**Fortran 2008 Quick Reference**

1. **Program Structure**

**module** m\_interface

use, intrinsic :: iso\_fortran\_env

implicit none

integer , parameter :: kd = real64

real(kd), parameter :: pi = 4 \* atan(1.0\_kd)

interface

module pure elemental real(kd) function f(x)

real(kd), intent(in) :: x

end function f

end interface

**end module** m\_interface

**submodule** (m\_interface) m\_implemantation

contains

module procedure f

f = sin(2\*pi\*x)

end procedure f

**end submodule** m\_implemantation

**module** m\_mod

use m\_interface

implicit none

contains

impure elemental subroutine g(x)

real(kd), value, optional :: x

if (.not.present(x)) x = pi

print \*, f(x) - h(x)

contains ! internal subprogram

pure real(kd) function h(x)

real(kd), intent(in) :: x

h = 2 \* sin(pi \* x) \* cos(pi \* x)

end function h

end subroutine g

**end module** m\_mod

**program** test01

use m\_mod, only: kd, o => g

implicit none

real(kd) :: x, y

read \*, x

call o([x, 2\*x, 3\*x])

**end program** test01

1. **Subprograms** f95~ deallocs allocatables at exit

**subroutine** 　 pointers are not dealloced at exit

**pure elemental subroutine** s(a, x, y) ! f95 elemental

real, intent(in), value :: a; real, intent(in) :: x

real, intent(in out) :: y

y = a \* x + y

**end subroutine** s

call s(a, x, y);call s(a(1), x, y);call s(a, x(1), y)!a,x,y arr

**function** (functions should be pure!)

**elemental pure real function** f(x, a)

real, intent(in) :: x ; real, intent(in), optional :: a

if (present(a)) then f = sin(a\*x); else; f = sin(x); endif

**end function** f

**recursive pure function** f(i) **result**(ires)

integer, intent(in) :: i; integer::ires

if (i==0) then; ires=1;else; ires = i \* f(i – 1);endif

**end function** f ! factorial

**pure function** f(x) ; real, allocatable :: f(:)

real, intent(in), contiguous::x(:) ! f08 optimize info.

f = [0.0, x] + [x, 0.0] ! [x(1), x(1)+x(2),…]

**end function** f !x=[1];do;x=f(x);end do;Pascal’s triangle

**function storage**(key) result(loc) ;integer, pointer::loc

integer, intent(in) :: key; integer,target :: val(100)=0

loc => m(key) ! storage(i) = 2\*i+1; print \*, storage(i)

**end function** storage

**array-valued function** ! stack / heap

function f(x); real:: f(size(x)) / real, allocatable::f(:) …

1. **Interface Block**

**interface** gamma ! generic name

[module] procedure [::] s\_gamma, d\_gamma

**end interface**

**unary** (.op.,+,-) **and binary** (.op.,+,-,\*,/,//) **operators**

**interface operator** (.op.) ! unary: .op. x, binary: x .op. y

[module] procedure [::] func ! unaryf(x), binary f(x,y)

**end interface** ! intent(in) :: x, y

**interface assignment** (=) ! y = x 🡨🡪 subroutine(y, x)

[module]procedure[::]sub !intent([in]out)::y;intent(in)::x

**end interface**

**abstract interface**

pure real function t\_f(x); real,intent(in)::x;

end function t\_f

subroutine t\_noargs(); end subroutine t\_noargs

**end interface**

procedure (t\_f) :: f0;procedure (t\_noargs), pointer :: g0

procedure (real(8)), pointer::f; f=>cos; f(pi) !🡪 -1.0d0

procedure (sin), pointer::f; f=>cos ! intrinsic sub/fun

1. **Control Structures**

label: **block !**(block,associate,do,if,select case/type)

**exit** label

**end block** label

**Loop**

**do** i = i0, i1 [, i2] ! i = i0, i0+i2,…,i0+ i2\* ((i1- i0)/i2)

**…**

**end do** ! i =i0+ i2\* ((i1-i0+i2) / i2)

**do while** (logical)

**…**

**end do**

outer: **do**

inner: **do**

if (logical) cycle inner

if (logical) exit outer

**end do**

**end do**

**do concurrent** (integer::i=1:n, j=1:n, i > j) ! local i,j

**block** ! no order in sequences

real x, y ! thread local x,y

x = a \* i + b; y = c \* j + d

z(i, j) = f(x, y)

**end block**

**end do**

**Branch**

**if** (logical) … ! simple if

**if** (logical) **then** ! block if

**…**

**else if** (logical) **then**

**…**

**else**

**…**

**end if**

**select case** (x) ! discrete value int/char/(logical)

**case** (:0) ; …

**case** (1,3,5:) ; …

**case** **default** ; …

**end select**

**select type** (t) ! type/class

**type is** (real) ; …

**type is** (real(kd)) ; …

**class is** (t\_mytype) ; …

**class default** ; …

**end select**

1. **Array**

**Array functions**

**reduction functions / mask**

all(m), any(m), parity(m), count(m) ! mask = a > 0.0

sum(a, m), product(a, m)

iall(a,m), iany(a,m), iparity(a,m), popcnt(i) ! bitwise

**location** minloc(a)→array[i],miloc(a,dim=1)→scalar i

minloc(a,m), maxloc(a,m), minval(a,m), maxval(a,m)

findloc(a,key[,dim][,m][,kind][,back]) ! 0 if not found

**filter** pack/unpack

x = pack(a, mask a > 0.0) ! filter a > 0.0

a = unpack( elem\_func( pack(a, mask) ), mask)

**inquiry** ! [,d,k]=[,dim][,kind]

shape(a), size(a,[d,k]), lbound(a[,d,k]), ubound(a,[,d,k])

allocated(a)

**array operation**

reshape(src, shape [,pad][,order])

merge(a\_true, a\_false, mask), spread(src, dim, ncopy)

cshift(a, shift [,dim]) ! cyclic +shift ←,↑

eoshift(a, shift [,boundary][,dim]) ! end-off pad

transpose(a)

**Array assignments**

real, allocatable :: a(:), b(:)

a = [1.0,2.0,3.0] ; b = [(i, i = 1, 10)] ! allocation by assign

a = b ! reallocated a(10)

a(:) = b ! not-reallocated a(3)

call move\_alloc(from, to)

allocate(a(0)); allocate(a, source=[real::]); a=[real::]

a = [0.0, b]; a = [a, [11.0, 12.0]] ! a→0.0..12.0

a = a(13:1:-1) ! a→12.0..0.0

!do i = 1,13; a(i) = a(14 - i); end do ! a→12.0..6.0..12.0

Integer::fib(2) = [0,1];do; fib=[fib(2),fib(1)+fib(2)]; end do

integer :: indx(3) = [1,3,2]

a(indx) = a; a = a(indx); a(indx(3:1:-1))=a(indx)

a = sin(a) ! elemental function :a = map sin a

**where** (a > 0.0) ! value dependent assignment

a = sqrt(a)

**elsewhere**

a = 0.0

**end wherer**

**forall** (integer::k=1:13, a(k) > 0.0) ! index dependent

a(k) = sqrt(k \* a(k)) ! assignment

b(k) = a(k) / k

**end** forall

**forall** (i=1:n, j=1:n) b(i, j) = 1.0 / (i + j)

1. **Data Types**

|  |  |  |
| --- | --- | --- |
| **logical** | .true. .false. | .eqv., .neqv. |
| **integer** | huge ~2e9=2G, ~9d18 | |
| **real** | epsilon ~e-7, ~d-16, ~q-34 | |
| **complex** | complex\*16 = complex(8) | |
| **character** (len=:), allocatable :: text(:) | | |

use, intrinsic::iso\_fortran\_env   
 int8, int16, int32, int64, real32, real64, real128

integer, parameter :: ks = kind(0.0), kd = kind(0.0d0)

selected\_int\_kind(k),selected\_real\_kind(i,k)! [0..i]\*Ek

real(kd), parameter :: pi=4\*atan(1.0\_kd)

integer :: n = 0 ! implicitly **save** attribute

integer :: i = 11 ! B’1011’, O’13’, Z’0B’ (Bin Oct Hex)

complex(ks)::c=(0.1d0, 0.1d0), d=cmplx(1.0)

complex(kd)::c=cmplx(0.1d0, 0.1d0, **kind=kd**)

cmplx(a[,b],kind) type conversion

print \*, conjg(c), real(c), imag(c), c%re, c%im

**character type**  ~ ^ \ { } ` [ ] | # @ !*f03*

character, parameter(len = \*) :: o = ‘fortran’(2:4) ! ‘ort’

character(len=5)::text=repeat(‘X’, 3) ! ‘XXX ’ ! pad ’ ‘

character [(len=1)] :: char(5) = ‘x’ ! [‘x’, ‘x’, ‘x’, ‘x’, ‘x’]

text = ‘ab’ // ”c“ !🡪 ‘abc ‘ ; c\_arr(2:4)(3:4)! indx,pos

print \*,len(text), text%len, len\_trim(text), trim(text)

adjustl(‘ abc’) !🡪’abc ‘ ; adjustr(‘abc ‘) !🡪’ abc’

char = transfer(text, ‘ ‘, size(char)) ! [‘a’,’b’,’c’,’ ‘,’ ‘]

text = transfer(char, text) ! string 🡨🡪 array

index(text, str [,back=.true.] [,kind]) ! not found=0

scan(text, set [,back=.true.] [,kind]) ! not in set=0

verify((text, set [,back=.true.] [,kind])) ! all in set=0

print \*,‘A’<‘a’, ‘ab’>=‘abc’ !T,F;ASCII lexical;pad space

**deferred/assumed length**

character (len=:), allocatable :: text ! *f03*

text = ‘abc’ ;text= text // ‘def’ ! variable length

character(:), allocatable :: text(:) ! var.len.array

character (len=\*), intent(in) :: text ! subprogram arg.

**internal file** character 🡨🡪 number

read(text, ‘(3i4)’) i, j, k ! string🡪 num

write(text,’(3f5.1)’) x, y, z ! num 🡪 string

**character code**

iachar(c),achar(i) !ASCII; ichar(c),char(i) !EBCDIC etc

character(len=\*), parameter :: txt = ‘abcd’ ! → ‘ABCD’

transfer(achar(iachar(transfer(txt,' ', txt%len))-32), txt)

1. **Derived Types**

type :: t\_base

integer :: i

end type t\_base

**type extension**

type, extends(t\_base) :: t\_type

real :: x = 1.0, y = 2.0

end type t\_type

type (t\_type) :: a = t\_type(0,2.0,4.0) !default constructor

**interface** t\_type! user-defined constructor

[module] procedure [::] fun !

**end interface** ! type(t\_type) function fun(i, x, y) …

same\_type\_as(a, b),extends\_type\_of(a, mold)

**parameterized derived type**

type :: t\_type(knd, len)

integer, kind :: knd = kind(0.0d0)

integer, len :: len = 10

real (knd) :: x(len)

end type t\_ptype

type(t\_type(knd=4)) :: a = t\_type(knd=4)(0.0)

type(t\_type(kind(0.0), 5), target :: a

type(t\_type(kind(0.0), : ), pointer :: p => a

type(t\_type(kind(0.0), \*), intent(in) :: a

1. **Non-default Derived Type I/O**

**type :: t\_dt**

integer :: i, j

**contains**  ! type-bound procedure

procedure :: wr

generic :: write(formatted) => wr

**end type t\_dt**

**subroutine wr(dtv, unit, iotype, vlist, io, iomsg)**

class(t\_dt), intent(in)::dtv; integer, intent(in)::unit

character(\*), intent(in)::iotype; integer,intent(in)::vlist(:)

integer, intent(out)::io; character(\*), intent(in out):: iomsg; character(20)::fmt

if (**iotype = ‘LISTDIRECTED’**) then

write(unit,\*,iostat=io) dtv%i, dtv%j

else if (**iotype == ‘DTtest’**) then ! ‘DT[name]’

write(fmt,’(a,g0,a,g0,a)’,iostat=io) ‘(i’,vlist(1),’,i’,vlist(2),’)’

write(unit, fmt) dtv%i, dtv%j ! else for error

end if

**end subroutine wr**

type (t\_dt) :: d = t\_test(1, 2)

print \*, d ! list directed

print ‘(**DT”test”(8,8)**)’, d ! DT[’name’] formatted

1. **Type-bound Procedures**

**static binding**

module m\_mod ! type(t\_type), allocatable::a

type :: t\_type ! print \*, a%f();call a%f3(1.0)

real :: x ! deallocate(a)

contains

procedure :: f1, f2 ; procedure, nopass :: f3

generic :: f => f1, f2; generic :: operator(+) => f1, f2

final :: fin ! finalizer

end type t\_type

contains

pure real function f1(a)

class(t\_type), intent(in)::a

end function f1 …

subroutine fin(a)

type(t\_type), intent(in out) :: a

end subroutine fin

end module m\_mod

**deferred binding**

type, abstract :: t\_base

contains

procedure(p), deferred :: pf

end type t\_base

abstract interface

subroutine p(this)

import; class(t\_base), intent(in out) :: this

end subroutine p

end interface

type, extends(t\_base) :: t\_type

real :: x

contains

procedure :: pf => my\_pf

end type t\_type

contains

subroutine my\_pf(this)

class(t\_type), intent(in out) :: this

end subroutine my\_pf

**dynamic binding** (non-extendable: bind(c), sequence)

type, bind(c) :: t\_type

real::x, y; procedure(p\_f), pointer :: fun => null()

end type t\_type

**contains**

**pure elemental real function** f(this)

type(t\_p), intent(in) :: this

**end function** f

a%fun => g; print \*, a%fun(); a=t\_type(f)

1. **Impure Intrinsic Subroutines**

**date & Time**

character(8) :: date !CCYYMMDD

character(10) :: time ! hhmmss.sss

character(5) :: zone ! +hhmm,-hhmm

integer::value(8) ! y,m,d,z(min),h,m,s,ms

call date\_and\_time([date][,time][,zone][,value])

integer ::count, c\_rate, c\_max

call system\_clock([count],[c\_rate],[c\_max])

print \*, (count – count0) / c\_rate, ‘(sec)’

call cpu\_time(time) ! (real :: time) !sum of CPUs

**random Numbers**

call random\_seed([size][,put][,get])

call random\_number(x) ! 0< x(:) <=1

**OS** compiler\_version(),compiler\_option() !iso\_f\*\_env

call get\_command\_argument(n[,val][,len][,stat])! [,msg])

call execute\_command\_line(cmd[,wait][,istat1][,istat2]

1. **I/O**

**file info**

use, intrinsic::iso\_fortran\_env

iostat\_end=-1, iostat\_eor=-2,error\_unit=0,

file\_storage\_size, numeric\_storage\_size ! in bits

is\_iostat\_end(iostat); is\_iostat\_eor(iostat) ! logical

input\_unit=5,outpu\_unit=6 ! read \*, x; print \*,x

inquire(file=’fort.9’, exist=lx, opened=lo) ! logical

inquire(iolength=i) olist !in recl ; i=storage\_size(a) !in bits

inquire(iolength=i)x,y;print \*,i,storgaze\_size(x) !→2 32

open(newunit=in, status=’unknown’) !auto unit no.

**sequntial file**

open([unit=]n,iostat=io,iomsg=text, err=99, end=999)

if (io==iostat\_eor)goto 99;if (io==iostat\_end)goto 999

open(n, status=’scratch’)!’unknown’’new’’old’’replace’

close(n[, status=’delete’])! *default* ’keep’

open(n, asynchronous=’yes’, action=’readwrite’);

read(n, asynchronous=’yes’) x;[*statements*];wait(n)

**move position** flush(n)

open(n,status=’old’, position=’append’)!’asis’ ’rewind’

backspace(n); rewind(n); endfile(n)![,iostat=io,err=9]

!skip🡪read(n,’(/)’) !goto eof🡪read(n,’(\*(/))’,iostat=io)

write(\*,’(a)’,advance=’no’) ‘input?’;read \*,n !no CR/LF

**direct access file / stream file**

open(n,access=’direct’,recl=10,form=’unformatted’)

open(n,access=’stream’,form=’formatted’)

new\_line(’ ’) ! ‘\n’

open(n,access=’stream’,form=’unformatted’)

inquire(n,pos=i); write(n, rec = i) x ! cuurent position

1. **Format**

G0, Gw.d, Gw.dEe ! General any intrinsic data

A, Aw !Character A auto adjusts width

Lw ! Logical

Iw, Iw.m ! Integer : ‘(sp, i4.3)’, 3 => +003

Bw.m, Ow.m, Zw.m ! Bin, Oct, Hex Integer

Fw.d ! Fixed : use d = 0 for read

Ew.d, Ew.dEe ! Exponential -0.12E+03

ESw.d, ESw.dEe ! Scientific E -1.23E+02

ENw.d, ENw.dEe ! Engineering E-123.45E+00

Tc ! Tab (absolute column)

TLc, TRc, [n]X ! relative move: TL left, TRn=nX right

[n](…), \*(..) ! grouping:[n]times or unlimited repeat

/ ! skip line CR/LF; ‘/’ aborts read, if read as num.

: ! abort if EOR; print “(\*(g0, :, ‘+’))”, 1,2,3 !🡪1+2+3

S SP SS ! Default,Show Plus,Suppress plus Sign

BZ BN ! read Blank as Zero, Blank as Null

character(len = 10) :: fmt = ’(2f5.0)’

read(\*, fmt) x, y; write(\*, “(a,2es9.1,’%’)”),’xy=’, x,y

print ‘(\*(g0, x))’, 1, 1.0, 1.0d0, ‘abc’, .true.

print ‘(b32.32)’, 0.123, x ! bit pattern

**List-directed** ! character(3)::txt(5); read \*, txt

input>x\*3; 3\*xxx; 9/x → ‘x\*3’,’xxx’,’xxx’,’xxx’,’9’

1. **Miscellaneous**

**associate construct**

associate(x=>a(i)%pos%r, y=>a(i)%pos%s) ! pointer

euclid\_norm = hypot(x, y)

end associate

**allocate/deallocate**

character(:), allocatable::t(:); class(\*), allocatable::q

allocate(character(n) :: t(m)); allocate(real(16)::q)

allocate(a(lbound(b,1)):ubound(b,1)),source=b)

deallocate(a, stat=stat, errmsg=text) ! stat=0 🡪 ok

**miscellaneous intrinsic functions**

abs, int, nint, mod ! towards zero, symmetric at zero

floor, ceiling, modulo ! translationally symmetric

sign(a, b) ! sign\_of\_b \* |a| ! floor(x) <= x <= ceiling(x)

atan(y[, x]), hypot(x, y) ! = abs(cmplx(x, y))

dot\_product(c, d), norm2(z) ! <c|d>, |z|2

min(a1, a2[,a3,..]),max(a1,a2[,a3,..]) ! unlimited args.

**carriage control** *(f03 deleted)*

|  |  |  |
| --- | --- | --- |
| 1H | ‘ ‘ | newline *e.g.*100 format(1H ,…) |
| 1H+ | ‘+’ | overprint |
| 1H0 | ‘0’ | skip a line |
| 1H1 | ‘1’ | next page |

1. **Coarrays**

**PGAS** (Partitioned Global Address Space)

Images 1..n, array(row, col…)[co1, co2..]

**Query functions**

integer :: k[\*], m(10)[4, \*], im(2)

me=**this\_image()**, ni=**num\_images()**

im = **this\_image**(m[,dim]), **image\_index**(m, im) ! me

**lcobound**(m[, dim][,kind]), **ucobound**(m[, dim][,kind])

**Memory move/allocation**

**put** k[2] = k, **get** k = k[2], **move** k[2]=k[3] !put is better

**Stop** ! stop terminates a single image

error stop [stop-code] ! terminates all images

!

**no order / independent** i in{1..n}

program no\_order

implicit none

integer::me, ni

me=this\_image()

ni=num\_images()

print \*, 'image ', me, '/', ni ! 2,4,1,3,..,ni

end program no\_order

**no order / mutual exclusion (statements)**

program noorder

implicit none

if (this\_image()==1) then

open(9,file='caf',status='replace')

close(9)

end if

**sync all** ! wait

**critical** ! mutual-exclusive statements

open(9,file='caf',position='append')

write(9,\*) this\_image()

close(9)

**end critical**

end program noorder

**total order 1,2,3…ni**

program order

implicit none

integer :: me, ni

me = this\_image()

ni = num\_images()

if (me > 1 ) **sync images (me - 1)**

print \*, 'image ', me, '/', ni ! 1, 2, 3,…,ni

if (me < ni) **sync images (me + 1)**

end program order

**partial order 1, others**

program oneothers

implicit none

integer :: me

me = this\_image()

if (me==1) then

print \*, 'first ', me ! segment 1

**sync images (\*)**

else

**sync images (1)**

print \*, 'others', me ! segment 2

end if

end program oneothers

**lock/mutual exclusion (statements; blocking)**

program blocking

use,intrinsic::iso\_fortran\_env,only:lock\_type

implicit none

**type(lock\_type)**::lock[\*]

if (this\_image()==1) then

open(9,file='caf',status='replace')

close(9)

end if

**lock(lock[1])** ! critical…end critical

open(9,file='caf',position='append')

write(9,\*) this\_image()

close(9)

**unlock(lock[1])**

sync all

end program blocking

**lock/mutual exclusion (statements;non-blocking)**

program nonblocking

use,intrinsic::iso\_fortran\_env,only:lock\_type

implicit none

**type(lock\_type)**::lock[\*]

logical::locked

**lock(lock[1], acquired\_lock=locked)**

if (locked) then

print \*, 'got lock', this\_image()

else

print \*, ' locked', this\_image()

end if

**unlock(lock[1])**

end program nonblocking

**implicitly sync all** ;real, allocatable:: x(:)[\*]

allocate(x(n)[\*]); deallocate(x)

auto dealloc at return from subprogram

**atomic/mutual exclusion (variable; non-blocking)**

program spinwait

use, intrinsic :: iso\_fortran\_env

implicit none

**logical(atomic\_logical\_kind)**::locked[\*]=.true.

logical::val

if (this\_image()==1) then

**sync memory**

**call atomic\_define(locked[2], .false.)**

print \*, 'unlock!'

else if (this\_image()==2) then

val=.true.

do while(val)

**call atomic\_ref(val, locked)**

print \*, 'loop 2'

end do

**sync memory**

else

print \*, ' pass through', this\_image()

end if

end program spinwait

1. **Coarrays in Fortran2018**

**reduction**

call co broadcast( a, source image[ , stat, errmsg] )

call co max( a[ , result image, stat, errmsg] )

call co min( a[ , result image, stat, errmsg] )

call co sum( a[ , result image, stat, errmsg] )

call co reduce(a,operation[,result image,stat,errmsg])

**atomic**call atomic add ( atom, value[ , stat] )

call atomic and ( atom, value[ , stat] )

call atomic or ( atom, value[ , stat] )

call atomic xor ( atom, value[ , stat] )

call atomic fetch add ( atom, value, old[ , stat] )

call atomic fetch and ( atom, value, old[ , stat] )

call atomic fetch or ( atom, value, old[ , stat] )

call atomic fetch xor ( atom, value, old[ , stat] )

call atomic cas ( atom, old, compare, new[ , stat] ) **references**

M.Metcalf, J.Reid, M.Cohen, *Modern Fortran Explained* (2011) Oxford Press.

J.Reid, ’The new features of Fortran 2018’ (2018).

fortran66のブログhttp://fortran66.hatenablog.com/

　H30.2.26 ver.0.1 <https://qiita.com/cure_honey>