize the first pointer

  ```fortran
  allocate(first) ; first%next => null(); first%previous => null()
current => first
  ```
Hands-on Session #3

- Write a program that:
  - reads an ‘arbitrarily’ long column of real numbers from an ASCII file
  - store the values in a double-linked list
    ```
    type line_list
        real :: a
        type(line_list), pointer :: next
        type(line_list), pointer :: previous
    endtype line_list
    ```
  - Start by declaring the first and current pointers
    ```
    type(line_list), pointer :: first=>null(), current=>null()
    ```
  - Next, allocate and initialize the `first` pointer
    ```
    allocate(first) ; first%next => null(); first%previous => null()
    current => first
    ```
  - Then loop over the lines of the file until a invalid read occurs
Write a program that:

- reads an `arbitrarily` long column of real numbers from an ASCII file
- store the values in a double-linked list

```fortran
  type line_list
    real :: a
    type(line_list), pointer :: next
    type(line_list), pointer :: previous
  endtype line_list
```

- Start by declaring the first and current pointers

```fortran
  type(line_list), pointer :: first=>null(), current=>null()
```

- Next, allocate and initialize the `first` pointer

```fortran
  allocate(first) ; first%next => null(); first%previous => null()
current => first
```

- Then loop over the lines of the file until a invalid read occurs
- For each valid read, add an element to the list and advance...
Outline

Extending the Language

Managing Memory
- Dynamic Memory Allocation
- Fortran Pointers
- Sketchy Ideas on Data Structures
- Bridging the Gap with C

Conclusions
Nonuniform Grids

Let’s imagine we have to solve a PDE
Let’s imagine we have to solve a PDE
On a dense, Cartesian, non uniform grid
  - Mesh axes are parallel to coordinate ones
  - Steps along each direction differ in size from point to point
Keeping Information Together

```fortran
! Related information is best kept together
```
```fortran
type nonuniform_grid
  integer :: nx, ny, nz
  ! Grid steps
  real(rk), dimension(:), allocatable :: deltax
  real(rk), dimension(:), allocatable :: deltay
  real(rk), dimension(:), allocatable :: deltaz
end type

! type(nonuniform_grid) :: my_grid
integer :: alloc_stat

allocate(my_grid%deltax(nx),my_grid%deltay(ny), &
         my_grid%deltaz(nz), STAT=alloc_stat)
if (alloc_stat > 0) then
  ! graceful failure
end if
```
Keeping Information Together

```fortran
! Keeping related information together

type nonuniform_grid
    integer :: nx, ny, nz
    ! Grid steps
    real(rk), dimension(:,), allocatable :: deltax
    real(rk), dimension(:,), allocatable :: deltay
    real(rk), dimension(:,), allocatable :: deltaz
end type

!...
type(nonuniform_grid) :: my_grid
integer :: alloc_stat
!...
allocate(my_grid%deltax(nx),my_grid%deltay(ny), &
        my_grid%deltaz(nz), STAT=alloc_stat)
if (alloc_stat > 0) then
    ! graceful failure
end if
```

- Related information is best kept together
- Grid size and grid steps are related information
Structured Grids in General Form

- Let’s imagine we have to solve a PDE
Structured Grids in General Form

Let’s imagine we have to solve a PDE

On a dense structured mesh
  ▶ Could be continuously morphed to a Cartesian grid
  ▶ Need to know coordinates of each mesh point
type meshpoint
  real(rk) :: x, y, z
end type

type, extends(meshpoint) :: normal
end type

type mesh
  integer :: nx, ny, nz

  type(meshpoint), dimension(:, :, :), allocatable :: coords

  type(normal), dimension(:, :, :), allocatable :: xnormals
  type(normal), dimension(:, :, :), allocatable :: ynormals
  type(normal), dimension(:, :, :), allocatable :: znormals

  real(rk), dimension(:, :, :), allocatable :: volumes
end type

!...
type(mesh) :: my_mesh

! allocate my_mesh components with extents nx, ny, nz
! immediately checking for failures!
A Recurrent Issue, Again

\begin{verbatim}
real(rk) :: x(NX, NY, NZ)
real(rk) :: y(NX, NY, NZ)
real(rk) :: z(NX, NY, NZ)

Or

type meshpoint
   real(rk) :: x, y, z
end type


type(meshpoint), dimension(NX, NY, NZ) :: coords
\end{verbatim}

Which one is best?
A Recurrent Issue, Again

```plaintext
real(rk) :: x(NX,NY,NZ)
real(rk) :: y(NX,NY,NZ)
real(rk) :: z(NX,NY,NZ)

Or

type meshpoint
  real(rk) :: x, y, z
end type

type(meshpoint), dimension(NX,NY,NZ) :: coords
```

Which one is best?

Again, both have merits
A Recurrent Issue, Again

- real(rk) :: x(NX,NY,NZ)
  real(rk) :: y(NX,NY,NZ)
  real(rk) :: z(NX,NY,NZ)

Or

type meshpoint
  real(rk) :: x, y, z
end type

type(meshpoint), dimension(NX,NY,NZ) :: coords

Which one is best?

- Again, both have merits
  - The former (if done properly) allows hardware to play efficient tricks in memory accesses
A Recurrent Issue, Again

- real(rk) :: x(NX,NY,NZ)
- real(rk) :: y(NX,NY,NZ)
- real(rk) :: z(NX,NY,NZ)

Or

type meshpoint
  real(rk) :: x, y, z
end type

type(meshpoint), dimension(NX,NY,NZ) :: coords

Which one is best?

Again, both have merits

- The former (if done properly) allows hardware to play efficient tricks in memory accesses
- The latter brings in cache all values related to a grid point as soon as one component is accessed
A Recurrent Issue, Again

- real(rk) :: x(NX,NY,NZ)
- real(rk) :: y(NX,NY,NZ)
- real(rk) :: z(NX,NY,NZ)

Or

type meshpoint
  real(rk) :: x, y, z
end type

type(meshpoint), dimension(NX,NY,NZ) :: coords

Which one is best?

- Again, both have merits
  - The former (if done properly) allows hardware to play efficient tricks in memory accesses
  - The latter brings in cache all values related to a grid point as soon as one component is accessed

- Here, we lean to the latter
A Recurrent Issue, Again

- `real(rk) :: x(NX,NY,NZ)`
- `real(rk) :: y(NX,NY,NZ)`
- `real(rk) :: z(NX,NY,NZ)`

Or

```fortran
type meshpoint
    real(rk) :: x, y, z
end type
```

`type(meshpoint), dimension(NX,NY,NZ) :: coords`

Which one is best?

- Again, both have merits
  - The former (if done properly) allows hardware to play efficient tricks in memory accesses
  - The latter brings in cache all values related to a grid point as soon as one component is accessed

- Here, we lean to the latter
  - As in most numerical schemes, $x$, $y$, and $z$ components of the same mesh point are accessed together
A multiblock mesh is an assembly of connected structured meshes.
Multiblock Meshes and More

- A multiblock mesh is an assembly of connected structured meshes
  - You could dynamically allocate a mesh array
  - Or build a block type including a mesh and connectivity information

- Adaptive Mesh Refinement
  - You want your blocks resolution to adapt to dynamical behavior of PDE solution
  - Which means splitting blocks to substitute part of them with more resolved meshes

- Eventually, you'll need more advanced data structures
  - Like lists
  - Like binary trees, oct-trees, n-ary trees
Multiblock Meshes and More

- A multiblock mesh is an assembly of connected structured meshes
  - You could dynamically allocate a mesh array
  - Or build a block type including a mesh and connectivity information

- Adaptive Mesh Refinement
A multiblock mesh is an assembly of connected structured meshes

- You could dynamically allocate a mesh array
- Or build a block type including a mesh and connectivity information

Adaptive Mesh Refinement

- You want your blocks resolution to adapt to dynamical behavior of PDE solution
- Which means splitting blocks to substitute part of them with more resolved meshes
Multiblock Meshes and More

- A multiblock mesh is an assembly of connected structured meshes
  - You could dynamically allocate a mesh array
  - Or build a block type including a mesh and connectivity information

- Adaptive Mesh Refinement
  - You want your blocks resolution to adapt to dynamical behavior of PDE solution
  - Which means splitting blocks to substitute part of them with more resolved meshes

- Eventually, you’ll need more advanced data structures
Multiblock Meshes and More

- A multiblock mesh is an assembly of connected structured meshes
  - You could dynamically allocate a mesh array
  - Or build a block type including a mesh and connectivity information

- Adaptive Mesh Refinement
  - You want your blocks resolution to adapt to dynamical behavior of PDE solution
  - Which means splitting blocks to substitute part of them with more resolved meshes

- Eventually, you’ll need more advanced data structures
  - Like lists
  - Like binary trees, oct-trees, n-ary trees
If You Read Code Like This...

type block_item
  type(block), pointer :: this_block

  type(block_item), pointer :: next
end type

!...
  do while (associated(p))
    call advance_block_in_time(p%this_block)
    p => p%next
  end do
If You Read Code Like This...

```plaintext
type block_item
  type(block), pointer :: this_block
  type(block_item), pointer :: next
end type

!...
  do while (associated(p))
    call advance_block_in_time(p%this_block)
    p => p%next
  end do
```

- It is processing a singly-linked list of mesh blocks
- You know how to handle it, now
And If You Read Code Like This...

```fortran
type block_tree_node
    type(block), pointer :: this_block

    integer :: children_no
    type(block_tree_node), pointer :: childrens

    type(block_tree_node), pointer :: next_sibling
end type

recursive subroutine tree_advance_in_time(n)
    type(block_tree_node) :: n
    type(block_tree_node), pointer :: p
    integer :: i

    p => n%childrens
    do i=0,n%children_no
        call tree_advance_in_time(p)
        p => p%next_sibling
    end do

    call advance_block_in_time(n%this_block)
end subroutine tree_advance_in_time
```

It is processing a tree of mesh blocks (AMR, probably)

You need to learn more on abstract data structures

Don't be afraid, it's not that difficult
And If You Read Code Like This...

```
type block_tree_node
  type(block), pointer :: this_block

  integer :: children_no
  type(block_tree_node), pointer :: childrens

  type(block_tree_node), pointer :: next_sibling
end type

!...
recursive subroutine tree_advance_in_time(n)
  type(block_tree_node) :: n
  type(block_tree_node), pointer :: p
  integer :: i

  p => n%childrens
  do i=0,n%children_no
    call tree_advance_in_time(p)
    p => p%next_sibling
  end do

  call advance_block_in_time(n%this_block)
end subroutine tree_advance_in_time
```

▶ It is processing a tree of mesh blocks (AMR, probably)
▶ You need to learn more on abstract data structures
▶ Don’t be afraid, it’s not that difficult
Outline

Extending the Language

Managing Memory
  Dynamic Memory Allocation
  Fortran Pointers
  Sketchy Ideas on Data Structures
  Bridging the Gap with C

Conclusions
Mixing C and Fortran

- You may want to call a C function from a Fortran program
- Or call a Fortran procedure from a C program
- And you don’t want to translate and re-debug
- Or you can’t, as you don’t have sources
- You may also want to share global data among C and Fortran program units

- This has been done in the past with non-standard tricks
- Fortran 2003 offers a better, standard way
- Let’s look at it in steps
Two Naive Examples

Imagine you have this C function:

```c
double avg_var(int n, const double a[], double *var) {
    double avg = 0.0;
    double avg2 = 0.0;
    for(int i=0;i<n;i++) {
        avg += a[i];
        avg2 += a[i]*a[i];
    }
    avg = avg/n;
    *var = avg2/n - avg*avg;
    return avg;
}
```

and you want to call it from your Fortran code like:

```
avg = avg_var(m,b,var)
```
Two Naive Examples

▶ Imagine you have this C function:

```c
double avg_var(int n, const double a[], double *var) {
    double avg = 0.0;
    double avg2 = 0.0;
    for(int i=0;i<n;i++) {
        avg += a[i];
        avg2 += a[i]*a[i];
    }
    avg = avg/n;
    *var = avg2/n - avg*avg;
    return avg;
}
```

and you want to call it from your Fortran code like:

```fortran
avg = avg_var(m,b,var)
```

▶ Or you have your favorite, thoroughly tested Poisson solver:

```fortran
interface
    subroutine myPoissonSolver(l, m, n, f)
        integer, intent(in) :: l, m, n
        real(kind(1.0D0)), intent(inout) :: f(l,m,n)
    end subroutine myPoissonSolver
end interface
```

and you want to call it from your C code like:

```c
myPoissonSolver(nx, ny, nz, field);
```
We could think that Fortran interfaces and C declarations are enough

And write, to call C from Fortran:

```fortran
interface
  function avg_var(n, a, var)
    integer, intent(in) :: n
    real(kind(1.0D0)), intent(in) :: a(*)
    real(kind(1.0D0)), intent(out) :: var
    real(kind(1.0D0)) :: avg_var
  end function avg_var
end interface
```

And to call Fortran from C, add on Fortran side:

```fortran
interface
  subroutine myPoissonSolver(l, m, n, f)
    integer, intent(in) :: l, m, n
    real(kind(1.0D0)), intent(inout) :: f(l,m,n)
  end subroutine myPoissonSolver
end interface
```

and on the C side, the declaration:

```c
void myPoissonSolver(int nx, int ny, int nz, field[nz][ny][nx]);
```

This is the right track, but still half way from our destination
Thou Shalt Not Mangle Names

- Fortran compilers mangle procedure names
  - All uppercase or all lowercase
  - Compilers may append/prepend one or two _ characters
  - And for module procedures is even worse
  - Used to be sorted out on the C side, in non-portable ways
Thou Shalt Not Mangle Names

- Fortran compilers mangle procedure names
  - All uppercase or all lowercase
  - Compilers may append/prepend one or two `_` characters
  - And for module procedures is even worse
  - Used to be sorted out on the C side, in non-portable ways

- Enter Fortran 2003 `bind` attribute
Thou Shalt Not Mangle Names

- Fortran compilers mangle procedure names
  - All uppercase or all lowercase
  - Compilers may append/prepend one or two _ characters
  - And for module procedures is even worse
  - Used to be sorted out on the C side, in non-portable ways

- Enter Fortran 2003 **bind** attribute

- For C to Fortran:

```fortran
interface
  function avg_var(n, a, var) bind(c)
    integer, intent(in) :: n
    real(kind(1.0D0)), intent(in) :: a(*)
    real(kind(1.0D0)), intent(out) :: var
    real(kind(1.0D0)) :: avg_var
  end function avg_var
end interface
```
Thou Shalt Not Mangle Names

- Fortran compilers mangle procedure names
  - All uppercase or all lowercase
  - Compilers may append/prepend one or two _ characters
  - And for module procedures is even worse
  - Used to be sorted out on the C side, in non-portable ways

- Enter Fortran 2003 **bind** attribute

- For C to Fortran:

```fortran
interface
  function avg_var(n, a, var) bind(c)
    integer, intent(in) :: n
    real(kind(1.0D0)), intent(in) :: a(*)
    real(kind(1.0D0)), intent(out) :: var
    real(kind(1.0D0)) :: avg_var
  end function avg_var
end interface
```

- For Fortran to C, Fortran side:

```fortran
interface
  subroutine myPoissonSolver(l, m, n, f) bind(c)
    integer, intent(in) :: l, m, n
    real(kind(1.0D0)), intent(inout) :: f(l, m, n)
  end subroutine myPoissonSolver
end interface
```

and on the C side, the declaration:

```c
void myPoissonSolver(int nx, int ny, int nz, field[nz][ny][nx]);
```
Thou Shalt Care for Argument Passing

- Fortran passes arguments by reference
  - Under the hood, it’s like a C pointer
  - Works for C arrays and pointers to scalar variables
  - But usually scalars are passed by value in C
Thou Shalt Care for Argument Passing

- Fortran passes arguments by reference
  - Under the hood, it’s like a C pointer
  - Works for C arrays and pointers to scalar variables
  - But usually scalars are passed by value in C

- Enter Fortran 2003 `value` attribute
Thou Shalt Care for Argument Passing

- Fortran passes arguments by reference
  - Under the hood, it’s like a C pointer
  - Works for C arrays and pointers to scalar variables
  - But usually scalars are passed by value in C

- Enter Fortran 2003 `value` attribute

- For C to Fortran:

  ```
  interface
    function avg_var(n, a, var) bind(c)
      integer, value :: n
      real(kind(1.0D0)), intent(in) :: a(*)
      real(kind(1.0D0)), intent(out) :: var
      real(kind(1.0D0)) :: avg_var
    end function avg_var
  end interface
  ```
Thou Shalt Care for Argument Passing

- Fortran passes arguments by reference
  - Under the hood, it's like a C pointer
  - Works for C arrays and pointers to scalar variables
  - But usually scalars are passed by value in C

- Enter Fortran 2003 `value` attribute

- For C to Fortran:

```fortran
interface
  function avg_var(n, a, var) bind(c)
    integer, value :: n
    real(kind(1.0D0)), intent(in) :: a(*)
    real(kind(1.0D0)), intent(out) :: var
    real(kind(1.0D0)) :: avg_var
  end function avg_var
end interface
```

- For Fortran to C, Fortran side:

```fortran
interface
  subroutine myPoissonSolver(l, m, n, f) bind(c)
    integer, value :: l, m, n
    real(kind(1.0D0)), intent(inout) :: f(l,m,n)
  end subroutine myPoissonSolver
end interface
```

and on the C side, still the declaration:
```c
void myPoissonSolver(int nx, int ny, int nz, field[nz][ny][nx]);
```
Thou Shalt Care for Data Size and Layout

- Fortran is quite liberal on data sizes
  - Implementations have a lot of freedom
  - And C is also quite liberal
Thou Shalt Care for Data Size and Layout

- Fortran is quite liberal on data sizes
  - Implementations have a lot of freedom
  - And C is also quite liberal

- Enter Fortran 2003 `iso_c_binding` module

```fortran
module iso_c_binding

contains

interface function avg_var(n, a, var) bind(c)
  use iso_c_binding
  integer(c_int), value :: n
  real(c_double), intent(in) :: a(*)
  real(c_double), intent(out) :: var
  real(c_double) :: avg_var
  end function avg_var
end interface

interface subroutine myPoissonSolver(l, m, n, f) bind(c)
  use iso_c_binding
  integer(c_int), value :: l, m, n
  real(c_double), intent(inout) :: f(l,m,n)
  end subroutine myPoissonSolver
end interface

end module iso_c_binding
```

```
and on the C side, still the declaration:

```c
void myPoissonSolver(int nx, int ny, int nz, field[nz][ny][nx]);
```
Thou Shalt Care for Data Size and Layout

- Fortran is quite liberal on data sizes
  - Implementations have a lot of freedom
  - And C is also quite liberal

- Enter Fortran 2003 `iso_c_binding` module

- For C to Fortran:
  ```fortran
  interface
    function avg_var(n, a, var) bind(c)
      use iso_c_binding
      integer(c_int), value :: n
      real(c_double), intent(in) :: a(*)
      real(c_double), intent(out) :: var
      real(c_double) :: avg_var
    end function avg_var
  end interface
  ```
Thou Shalt Care for Data Size and Layout

▶ Fortran is quite liberal on data sizes
  ▶ Implementations have a lot of freedom
  ▶ And C is also quite liberal

▶ Enter Fortran 2003 iso_c_binding module

▶ For C to Fortran:

```fortran
interface
 function avg_var(n, a, var) bind(c)
   use iso_c_binding
   integer(c_int), value :: n
   real(c_double), intent(in) :: a(*)
   real(c_double), intent(out) :: var
   real(c_double) :: avg_var
   end function avg_var
end interface
```

▶ For Fortran to C, Fortran side:

```fortran
interface
 subroutine myPoissonSolver(l, m, n, f) bind(c)
   use iso_c_binding
   integer(c_int), value :: l, m, n
   real(c_double), intent(inout) :: f(l,m,n)
   end subroutine myPoissonSolver
end interface
```

and on the C side, still the declaration:

```c
void myPoissonSolver(int nx, int ny, int nz, field[nz][ny][nx]);
```
More from *iso_c_binding*

- *iso_c_binding* defines named constants holding kind type parameter values for intrinsic types for the platform
- `integer(c_int)` is the kind value corresponding to a C `int`
- Negative values are used for unsupported C types, so the compiler will flag the problem

<table>
<thead>
<tr>
<th>Type</th>
<th>Kind</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td><code>c_int</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td></td>
<td><code>c_short</code></td>
<td><code>short int</code></td>
</tr>
<tr>
<td>real</td>
<td><code>c_float</code></td>
<td><code>float</code></td>
</tr>
<tr>
<td></td>
<td><code>c_double</code></td>
<td><code>double</code></td>
</tr>
<tr>
<td>complex</td>
<td><code>c_float_complex</code></td>
<td><code>float _Complex</code></td>
</tr>
<tr>
<td></td>
<td><code>c_double_complex</code></td>
<td><code>double _Complex</code></td>
</tr>
<tr>
<td>logical</td>
<td><code>c_bool</code></td>
<td><code>_Bool</code></td>
</tr>
<tr>
<td>character</td>
<td><code>c_char</code></td>
<td><code>char</code></td>
</tr>
</tbody>
</table>

- Fortran 2008 adds `c_sizeof()`, check with your compiler!
Mapping Arrays

- Fortran has multidimensional arrays
- C has arrays of arrays (of arrays...)
- Thus the mapping of array indexes to actual data layout in memory is inverted
  - Fortran array $a(L, M, N)$
  - maps to C array $a[N][M][L]$
Mapping Arrays

- Fortran has multidimensional arrays
- C has arrays of arrays (of arrays...)
- Thus the mapping of array indexes to actual data layout in memory is inverted
  - Fortran array \( a(L, M, N) \)
  - maps to C array \( a[N][M][L] \)

- Before C99, the leading dimension of an array function parameter could not be specified in C
  - C array parameter \( a[] \)
  - maps to Fortran assumed size array parameter \( a(*) \)
Mapping Arrays

- Fortran has multidimensional arrays
- C has arrays of arrays (of arrays...)
- Thus the mapping of array indexes to actual data layout in memory is inverted
  - Fortran array \( a(L, M, N) \)
  - maps to C array \( a[N][M][L] \)
- Before C99, the leading dimension of an array function parameter could not be specified in C
  - C array parameter \( a[] \)
  - maps to Fortran assumed size array parameter \( a(*) \)
- In C99, Variable Length Arrays were introduced
  - C99 array parameter \( a[nz][ny][nx] \)
  - maps to Fortran array parameter \( a(nx, ny, nz) \)
### Derived Types and Global Data

- **bind** also helps for derived types and global data
- For derived types, each component must be interoperable

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C</th>
</tr>
</thead>
</table>
| type, bind(c) :: particle  
  integer(c_int) :: n  
  real(c_float) :: x,y,z  
  real(c_float) :: vx,vy,vz  
end type particle | typedef struct particle {
  int n;
  float x,y,z;
  float vx,vy,vz;
} particle; |

- For module variables or common blocks, use

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C</th>
</tr>
</thead>
</table>
| integer(c_long), bind(c) :: n  
  real(c_double) :: m,k  
  common /com_mk/ m,k  
  bind(c) :: /com_mk/ | extern long n;
  extern struct mk {
    double m, k;
  } com_mk; |

- Note: common blocks become C *structs*
Fortran Pointers vs. C Pointers

- As of argument passing, not a problem
- But Fortran pointers are not interoperable with C
- Fortran pointers sport richer semantics, notably:
  - multidimensional arrays
  - non-contiguous memory areas

- C functions returning a pointer must have `type(c_ptr)` type (from `iso_c_binding`)
- Ditto for C pointer variables and pointer members of C structs:

```fortran
  type, bind(c) :: block
    integer(c_int) :: n_neighbors
    type(c_ptr) :: neighbors
    type(c_ptr) :: grid
  end type block
```

```c
  typedef struct {
    int n_neighbors;
    int *neighbors;
    mesh *grid;
  } block;
```
Translating Pointers Back and Forth

- **iso_c_binding** module provides much needed help
Translating Pointers Back and Forth

- **iso_c_binding** module provides much needed help

- **c_loc(x)** returns a valid C pointer to the content of variable \( x \)

- **c_f_pointer(cp, fp[, shape])** performs the opposite translation, writing the result in the Fortran pointer \( fp \)
  - An optional *shape* argument like \((/n/\) or \((/l,m,n/)\) gives it a shape for array pointers
Translating Pointers Back and Forth

- **iso_c_binding** module provides much needed help

- **c_loc** returns a valid C pointer to the content of variable `x`

- **c_f_pointer** performs the opposite translation, writing the result in the Fortran pointer `fptr`
  - An optional `shape` argument like `/n/` or `/l,m,n/` gives it a shape for array pointers

- If `f_proc` is an interoperable Fortran procedure, `c_funloc(f_proc)` returns a valid C pointer (type(c_funptr)) to it

- **c_f_procpointer** performs the opposite translation, writing the result in the Fortran procedure pointer `fp.ptr`
Thou Shalt Compile and Link Properly

- Obviously, C and Fortran sources must be separately compiled and then linked

  ```
  user@cineca$> gcc -c fun_cmd.c
  user@cineca$> gfortran -c main_cmd.f90
  user@cineca$> gfortran fun_cmd.o main_cmd.o -o main_cmd
  ```

- Easy, if calling C functions from a Fortran program
  - Fortran Runtime Library is usually built on top of C one

- Your mileage may vary, browse your compiler manuals
Thou Shalt Compile and Link Properly

▶ Obviously, C and Fortran sources must be separately compiled and then linked

```bash
user@cineca$> gcc -c fun_cmd.c
user@cineca$> gfortran -c main_cmd.f90
user@cineca$> gfortran fun_cmd.o main_cmd.o -o main_cmd
```

▶ Easy, if calling C functions from a Fortran program
  ▶ Fortran Runtime Library is usually built on top of C one

▶ Less so if calling Fortran procedures from a C program
  ▶ Fortran compiler might insert calls to its Runtime Library

▶ Best practice:

```bash
user@cineca$> gcc -lgfortran procedures.o main.c
```

▶ Your mileage may vary, browse your compiler manuals
Hands-on 4: qsort

Write the Fortran interface to C qsort

```fortran
module qsort_c_to_fortran
    use iso_c_binding
    integer, parameter :: sp = kind(1.0)
    interface
        !Write the Fortran interface to C qsort!
        !void qsort(void *base,
        ! size_t nmemb,
        ! size_t size,
        ! int (*compar)(const void *,const void *));
    end interface
    contains
    function compare_reals(a,b) bind(c)
        integer(c_int) :: compare_reals
        real(c_float) :: a,b
        if(a>b) then
            compare_reals=1
        else if(a<b) then
            compare_reals=-1
        else
            compare_reals=0
        endif
    end function compare_reals
end module qsort_c_to_fortran
```

```fortran
program test_qsort_c
    use qsort_c_to_fortran
    integer(c_size_t), parameter :: n=7
    real(c_float), pointer :: a(:)
    allocate(a(n))
    call random_number(a)
    print*, 'Unordered a: '
    print*, a
    call qsort(c_loc(a(1)), n, c_sizeof(a(1)), &
               c_funloc(compare_reals));
    print*, 'Ordered a: '
    print*, a
end program test_qsort_c
```
Outline

Extending the Language

Managing Memory

Conclusions
What We Left Out

► More Fortran practice
  ► Time was tight, and that’s your job
What We Left Out

- More Fortran practice
  - Time was tight, and that’s your job

- More about programming
  - Code development management tools
  - Debugging tools
  - Look among CINECA HPC courses
What We Left Out

- More Fortran practice
  - Time was tight, and that’s your job

- More about programming
  - Code development management tools
  - Debugging tools
  - Look among CINECA HPC courses

- More Fortran
  - Full object oriented programming
  - Floating point environment
  - Direct I/O
  - Asynchronous I/O
  - Submodules
  - Even more format edit descriptors
  - A few more statements and quite a few intrinsics
  - Coarrays
Looking for More

J3 US Fortran Standards Committee
http://www.j3-fortran.org/

ISO WG5 Committee
http://www.nag.co.uk/sc22wg5/

Fortran 2003 Standard Final Draft
Search Internet for n3661.pdf

Fortran Wiki
http://fortranwiki.org/

M. Metcalf, J. Reid, M. Cohen
Fortran 95/2003 Explained
Oxford University Press, corrected ed., 2008

M. Metcalf, J. Reid, M. Cohen
Modern Fortran Explained
Oxford University Press, 2011

S. Chapman
Fortran 95/2003 for Scientists & Engineers

Adams, J.C., Brainerd, W.S., Hendrickson, R.A., Maine, R.E., Martin, J.T., Smith, B.T.
The Fortran 2003 Handbook
Springer, 2009
Advanced Fortran Coding Examples

Salvatore Filippone's Home Page
www.ce.uniroma2.it/people/filippone.html

Parallel Sparse Basic Linear Algebra Subroutines
www.ce.uniroma2.it/psblas/index.html

Numerical Engine (for) Multiphysics Operators
www.ce.uniroma2.it/nemo/index.html

Portable Fortran Interfaces to the Trilinos C++ Package
trilinos.sandia.gov/packages/fortrilinos/

Stefano Toninel
Development of a New Parallel Code for Computational Continuum Mechanics Using Object-Oriented Techniques
These slides are ©CINECA 2013 and are released under the Attribution-NonCommercial-NoDerivs (CC BY-NC-ND) Creative Commons license, version 3.0.

Uses not allowed by the above license need explicit, written permission from the copyright owner. For more information see:

http://creativecommons.org/licenses/by-nc-nd/3.0/

Slides and examples were authored by:

- Federico Massaioli
- Marco Rorro
- Michela Botti
- Francesco Salvadore