Introduction to Fortran

Why Fortran?
- A lot of legacy software written in it (well, often in Fortran 77).
- High-level abstraction (as compared to C); e.g. handling of arrays.
- Large number of numerical libraries built for Fortran.

Why not Fortran?
- Compilers (particularly free ones) not so easy to find on different environments.
- Some things easier to implement in C.

Often a question of personal taste and the fact which language you learned first.

Different generations of Fortran¹:

1. FORTRAN66
2. FORTRAN77 These two must not be used anymore!
3. Fortran 90 These two are now in use.
4. Fortran 95 No large differences between F90 and F95.
5. Fortran 2003 Most features implemented by most compilers.
6. Fortran 2008 Some features implemented by many compilers.

- From now on I call the language simply Fortran or F90+ (meaning all other version except FORTRAN77).
- If there is need to distinguish between 90, 95, 2003, and 2008 versions will tell it explicitly.
- A good source of compiler support of the latest Fortran features can be found at

¹. For more info on history of programming languages see e.g. http://www.levenez.com/lang/

Introduction to Fortran

Material

- Fortran 95/2003 Explained by Michael Metcalf, John Reid and Malcolm Cohen, Oxford University Press
- A newer version Modern Fortran Explained by the same authors and publisher
- CSC has published a free Fortran 95/2003 guide in Finnish (see http://www.csc.fi/csc/julkaisut/oppaat)
- Small example programs presented in these lecture notes can be downloaded from
  http://www.physics.helsinki.fi/courses/s/stltk/progs/
- IBM has a nice reference manual for their compiler at
  (choose the ‘XL Fortran Advanced Edition for Linux PDF files’ and there ‘Language Reference...’)
- Note, however, that it is a language reference manual not a textbook for learning Fortran.
Introduction to Fortran

- Structure of a Fortran program:

```fortran
program name_of_the_program
! This is a comment
implicit none
! type and variable definitions
!
! executable statements
!
! stop
contains ! Comment again
! local subroutine definitions
!
end program name_of_the_program
```

- `implicit none` is the most important statement in F90

- In older Fortrans `implicit type definitions` were commonly used:
  - variables beginning with i,j,k,l,m,n are integers
  - others are real.

- If you leave out `implicit none` the same convention is used in F90+! Then...you get what you deserve!

- Subroutines defined after `contains` statement are local in the sense that they are not visible to any other routines and that they see the local variables.

- Comments begin with exclamation mark. The rest of the line is ignored (just as the `//`-comment in C++).

Urban legend (?): Fortran compilers used to ignore white space. This combined with implicit typing caused a typo go unnoticed: in a do loop start a period was written instead of a comma:

```fortran
do 100 i=1,10
```

This was interpreted as an assignment: `do100i=1.10` i.e. no loop at all!

- Note that Fortran is case insensitive!

Introduction to Fortran

- Structure of a Fortran procedure definition

```fortran
subroutine sub(p1,p2)
implicit none
! procedure parameter definitions
!
! local type and variable definitions
!
! executable statements
!
return
contains ! Comment again
! local subroutine definitions
!
end subroutine sub
```

```fortran
function func(p1,p2)
implicit none
! definitions of the function and it parameters
!
! local type and variable definitions
!
! executable statements
!
func=... ! assign a return value
return
contains ! Comment again
! local subroutine definitions
!
end function func
```

- `subroutine` never returns a value (as a C function defined as `void`)
  - Refer to subroutine as `call sub(p1,p2)`
- `function` always returns a value
  - Refer to function simply by using it in an expression or as a procedure parameter `x=func(p1,p2)`
Introduction to Fortran

• Built-in datatypes

- Integer

```fortran
integer, parameter :: ik = selected_int_kind(9)
integer(kind=ik) :: i
```

- With the kind keyword and intrinsic function `selected_int_kind()` one can select the ‘size’ of the datatype:

  `selected_int_kind(p)` returns the kind type that represents all integers $n$ in the range $-10^p < n < 10^p$.

- Example

```fortran
program intkind
  implicit none
  integer, parameter :: ik = selected_int_kind(2)
  integer(kind=ik) :: i
  integer :: j
  i=1
  do j=1,20
    i=i*2
    write(0,*), j,i
  end do
end program intkind
```

- Default integer (i.e. if you say only `integer :: i`) is in most environments of size 32 bits (corresponding to $p = 9$).

- Real

```fortran
integer, parameter :: rk = selected_real_kind(6,15)
real(kind=rk) :: x
```

- Parameters of the function `selected_real_kind(p,r)` tell the minimum decimal precision ($p$) and range of the exponent ($r$) of the real type.

- Example

```fortran
program realkind
  implicit none
  integer, parameter :: rk = selected_real_kind(6,15)
  real(kind=rk) :: x
  integer :: j
  x=1.0
  do j=1,64
    x=x*100.0
    write(0,*), j,x
  end do
end program realkind
```

- Default real (using only `real :: x`) is in most systems 32 bit long (i.e. corresponds to `float` in C).
- But not always!

- This kind-qualifier is exactly the feature that should prevent problems with different word lengths.
- By using it your code is easier to port to other systems. (Well, in practice, most machines today use IEEE floating point numbers)
Introduction to Fortran

- **Complex**

  ```fortran
  integer, parameter :: rk=selected_real_kind(10,40)
  complex(kind=rk) :: z
  z=(1.0,1.0)
  ```

- **Boolean**

  ```fortran
  logical :: firsttime=.true.
  ...
  if (firsttime) then
    ...
  firsttime=.false.
  end if
  ```

- **Strings**

  ```fortran
  character(len=80) :: c
  c='No, joo!
  write(0,*) c
  ```

  Substring: `c(5:10), c(:4), c(11:)`

  - Note that string handling in F90 is much easier than in C, methinks:

    ```fortran
    character(len=80) :: c,d,e
    c="first"
    d="last"
    e=trim(c)\" and \"//trim(d)
    write(0,*) e
    ```

- **Comparison**

  ```fortran
  character(len=80) :: c,d
  c="first"
  ...
  d="last"
  if (c==d) then
    write(0,*) "equal!"
  endif
  ```

  Function `trim` strips the trailing spaces from the string.

- **Own type definitions** (structs)

  ```fortran
  program typedefest
  implicit none
  
  type person
    character(len=20) :: first,last
    integer :: age
  end type person
  
  type(person) :: student
  student%first='Matti'
  student%last='Meikäläinen'
  student%age=75
  
  ! Not sure whether outputting the whole ! struct is standard conforming.
  ! write(6,*) student
  
  stop
  end program typedefest
  ```

  Command:

  ```bash
  progs> gfortran typedef.f90
  progs> a.out
  Matti   Meikäläinen   75
  ```
Introduction to Fortran

**Conditional execution**
If-then-else

```fortran
if (logical expression) then
   executable statements
else if (logical expression) then
   executable statements
else
   executable statements
end if
```

In case there is only one statement after the if construction then-endif pair can be left out; e.g.,

```fortran
if (i>10) a(i)=0.0
```

- Instead of integers you can also use logical and character expressions in the case construct.
- Individual cases can also be lists of values and ranges, e.g. `case(4:10,12,14)`
- No need to use break commands as in C.

```fortran
program casetest
implicit none
integer :: i
character(len=10) :: c
read(5,*) i
select case(i)
case(1)
c='one'
case(2)
c='two'
case(3)
c='three'
case default
c='many'
end select
write(0,*) c
stop
end program casetest
```

**Comparison operators**

In FORTRAN77 the comparison operators had the dotted forms given below in the last column:

<table>
<thead>
<tr>
<th>Operator</th>
<th>New form</th>
<th>Old form</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal</td>
<td>==</td>
<td>.eq.</td>
</tr>
<tr>
<td>not equal</td>
<td>/=</td>
<td>.ne.</td>
</tr>
<tr>
<td>less</td>
<td>&lt;</td>
<td>.lt.</td>
</tr>
<tr>
<td>greater</td>
<td>&gt;</td>
<td>.gt.</td>
</tr>
<tr>
<td>less or equal</td>
<td>&lt;=</td>
<td>.le.</td>
</tr>
<tr>
<td>greater or equal</td>
<td>&gt;=</td>
<td>.ge.</td>
</tr>
</tbody>
</table>

- Both forms can be used in F90+.

**Logical operators**
- These only have the dotted forms.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negation</td>
<td>.not.</td>
</tr>
<tr>
<td>Logical and</td>
<td>.and.</td>
</tr>
<tr>
<td>Logical or</td>
<td>.or.</td>
</tr>
<tr>
<td>Logical equivalence</td>
<td>.eqv.</td>
</tr>
<tr>
<td>Logical inequivalence</td>
<td>.neqv.</td>
</tr>
</tbody>
</table>

Examples:

```
.not.(a<b.and.b>c)
a>=b .or. b>=c
```
Introduction to Fortran

- **Iteration** (do loop)

  
  ```fortran
  [name:] do iter_var=first,last[,step]
  executable statements
  end do [name]
  ```

- Inside the do loop you can use the statement `exit` to stop the iteration (jump out of the loop) or `cycle` to stop the current iteration and continue the next one.

- **Examples:**

  ```fortran
  loop1: do i=1,n
    do j=i,m
      if (a(j)>b(i)) then
        a(j)=b(i)
        cycle loop1
      end if
    end do
  end do
  ```

  ```fortran
  Do loop without an iteration variable:

  do
    executable statements
    ... if (x<-1.0) exit ! To prevent an infinite loop
  end do
  ```

- In F77 real iteration variables were allowed. This is no more the case in F90+
- Naming the loops is helpful if you have many loops within each other and want e.g. jump out from the outer most one.
- It also makes the code more readable.

Introduction to Fortran

- **Arrays**

  - In F77 only statically allocated arrays were allowed.
  - F90 introduced allocatable arrays.

  ```fortran
  Static allocation
  ```

  ```fortran
  integer, dimension(10) :: k
  real, dimension(10,10) :: x
  real, dimension(100) :: x1
  real(kind=rk), dimension(4,4,4) :: y
  ```

  Or more tersely

  ```fortran
  integer :: k(10)
  real :: x(10,10), x1(100)
  real(kind=rk) :: y(4,4,4)
  ```

  ```fortran
  Dynamic allocation
  ```

  ```fortran
  real(kind=rk), dimension(:), allocatable :: x
  real(kind=rk), dimension(:,:), allocatable :: y
  integer :: n,err
  ...
  print *,'Give array size.'
  read *,n
  allocate(x(n),y(0:n,0:n),stat=err)
  if (err/=0) then
    print *,"Could not allocate."
    stop
  end if
  ...
  x=1.0
  y=2.0
  ...
  deallocate(x,y)
  ```

- Some restrictions of allocatable arrays: they can’t be used in
  1. formal parameters of subroutines (use modules instead)
  2. function results
  3. user datatypes (use pointers instead)
Introduction to Fortran

- Arrays or sections of arrays can be easily handled:
  
  ```fortran
  real :: x(100), y(100), z(100)
  x=1.0
  x(4:6)=2.0
  y=2.0
  z=x*y ! Multiply element by element
  ```

- More about array handling later.

- **Pointers** (material from [http://www.personal.psu.edu/faculty/j/h/jhm/f90/lectures/42.html](http://www.personal.psu.edu/faculty/j/h/jhm/f90/lectures/42.html))
  - Pointers in Fortran are more abstract than those in C.
  - Those variables that you want to be pointed by pointers must have the `target` qualifier:
    ```fortran
    real, target :: a, b(1000), c(10,10)
    integer, target :: i, j(100), k(20,20)
    ```
  - Pointers itself have the `pointer` qualifier:
    ```fortran
    real, pointer :: pa, aptr, pb(:), pcl(:), pc2(:,:)
    ```
  - To associate a pointer with a variable you use the `=>` operator
    ```fortran
    pa => a
    ```
    Now you can use `pa` and `a` interchangeably:
    ```fortran
    pa = 1.23456
    print *, 'a = ', a
    ```
  - With array pointer you can do tricks as
    ```fortran
    pb => b
    pb => b(101:200)
    pc2 => c(3:5, 4:6)
    ```
  - Pointers in F90+ can be seen as a generalization of the `equivalence` statement in F77.
  - Personal opinion: Not much needed in HPC.

1. `equivalence(x,y)` --> variables `x` and `y` can now be used interchangeably. In F77, however, this is a static definition. A good tool for making obfuscated code.

Introduction to Fortran

- **Reading and writing files**

  - Open a file: unit number identifies an open file (cf. C file pointer)
    ```fortran
    integer :: st
    character(len=40) :: fname
    real :: x
    ... 
    fname='in.dat'
    open(unit=10,file=fname,status='old',iostat=st)
    if (st/=0) then
      print *, 'Error in opening file.'
      stop
    else
      print *, 'File opened successfully.'
    end if
    ```

  - Reading from a file
    ```fortran
    n=0
    do 
      read(unit=10,iostat=st,fmt='(a)') string
      if (st==0) exit ! end of file
      if (string(1:1)=='#') cycle ! comment line
      if (len_trim(string)==0) cycle ! ignore empty lines
      read(unit=string,fmt='*',iostat=st) x,y,z
      if (st/=0) then
        print '*',"file line ",trim(string)," in wrong format"
        stop
      end if
      n=n+1
      atom(n)%x=x
      atom(n)%y=y
      atom(n)%z=z
    end do
    close(unit=10)
    ```

Specifier `iostat=st` set the status value of the io operation to integer variable `st`.

- `st==0` indicates success
- `st<0` means that EOF occurred
Introduction to Fortran

- Writing to a file (formatted, i.e. ASCII)
  
  ```fortran
  open(unit=10, file=fname, status='replace', iostat=ios)
  write(unit=10,*) 'x =', x
  close(unit=10)
  ```

- It is usual in `open`, `close`, `read`, and `write` statements to leave out the keywords `unit` and `fmt`
  
  ```fortran
  open(unit=10, file=fname) → open(10, file=fname)
  close(unit=10) → close(10)
  write(unit=10, fmt='*') → write(10, '*')
  read(unit=10, fmt='(a)') → read(10, '(a)')
  ```

- The format string defined by keyword `fmt` determines in which form data is read in and printed out (cf. format string in `sprintf` in C; see next slide)
  - Probably the easiest way (in the beginning at least) is to use so called list-directed formatting; i.e. using `'*'` as the format string.
  - In this case the system decides what format to use based on what is being printed or read; e.g.
    ```fortran
    write(6,*) 'Time step ', dt
    read(5,*) x
    ```

- Predetermined units:
  
  - `0 = stderr`
  - `5 = stdin`
  - `6 = stdout`
  - These units need not be explicitly opened.
  - Units 5 and 6 can be replaced by `'*'` in reading and writing, respectively.
  - Output to stdout can also be done using the `print` statement:
    ```fortran
    print format_specifier, io_list
    ```
  - E.g. `print *, 'X is now ', x`

Introduction to Fortran

- The most important format specifiers:
  
  **Writing**

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw</code></td>
<td>write an integer to next <code>w</code> positions</td>
</tr>
<tr>
<td><code>fw.d</code></td>
<td>write a real with <code>d</code> decimals to next <code>w</code> positions</td>
</tr>
<tr>
<td><code>ew.d</code></td>
<td>write a real with <code>d</code> decimals in exponent form to next <code>w</code> positions</td>
</tr>
<tr>
<td><code>gw.d</code></td>
<td>write a real with <code>d</code> decimals in normal or exponent form to next <code>w</code> positions</td>
</tr>
<tr>
<td><code>aw</code></td>
<td>write a string to next <code>w</code> positions</td>
</tr>
<tr>
<td><code>a</code></td>
<td>write a string starting from the next position</td>
</tr>
<tr>
<td><code>nx</code></td>
<td>skip <code>n</code> next positions</td>
</tr>
<tr>
<td><code>/</code></td>
<td>line break</td>
</tr>
</tbody>
</table>

  **Reading**

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw</code></td>
<td>read an integer from next <code>w</code> positions</td>
</tr>
<tr>
<td><code>fw.d</code></td>
<td>read next <code>w</code> characters as a real with <code>d</code> decimals in case there is no period</td>
</tr>
<tr>
<td><code>ew.d</code></td>
<td>same as above</td>
</tr>
<tr>
<td><code>aw</code></td>
<td>read next <code>w</code> characters as a string</td>
</tr>
<tr>
<td><code>a</code></td>
<td>read as many characters as is needed to fill the character variable</td>
</tr>
<tr>
<td><code>nx</code></td>
<td>skip <code>n</code> next positions</td>
</tr>
<tr>
<td><code>/</code></td>
<td>line break</td>
</tr>
</tbody>
</table>

- You can use these for detailed formatting of io but the easiest way is to use the list-directed io (`fmt=*`)
Introduction to Fortran

- A simple example of reading from stdin and writing to stdout:

```fortran
program readtest
  implicit none
  real :: x
  integer :: i,s
  character(len=20) :: c
  do
    print *,'Give x,i,c.'
    read(*,*,iostat=s) x,i,c
    if (s/=0) exit
    print *,2*x,10*i,len_trim(c)
  end do
  print *,'Bye!'
  stop
end program readtest
```

- Functions are called as they are in C: as a part of an expression
  \[ x=\sin(\theta) \]
  \[ y=myfunc(i,j,z) \]

- Definition of a function; example

```fortran
function area(r)
  implicit none
  integer,parameter :: rk=selected_real_kind(10,40)
  real(rk) :: area
  real(rk),intent(in) :: r
  area=4.0*atan2(1.0,1.0)*r**2
  return
end function area
```

- With `intent(in), intent(out), intent(inout)` you can specify whether the function or subroutine parameters are only input, only output or both input and output parameters, respectively. For example:

```fortran
program intenttest
  real :: x,y,f
  x=1.0
  y=f(x)
  stop
end program intenttest
```

```fortran
function f(x)
  implicit none
  real :: f
  real,intent(in) :: x
  x=2.0*x
  f=x
  return
end function f
```

Compilation:
```
progs> gfortran intenttest.f90
In file intenttest.f90:11
  x=2.0*x
  1  Error: Can't assign to INTENT(IN) variable 'x' at (1)
```
Introduction to Fortran

- Subroutines are called by statement **call**:

  ```fortran
  call mysub(a,b,1,j)
  ```

- Definition of a subroutine; example

  ```fortran
  subroutine sub(x,y,n)
  implicit none
  integer, parameter :: rk=selected_real_kind(10,40)
  integer, intent(in) :: n
  real(kind=rk), intent(inout) :: x(n)
  real(kind=rk), intent(inout) :: y(0:n,0:n)

  x=2.0*x
  y=2.0*y
  return
  end subroutine
  ```

- And call

  ```fortran
  call sub(x,y,n)
  ```

- Just as in C it is advisable to use function prototypes to get rid of errors in parameters, in Fortran one can use interface blocks in those program units where the procedure is called:

  ```fortran
  interface
  function area(r)
  implicit none
  integer, parameter :: rk=selected_real_kind(10,40)
  real(kind=rk) :: area
  real(kind=rk), intent(in) :: r
  end function area
  subroutine sub(x,y)
  implicit none
  real :: x(:), y(:,:)
  end subroutine sub
  end interface
  ```

- Each interface body consists of the procedure definition without the actual statements; see on the right.

Introduction to Fortran

- An array of a character string as a parameter of a procedure can be of assumed length or shape.

  - In this case the procedure interface has to be written out explicitly.

```
program assumedtest
  implicit none
  interface
    subroutine sub(x,y)
      implicit none
      real :: x(:), y(:,:)
      print *, x, y
    end subroutine sub
  end interface
  real :: x(3), y(3,3)
  x = 1.0
  y = 2.0
  call sub(x,y)
  stop
end program assumedtest
```

```
interface
  function area(r)
  implicit none
  integer, parameter :: rk=selected_real_kind(10,40)
  real(kind=rk) :: area
  real(kind=rk), intent(in) :: r
  end function area
  subroutine sub(x,y,n)
  implicit none
  integer, parameter :: rk=selected_real_kind(10,40)
  integer, intent(in) :: n
  real(kind=rk), intent(inout) :: x(n)
  real(kind=rk), intent(inout) :: y(0:n,0:n)
  end subroutine sub
end interface
```

Personal opinion: The clearest way to pass arrays to subroutines is to pass also the sizes; for example:

```
call sub(x,y,n)
...`
```
Introduction to Fortran

- If you want to save the value of a local procedure variable between procedure calls you can use the (you guessed it!) `save` specifier.

- For example, your subroutine needs to do some initialization only once:

```fortran
subroutine sub(x,y,n)
  ...
  logical,save :: firsttime=.true.
  ...
  if (firsttime) then
    ! Initialization
    ...
    firsttime=.false.
  endif
  ! Normal execution
  ...
end subroutine sub
```

- Same as `static` in C.

Introduction to Fortran

- Argument association may be `positional` (normal way) or by `keywords`:

```fortran
program argassoc
  implicit none
  integer :: i,j
  i=1;j=2
  call sub(i,j) ! Positional
  write(0,*), i,j
  i=1;j=2
  call sub(arg1=i,arg2=j) ! By keyword
  write(0,*), i,j
  i=1;j=2
  call sub(arg1=j,arg2=i) ! By keyword
  write(0,*), i,j
  stop
end program argassoc

subroutine sub(arg1,arg2)
  implicit none
  integer,intent(inout) :: arg1,arg2
  arg1=arg1*arg2
  arg2=-arg1
  return
end subroutine sub
```

progs> gfortran argassoc.f90
progs> a.out
2  -2
2  -2
-2  2
Introduction to Fortran

• Modules

- Modules is one of the most handy features of F90+.
- It replaces the old Fortran **common** areas but it is not restricted to that.
- Modules can contain global variables, constants, commonly used procedures.
- Example:

```fortran
module stuff
  implicit none
  integer, parameter ::
    & rk = selected_real_kind(p=15,r=100)
  real(kind=rk) :: pi = 3.14159265358979
contains
  function area(r)
    real(kind=rk), intent(in) :: r
    real(kind=rk) :: area
    area = pi * r**2
  end function area
  subroutine readr(r)
    real(kind=rk), intent(out) :: r
    print *, "Give r"
    read *, r
  end subroutine readr
end module stuff
```

- You can only use a part of the module by giving the **only** qualifier:

  ```fortran
  use stuff, only : rk
  ```

- The name of the object in the module can also be changed:

  ```fortran
  use stuff, only : rk => real_type
  ```

• Main differences between Fortran and C

<table>
<thead>
<tr>
<th></th>
<th>Fortran</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array indexing</td>
<td>a()</td>
<td>a[]</td>
</tr>
<tr>
<td></td>
<td>starts from 1</td>
<td>starts from 0</td>
</tr>
<tr>
<td>Multidimensional</td>
<td>columnwise</td>
<td>rowwise</td>
</tr>
<tr>
<td>arrays stored</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Procedure parameters</td>
<td>address</td>
<td>value</td>
</tr>
</tbody>
</table>

- It is possible to use both Fortran and C in the same program but the Fortran-C interface is not (yet) standardized.
Introduction to Fortran

- **Intrinsic procedures**
  - Subroutines and functions that are a part of the language standard
  - Math functions
    - Generic and specific names; e.g. `sqrt()`, `dsqrt()`, `csqrt()`
    - Generic names: type of result depends on the argument:
      ```fortran
      program genericfunctionname
          implicit none
          integer, parameter :: r1 = selected_real_kind(5, 10)
          integer, parameter :: r2 = selected_real_kind(10, 40)
          print *, sqrt(2.0_r1)
          print *, sqrt(2.0_r2)
          print *, kind(sqrt(2.0_r1))
          print *, kind(sqrt(2.0_r2))
      stop
      end program genericfunctionname
      ``
      - No reason to use specific names!
    - The most common math functions included.

- What's going on here?

```fortran
program pi_accuracy
    integer, parameter :: rk0 = selected_real_kind(5, 10)
    integer, parameter :: rk1 = selected_real_kind(10, 40)
    integer, parameter :: rk2 = selected_real_kind(30, 200)
    real(rk2) :: pi0
    real(rk2) :: pi1
    real(rk2) :: pi2
    pi0 = 4.0*atan2(1.0_rk0, 1.0_rk0)
    pi1 = 4.0*atan2(1.0_rk1, 1.0_rk1)
    pi2 = 4.0*atan2(1.0_rk2, 1.0_rk2)
    print *, pi0
    print *, pi1
    print *, pi2
stop
end program pi_accuracy
```

- Underscore notation for giving the kind parameter of a literal constant.
Introduction to Fortran

Below is a list of the standard intrinsic functions (from Intel Fortran compiler Language Reference Manual, /opt/intel_fc_80/doc/for_lang.pdf):

<table>
<thead>
<tr>
<th>Function name</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS(a)</td>
<td>E</td>
<td>The absolute value of an argument</td>
</tr>
<tr>
<td>ACHAR(i)</td>
<td>E</td>
<td>The character in the specified position of the ASCII character set</td>
</tr>
<tr>
<td>ACOS(x)</td>
<td>E</td>
<td>The arccosine (in radians) of the argument</td>
</tr>
<tr>
<td>ADJUSTL(string)</td>
<td>E</td>
<td>The specified string with leading blanks removed and placed at the end of the string</td>
</tr>
<tr>
<td>ADJUSTR(string)</td>
<td>E</td>
<td>The specified string with trailing blanks removed and placed at the beginning of the string</td>
</tr>
<tr>
<td>AIMAG(z)</td>
<td>E</td>
<td>The imaginary part of a complex argument</td>
</tr>
<tr>
<td>AINT(a[,kind])</td>
<td>E</td>
<td>A real value truncated to a whole number</td>
</tr>
<tr>
<td>ALLOCATED(array)</td>
<td>I</td>
<td>The allocation status of the argument array</td>
</tr>
<tr>
<td>ALL(mask[,dim])</td>
<td>T</td>
<td>.TRUE. if all elements of the masked array are true</td>
</tr>
<tr>
<td>ANY(mask[,dim])</td>
<td>T</td>
<td>.TRUE. if any elements of the masked array are true</td>
</tr>
<tr>
<td>ANINT(a[,kind])</td>
<td>E</td>
<td>A real value rounded to a whole number</td>
</tr>
<tr>
<td>ASIN(x)</td>
<td>E</td>
<td>The arcsine (in radians) of the argument</td>
</tr>
<tr>
<td>ASSOCIATED(pointer[,target])</td>
<td>I</td>
<td>.TRUE. if the pointer argument is associated or the pointer is associated with the specified target</td>
</tr>
<tr>
<td>ATAN(x)</td>
<td>E</td>
<td>The arctangent (in radians) of the argument</td>
</tr>
<tr>
<td>ATAN2(y,x)</td>
<td>E</td>
<td>The arctangent (in radians) of the arguments</td>
</tr>
<tr>
<td>BIT_SIZE(i)</td>
<td>I</td>
<td>The number of bits (s) in the bit model</td>
</tr>
<tr>
<td>BTEST(i,pos)</td>
<td>E</td>
<td>.TRUE. if the specified position of argument i is one</td>
</tr>
<tr>
<td>CEILING(a[,kind])</td>
<td>E</td>
<td>The smallest integer greater than or equal to the argument value</td>
</tr>
<tr>
<td>CHAR(i[,kind])</td>
<td>E</td>
<td>The character in the specified position of the processor character set</td>
</tr>
<tr>
<td>CONJG(z)</td>
<td>E</td>
<td>The conjugate of a complex number</td>
</tr>
<tr>
<td>COS(x)</td>
<td>E</td>
<td>The cosine of the argument, which is in radians</td>
</tr>
<tr>
<td>COSH(x)</td>
<td>E</td>
<td>The hyperbolic cosine of the argument</td>
</tr>
<tr>
<td>COUNT(mask[,dim][,kind])</td>
<td>T</td>
<td>The number of .TRUE. elements in the argument array</td>
</tr>
<tr>
<td>CSIZE(array,shift[,dim])</td>
<td>T</td>
<td>An array that has the elements of the argument array circularly shifted</td>
</tr>
<tr>
<td>DBLE(a)</td>
<td>E</td>
<td>The corresponding double precision value of the argument integer argument</td>
</tr>
<tr>
<td>DIGITS(x)</td>
<td>I</td>
<td>The number of significant digits in the model for the argument</td>
</tr>
<tr>
<td>DIM(x,y)</td>
<td>E</td>
<td>The positive difference between the two arguments</td>
</tr>
<tr>
<td>DOT_PRODUCT(vector_a,vector_b)</td>
<td>T</td>
<td>The dot product of two rank-one arrays (also called a vector multiply function)</td>
</tr>
<tr>
<td>EOSHIFT(array,shift[,boundary][,dim])</td>
<td>T</td>
<td>An array that has the elements of the argument array end-off shifted</td>
</tr>
<tr>
<td>EPSILON(x)</td>
<td>I</td>
<td>The number that is almost negligible when compared to one</td>
</tr>
<tr>
<td>EXP(x)</td>
<td>E</td>
<td>The exponential ex for the argument x</td>
</tr>
<tr>
<td>EXPONENT(x)</td>
<td>E</td>
<td>The value of the exponent part of a real argument</td>
</tr>
<tr>
<td>FLOOR(a[,kind])</td>
<td>E</td>
<td>The largest integer less than or equal to the argument value</td>
</tr>
<tr>
<td>FRACTION(x)</td>
<td>E</td>
<td>The fractional part of a real argument</td>
</tr>
<tr>
<td>HUGE(x)</td>
<td>I</td>
<td>The largest number in the model for the argument</td>
</tr>
<tr>
<td>IACHAR(c)</td>
<td>E</td>
<td>The position of the specified character in the ASCII character set</td>
</tr>
<tr>
<td>IAND(i,j)</td>
<td>E</td>
<td>The logical AND of the two arguments</td>
</tr>
<tr>
<td>IBCLR(pos)</td>
<td>E</td>
<td>The specified position of argument I cleared (set to zero)</td>
</tr>
<tr>
<td>Function name</td>
<td>Class</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IBITS(i,pos,len)</td>
<td>E</td>
<td>The specified substring of bits of argument I</td>
</tr>
<tr>
<td>IBSET(i,pos)</td>
<td>E</td>
<td>The specified bit in argument I set to one</td>
</tr>
<tr>
<td>ICHAR(c)</td>
<td>E</td>
<td>The position of the specified character in the processor character set</td>
</tr>
<tr>
<td>Ieor(i,j)</td>
<td>E</td>
<td>The logical exclusive OR of the corresponding bit arguments</td>
</tr>
<tr>
<td>INDEX(string,substring[,back][,kind])</td>
<td>E</td>
<td>The position of the specified substring in a character expression</td>
</tr>
<tr>
<td>INT(a[,kind])</td>
<td>E</td>
<td>The corresponding integer value(truncated) of the argument</td>
</tr>
<tr>
<td>IOR(i,j)</td>
<td>E</td>
<td>The logical inclusive OR of the corresponding bit arguments</td>
</tr>
<tr>
<td>ISHFT(i,shift)</td>
<td>E</td>
<td>The logical end-off shift of the bits in argument I</td>
</tr>
<tr>
<td>ISHFTC(i,shift,[size])</td>
<td>E</td>
<td>The logical circular shift of the bits in argument I</td>
</tr>
<tr>
<td>KIND(x)</td>
<td>I</td>
<td>The kind type parameter of the argument</td>
</tr>
<tr>
<td>LBOUND(array[,dim][,kind])</td>
<td>I</td>
<td>The lower bounds of an array(or one of its dimensions)</td>
</tr>
<tr>
<td>LEN(string[,kind])</td>
<td>I</td>
<td>The length(number of characters) of the argument character string</td>
</tr>
<tr>
<td>LEN_TRIM(string[,kind])</td>
<td>E</td>
<td>The length of the specified string without trailing blanks</td>
</tr>
<tr>
<td>LGE(string_a,string_b)</td>
<td>E</td>
<td>A logical value determined by a &gt; or = comparison of the arguments</td>
</tr>
<tr>
<td>LGT(string_a,string_b)</td>
<td>E</td>
<td>A logical value determined by a &gt; comparison of the arguments</td>
</tr>
<tr>
<td>LLE(string_a,string_b)</td>
<td>E</td>
<td>A logical value determined by a &lt; or = comparison of the arguments</td>
</tr>
<tr>
<td>LLT(string_a,string_b)</td>
<td>E</td>
<td>A logical value determined by a &lt; comparison of the arguments</td>
</tr>
<tr>
<td>LOG(x)</td>
<td>E</td>
<td>The natural logarithm of the argument</td>
</tr>
<tr>
<td>LOG10(x)</td>
<td>E</td>
<td>The common logarithm(base 10) of the argument</td>
</tr>
<tr>
<td>LOGICAL(l[,kind])</td>
<td>E</td>
<td>The logical value of the argument converted to a logical of type KIND</td>
</tr>
<tr>
<td>MATMUL(matrix_a,matrix_b)</td>
<td>T</td>
<td>The result of matrix multiplication(also called a matrix multiply function)</td>
</tr>
<tr>
<td>MAX(a1,a2[,a3,...])</td>
<td></td>
<td>The maximum value in the set of arguments</td>
</tr>
<tr>
<td>MAXEXPONENT(x)</td>
<td>I</td>
<td>The maximum exponent in the model for the argument</td>
</tr>
<tr>
<td>MAXLOC(array[,dim][,mask][,kind])</td>
<td>T</td>
<td>The rank-one array that has the location of the maximum element in the argument array</td>
</tr>
<tr>
<td>MAXVAL(array[,dim][,mask])</td>
<td>T</td>
<td>The maximum value of the elements in the argument array</td>
</tr>
<tr>
<td>MERGE(tsource,fsource,mask)</td>
<td>E</td>
<td>An array that is the combination of two conformable arrays(under a mask)</td>
</tr>
<tr>
<td>MIN(a1,a2[,a3,...])</td>
<td>E</td>
<td>The minimum value in the set of arguments</td>
</tr>
<tr>
<td>MINEXPONENT(x)</td>
<td>I</td>
<td>The minimum exponent in the model for the argument</td>
</tr>
<tr>
<td>MINLOC(array[,dim][,mask][,kind])</td>
<td>T</td>
<td>The rank-one array that has the location of the minimum element in the argument array</td>
</tr>
<tr>
<td>MINVAL(array[,dim][,mask])</td>
<td>T</td>
<td>The minimum value of the elements in the argument array</td>
</tr>
<tr>
<td>MOD(a,p)</td>
<td>E</td>
<td>The remainder of the arguments(has the sign of the first argument)</td>
</tr>
<tr>
<td>MODULO(a,p)</td>
<td>E</td>
<td>The modulo of the arguments(has the sign of the second argument)</td>
</tr>
<tr>
<td>NEAREST(x,s)</td>
<td>E</td>
<td>The nearest different machine-representable number in a given direction</td>
</tr>
<tr>
<td>NINT(a[,kind])</td>
<td>E</td>
<td>A real value rounded to the nearest integer</td>
</tr>
<tr>
<td>NOT(i)</td>
<td>E</td>
<td>The logical complement of the argument</td>
</tr>
<tr>
<td>NULL([mold])</td>
<td>T</td>
<td>A disassociated pointer</td>
</tr>
<tr>
<td>PACK(array,mask[,vector])</td>
<td>T</td>
<td>A packed array of rank one(under a mask)</td>
</tr>
<tr>
<td>PRECISION(x)</td>
<td>I</td>
<td>The decimal precision(real or complex) of the argument</td>
</tr>
<tr>
<td>PRESENT(a)</td>
<td>I</td>
<td>TRUE. if an actual argument has been provided for an optional dummy argument</td>
</tr>
<tr>
<td>PRODUCT(array[,dim][,mask])</td>
<td>T</td>
<td>The product of the elements of the argument array</td>
</tr>
<tr>
<td>RADIX(x)</td>
<td>I</td>
<td>The base of the model for the argument</td>
</tr>
<tr>
<td>RANGE(x)</td>
<td>I</td>
<td>The decimal exponent range of the model for the argument</td>
</tr>
</tbody>
</table>
### Function name

<table>
<thead>
<tr>
<th>Function name</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL(a[,kind])</td>
<td>E</td>
<td>The corresponding real value of the argument</td>
</tr>
<tr>
<td>REPEAT(string,ncopies)</td>
<td>T</td>
<td>The concatenation of zero or more copies of the specified string</td>
</tr>
<tr>
<td>RESHAPE(source,shape[,pad][,order])</td>
<td>T</td>
<td>An array that has a different shape than the argument array, but the same elements</td>
</tr>
<tr>
<td>RRSPACING(x)</td>
<td>E</td>
<td>The reciprocal of the relative spacing near the argument</td>
</tr>
<tr>
<td>SCALE(x,i)</td>
<td>E</td>
<td>The value of the exponent part(of the model for the argument) changed by a specified value</td>
</tr>
<tr>
<td>SCAN(string,SET[,back][,kind])</td>
<td>E</td>
<td>The position of the specified character(or set of characters) within a string</td>
</tr>
<tr>
<td>SELECTED_INT_KIND(r)</td>
<td>T</td>
<td>The integer kind parameter of the argument</td>
</tr>
<tr>
<td>SELECTED_REAL_KIND([p][,r])</td>
<td>T</td>
<td>The real kind parameter of the argument; one of the optional arguments must be specified</td>
</tr>
<tr>
<td>SET_EXPONENT(x,i)</td>
<td>E</td>
<td>The value of the exponent part(of the model for the argument) set to a specified value</td>
</tr>
<tr>
<td>SHAPE(SOURCE[,kind])</td>
<td>I</td>
<td>The shape(rank and extents) of an array or scalar</td>
</tr>
<tr>
<td>SIGN(a,b)</td>
<td>E</td>
<td>A value with the sign transferred from its second argument</td>
</tr>
<tr>
<td>SIN(x)</td>
<td>E</td>
<td>The sine of the argument, which is in radians</td>
</tr>
<tr>
<td>SINH(x)</td>
<td>E</td>
<td>The hyperbolic sine of the argument</td>
</tr>
<tr>
<td>SIZE(array[,dim][,kind])</td>
<td>I</td>
<td>The size (total number of elements) of the argument array(or one of its dimensions)</td>
</tr>
<tr>
<td>SPACING(x)</td>
<td>E</td>
<td>The value of the absolute spacing of model numbers near the argument</td>
</tr>
<tr>
<td>SPREAD(source,dim,ncopies)</td>
<td>T</td>
<td>A replicated array that has an added dimension</td>
</tr>
<tr>
<td>SORT(x)</td>
<td>E</td>
<td>The square root of the argument</td>
</tr>
<tr>
<td>SUM(array[,dim][,mask])</td>
<td>T</td>
<td>The sum of the elements of the argument array</td>
</tr>
<tr>
<td>TAN(x)</td>
<td>E</td>
<td>The tangent of the argument, which is in radians</td>
</tr>
</tbody>
</table>

### Subroutine name

<table>
<thead>
<tr>
<th>Subroutine name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU_TIME(time)</td>
<td>The processor time in seconds</td>
</tr>
<tr>
<td>DATE_AND_TIME([date][,time][,zone][,values])</td>
<td>Date and time information from the real-time clock</td>
</tr>
<tr>
<td>MVBITS(from,frompos,len,to,topos)</td>
<td>A sequence of bits(bit field) is copied from one location to another</td>
</tr>
<tr>
<td>RANDOM_NUMBER(harvest)</td>
<td>A pseudorandom number taken from a sequence of pseudorandom numbers uniformly distributed within the range 0.0 to 1.0</td>
</tr>
<tr>
<td>RANDOM_SEED([size][,put][,get])</td>
<td>Initializes or retrieves the pseudorandom number generator seed value</td>
</tr>
<tr>
<td>SYSTEM_CLOCK([count][,count_rate][,count_max])</td>
<td>Data from the processors real-time clock</td>
</tr>
</tbody>
</table>

- Below is a list of the standard intrinsic subroutines (from Intel Fortran compiler Language Reference Manual, /opt/intel_fc_80/doc/for_lang.pdf):
Introduction to Fortran

- Inquiry functions include functions describing the floating point system used:

```fortran
program fpinquiryfunctions
  implicit none
  integer, parameter ::
    r1=selected_real_kind(5,10)
    r2=selected_real_kind(10,40)
    r3=selected_real_kind(30,200)
  real(kind=r1) :: x
  real(kind=r2) :: y
  real(kind=r3) :: z
  print *,digits(x),digits(y),digits(z)
  print *,epsilon(x),epsilon(y),epsilon(z)
  print *,huge(x),huge(y),huge(z)
end program fpinquiryfunctions
```

```
progs> ifort fpinquiryfunctions.f90
progs> a.out
24          53         113
1.1920929E-07  2.220446049250313E-016
1.925929944387235853055977942584927E-0034
3.4028235E+38
```

- Also inquiries about the integer system:

```fortran
program bitsizeexample
  integer, parameter ::
    i1=selected_int_kind(3)
    i2=selected_int_kind(6)
    i3=selected_int_kind(12)
  integer(kind=i1) :: i
  integer(kind=i2) :: j
  print *,i,j
end program bitsizeexample
```

```
progs> ifort bitsizeexample.f90
progs> a.out
16          32                    64
```

Introduction to Fortran

- Functions for array handling (one of the greatest features of F90+):

```fortran
program anyexample
  implicit none
  integer :: i,j,a(10)
  do i=1,10
    a(i)=i
  end do
  print '(10(i0,x))',a
  if (any(a(5:10)>9)) print '(a)','yes'
  if (all(a(5:10)>9)) print '(a)','yes'
  print '(10l1)',a(5:10)>9
end program anyexample
```

```
progs> ifort anyexample.f90
progs> ./a.out
1 2 3 4 5 6 7 8 9 10
yes
```

```fortran
program dotproductexample
  implicit none
  integer, parameter :: &
    rk=selected_real_kind(10,40)
  real(kind=rk) :: x(5)=[1.0,1.0,2.0,4.0,7.0]
  real(kind=rk) :: y(5)=[2.0,3.0,1.0,0.0,5.0]
  print '(100f5.2)',x
  print '(100f5.2)',y
  print '(100f5.2)',dot_product(x,y)
end program dotproductexample
```

```
progs> ifort dotproductexample.f90
progs> ./a.out
1.00 1.00 2.00 4.00 7.00
2.00 3.00 1.00 0.00 5.00
42.00
```

Note the array initialization syntax.
## Introduction to Fortran

### More examples of array functions

```fortran
program matmulexample
  implicit none
  integer, parameter :: rk = selected_real_kind(10, 40)
  real(kind=rk) :: x(3,2), y(2,3), z(3,3)
  character(len=80) :: fm = '(a,x,100f6.2)'

  x = reshape([1.0, 2.0, 4.0, 5.0, 9.0, 1.0], [3, 2])
  y = reshape([7.0, 4.0, 4.0, 6.0, 1.0, 0.0], [2, 3])
  print fm, 'x', x
  print *
  print fm, 'y', y
  z = matmul(x, y)
  print fm, 'z', z
  print *, maxloc(z), maxval(z)
stop
end program matmulexample
```

Note the use of a string variable as the format specifier.

### Examples of string handling functions:

```fortran
program stringhandling
  character(len=20) :: s1
  read *, s1
  print *, s1
  print *, trim(s1)
  print *, len(s1), len_trim(s1)
stop
end program stringhandling
```

### The two time-related routines are `date_and_time()` and `cpu_time()`:

```fortran
program systemclock
  implicit none
  integer :: count, count_rate, count_max
  character (len=20) :: date, time, zone
  integer :: values(8)
  call date_and_time(date, time, zone, values)
  write(0,'(5a)') trim(date), ' ', trim(time), ' ', trim(zone)
  write(0,'(8(i0,x)/)') values
stop
end program systemclock
```

```fortran
progs> ifort matmulexample.f90
progs> ./a.out
x  1.00  2.00  4.00  5.00  9.00  1.00
y  7.00  4.00  4.00  6.00  1.00  0.00
z 27.00 50.00 32.00 34.00 62.00 22.00 1.00 2.00 4.00
max 0.00 0.00 62.00

Note the use of a string variable as the format specifier.

```fortran
progs> ifort stringhandling.f90
progs> a.out
aaabbbccc
|aaabbbccc|
20 9
```

```fortran
progs> ifort systemclock.f90
progs> ./a.out
20110927 153124.451 +0300
2011 9 27 180 15 31 24 451
values(1): year
values(2): month
values(3): day of the month
values(4): time difference with UTC in minutes
values(5): hour of the day
values(6): minutes of the hour
values(7): seconds of the minute
values(8): milliseconds of the second
```
Introduction to Fortran

- When optimizing programs it is essential to measure the CPU time used:

```fortran
program cputime
  implicit none
  integer, parameter :: rk=selected_real_kind(10,40), NMAX=10000000
  real(kind=rk) :: x, t1, t2
  integer :: n
  x=0.0
  call cpu_time(t1)
  do n=NMAX,1,-1
    x=x+1.0_rk/real(n,rk)
  end do
  call cpu_time(t2)
  print *,'Result ',x
  print *,'CPU time in seconds ', t2-t1
stop
end program cputime
```

progs> ifort cputime.f90
progs> a.out
Result    18.9978964138534
CPU time in seconds    1.63975200000000

F90+ has two constructs that helps the handling of arrays.

- With the `where` construct you can modify arrays without using explicit do-loops:

```fortran
program whereexample
  implicit none
  integer, parameter :: rk=selected_real_kind(10,40), i1=1, i2=10
  real(kind=rk), dimension(i1:i2) :: x, y
  integer :: n
  do n=i1,i2
    x(n)=n-i2/2
  end do
  where (x<0.0)
    y=-10.0
  else where
    y=0.0
  end where
  print '(10f6.1)', x
  print '(10f6.1)', y
stop
end program whereexample
```

progs> ifort whereexample.f90
progs> ./a.out

-4.0  -3.0  -2.0  -1.0   0.0   1.0   2.0   3.0   4.0   5.0
-10.0 -10.0 -10.0 -10.0   0.0   0.0   0.0   0.0   0.0   0.0
Introduction to Fortran

- In the `where` construct the mask expression can be any logical expression that has the same shape as the left hand sides of the assignments inside the construct:

```fortran
program whereexample1
  implicit none
  integer,parameter :: &
    & rk=selected_real_kind(10,40),i1=1,i2=10
  real(kind=rk),dimension(i1:i2) :: x
  logical :: mask(i1:i2)
  integer :: n
  x=1.0
  mask=.false.
  mask(i1:i2:2)=.true.
  where (mask)
    x=-1.0
  end where
  print '(50l6)',mask
  print '(50f6.1)',x
  stop
end program whereexample1
```

```bash
progs> ifort whereexample1.f90
progs> ./a.out
T     F     T     F     T     F     T     F     T     F
-1.0   1.0  -1.0   1.0  -1.0   1.0  -1.0   1.0  -1.0   1.0
```

Introduction to Fortran

- Fortran95 added the `forall` construct to the language.

- In a way it is a do-loop or an array construct where the order of iteration does not matter; i.e. it can be parallelized:

```fortran
program forallexample
  implicit none
  integer,parameter :: &
    & rk=selected_real_kind(10,40)
  integer :: a(10,10)
  integer :: i,j
  a=0
  forall (i=1:10:2,j=1:10:2,i+j>10)
    a(i,j)=(i+j)/2
  end forall
  print '(10i4)',a
  stop
end program forallexample
```

```bash
progs> ifort forallexample.f90
progs> a.out
  T     F     T     F     T     F     T     F     T     F
  -1.0   1.0  -1.0   1.0  -1.0   1.0  -1.0   1.0  -1.0   1.0
```