Fortran2003/08 Quick Reference

1. Primitive 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 !f08
    int32, int64, real32, real64, real128 etc.
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 function default kind=ks
print *, conjg(c),real(c), imag(c), c%re, c%im ! f08
array constructor f03→[1,2,3] , f90→ (/1,2,3/)
a=[integer::];allocate(a(0))! [type::] null array with type
fib=[fib(2),fib(1)+fib(2)]; a=[1,[1,2]] ! flatten a→[1,1,2]
```

2. Character

2.1 basics ~ ^ * { } ` [ ] | # @ !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),len_trim(text),trim(text) ! 5, 3, 'abc'
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
```

2.2 deferred/assumed length

```
character (len=:), allocatable :: text ! f03
text = 'abc' ;text= text // 'def'  ! variable length
character (len=*), intent(in) :: text ! subprogram arg.
```

2.3 character ←→ number

```
integer, kind :: knd
integer, len  :: n
character(kind) :: a(n)
end type t_a
type(t_a(knd(0.0), 10)), target :: a
type(t_a(knd(0.0), : )), pointer :: ap
type(t_a(knd(0.0), * )),intent(in):: ai
ap => a; print *, ap%knd, ap%n
```

3. Derived Types

3.1 basics

```
type :: t_pos
  real :: x, y
  type(t_pos), pointer :: p => null()
! type(t_pos),allocatable :: p !f08
end type t_pos
type, extends(t_pos) :: t_p
  real :: vx = 0.0, vy =0.0
end type t_p
type(t_p) :: p = t_p(1.0,1.0,null())
print *, p%x, p%vx, p%y, p%vy
```

3.2 Parametrized Derived Types

```
type :: t_a(knd, n)
  integer, kind :: knd
  integer, len  :: n
  real(knd) :: a(n)
end type t_a
type(t_a(knd(0.0), 10)) :: a
```

4. Program Structure

```
module m_sub
  use, intrinsic :: iso_c_bindings, only : c_ptr
  implicit none
  private; public :: sub, fun, rx ! f08 default save
  real, protected::rx=0.0 ! read only global variable
contains
  subroutine sub(x, y)
    .....
  contains                  ! internal subprograms
    real function f(s) ; ... ;
      end function
  end subroutine sub
end module m_sub
program main
  use m_sub, l_sub1 => sub, fun
  implicit none
  [specification-stmts]
  [executable-stmts]
contains                    ! use with care
  [internal subprograms] ! primarily, substitute for
end program main            ! statement function
```

5. Subprograms f95,f90 deallocs allocatables at exit.

5.1 subroutine pointers are not dealloced

```
pure elemental subroutine s(a, x, y) ! f95 elemental
  real, intent(in), value :: a !f03 call by value attrib.
  real, intent(in) :: x ! default call by reference
  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
```

5.2 Function (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(res) ; integer::ires
  integer, intent(in) :: i
  if (i==1) 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 = [1.0, x] + [x, 1.0]  ! [1.0+x(1), x(1)+x(2),...]
end function f  ! x = f(x) ; Pascal's triangle
```

6. Control constructs

```
if (logical) statement      ! if (n==0) return
[name:] if (logical) then
  statements
else if (logical) then
  statements
else
  statements
end if [name]
[name:] select case (x) ! x ← int or char
  case (1:3)      ! (1), (2,3), (:4) or (5:)
    statements;
  case default
    statements;
end select [name]
[outer:] do   ! do while(..) has optimization issues
  [inner:] do i = n0, n1, nstep
    if (logical) cycle [inner]
    if (logical) exit   [outer]
  end do [inner]
end do [outer]
```

7. parallel operations

7.1 subscript triplet / vector subscript

```
a=[1,2,3,4];a=a(4:1:-1)      ! a→[4,3,2,1]
do i=1,4;a(i)=a(5-i);end do ! a→[4,3,3,4]
a=[2,3,1];a=a(a) ! a→[3,1,2] subsc. must be bijective
```

7.2 zero-sized arrays y = [real::]←→allocate(y(0))

```
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) !skipped if i=n
end do
```

7.3 parallel constructs

```
where (a>0); a=sqrt(a);elsewhere;a=0.0;end where
forall (i=1:n, j=1:n,a>0.0) b(i, j) = a(i)/(i + j) ! indx.dep.
do concurrent (i=1:n, j=1:n, a>0.0) ! f08
  block ! f08 ALGOL-like block construct
    real:: x, y ! thread safe local var
    x = a * i + b; y = a * j + b; b(i, j) = a(i, j) * f(x, y)
  end block
end do ! forall is array assignment, not parallel do
forall (i=1:n); e(i)=1; d(i)=i; end forall
!X forall(..){e(i)=1;d(i)=i} Oforall(..)e(i)=1; forall(..)d(i)=i
```

c.f S.Lionel, Intel Parallel Universe Magazine (**11**), 22.

8. Format (avoid format statement)

```
character(len = 10) :: fmt = '(2f5.0)'
read(*, fmt) x, y;   write(*, "(a,2es9.1,'%')"),'xy=', x,y
```

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

```
print '(*(g0, x))', 1, 1.0, 1.0d0, 'abc', .true.
print '(b32.32)', 0.123, x
```

Fortran2003/08 Quick Reference

14. Non-default derived type I/O *f03*

```
type :: t_dt
  integer :: i, j
contains
  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
```

15. I/O

15.1 sequntial file

```
use, intrinsic::iso_fortran_env
  iostat_end=-1, iostat_eor=-2,error_unit=0,
  input_unit=5,outpu_unit=6 ! read *, x; print *,x
open(newunit=in, status='unknown') !auto unit no.f08
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)
```

15.2 move position

```
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
```

15.3 direct access file / binary file

```
open(n,access='direct',recl=10,form='unformatted')
open(n,form='binary',recl=1) ! byte/word undefined
inquire(file='fort.9',exist=lx, opened=lo) ! logical
inquire(n, nextrec = i); write(n, rec = i) x
```

16. Miscellaneous

16.1 associate construct

```
[name:] associate(x=>a(i)%pos%r, y=>a(i)%pos%s)
  euclid_norm = hypot(x, y)
end associate [name]
```

16.2 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
```

16.2 carriage control *(f03 deleted)*

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

references

M.Metcalf,J.Reid,M.Cohen,*ModernFortranExplained*

M.Cohen, '*Implementing the Standards… including Fortran 2003*' www.fortran.bcs.org/2007/jubilee/f50.pdf

B.Long, '*Fortran 2003 and Beyond*',Cray User Group 2005

9.2 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
```

9.3 random numbers

```
call random_seed([size][,put][,get])
call random_number(x) ! 0< x(:) <=1
```

9.4 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 <=ceiling
atan2(y, x)!re(iθ)←hypot(x,y),atan2(y,x)!f08atan(y,[x])
hypot(x, y)! f08 avoids overflow: =abs(cmplx(x, y))
dot_product(c, d), norm2(z)  ! <c|d>, f08 |z|2
min(a1, a2[,a3,..]),max(a1,a2[,a3,..]) ! unlimited args.
```

10. Array a:array,m:logical array

10.1 reduction / mask ¡ func(a[,dim][,m])

```
all(), any(m), parity(m), count(m) ¡∧, ∨, Σ01, Σ1
iall(a,m), iany(a,m), iparity(a,m) ¡ bitwise reduction
sum(a,m), product(a,m) ¡Σ, Π
findloc(a,key[,dim][,m][,kind][,back]) ¡ f08 not found 0
minloc(a,m),maxloc(a,m), minval(a,m),maxval(a,m)
¡Tip: minloc(a)→array [i], minloc(a, dim=1)→scalar i
```

10.2 pack/unpack filter function

```
x = pack(a,mask=a > 0.0) ¡filter a > 0 to allocatable x
a = unpack( elem_func( pack(a, m) ), m)
```

10.3 array operation ¡ ord=[..], pad=s[1..n], shp=[..]

```
reshape(src,shape[,pad][,ord])¡1Dsrc(ord)+pad→shp
merge(a_true, a_false, mask):spread(src, dim, ncpy)
cshift(a,sft[,dim]) ¡ cyclic sft=k a(..,n,..)=a(..k,.) in dim
eoshift(a,sft[,bndry][,dim]) ¡ padded by bndry ¡+sft=←
transfer(src,mold[,size]), matmul(a, b), transpose(a)
```

10.4 array inquiry [,d,k]=[,dim][,kind] *f03 kind* 64bit

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

10.5 move_alloc ¡moves address not whole data

```
move_alloc(from, to) ¡ 'from' deallocated after copy
a=[1,2];tmp=[a,0*a];call move_alloc(tmp,a)¡ [1,2,0,0]
```

11. Interface block

11.1 generic operator

```
interface gamma ¡ generic name
 [module] procedure :: s_gamma, d_gamma
end interface
```

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

```
interface operator (.op.)    ¡ unary: .op. x,   binary: x .op. y
 [module] procedure :: func ¡ function unary f(x), binary f(x,y)
end interface                ¡ intent(in) :: x, y
```

11.3 assignment operator

```
interface assignment (=)     ¡ y = x → subroutine(y, x)
 [module] procedure :: sub ¡ intent([in] out):;y; intent(in)::x
end interface
```

11.4 abstract 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, f1   ¡ subprogram interface
procedure(t_noargs), pointer :: g0, g1 ¡ procedure pointer
procedure(real(8)), pointer::f; f=>cos; f(pi) → -1.0d0¡ intrinsic
procedure(sin), pointer::f; f=>cos ¡ intrinsic type, subprogram
```

12. Object Oriented Programming 1

12.1 inheritance

```
module m_oop ... ;
 type(t_p), extends(t_pos) ::t_p
 real :: px, py
```

12.2 methods

```
 contains
  procedure::sub_int; procedure::sub_real; procedure::op
  generic :: sub=>sub_i, sub_r; generic :: operator(.op.)=>op
  final :: sub_cleanup
 end type t_p
 contains                    ¡ implementations
  subroutine sub_i(t,this,_i)
  class(t_p),intent(in)::this; integer, intent(in) :: i
  end subroutine sub_i
  subroutine sub_r(this, x)
  class(t_p),intent(in)::this; real, intent(in) :: x
  end subroutine sub_r
  logical function op(this)
  class(t_p),intent(in)::this; op = this%x < 0.0
  end function op
 end function func_p
 subroutine sub_cleanup(this)
 type(t_p) :: this
 print *,'cleanup',this%x
 end subroutine sub_cleanup
end module m_oop ... ;
```

13. Object Oriented Programming 2

use m_oop

```
type(t_pos), allocatable :: p1
class(t_pos), allocatable :: p2(:)
class(*), allocatable :: q
p1 = t_q(1.,1.,0.,0.) ¡ scalar allocation & init.
allocate(p2(5), mold=p1) ¡ class array allocation
allocate(q, source=p1) ¡ unlimited class allocation & init.
same_type_as(a, b); extends_type_of(a, mold) ¡ type inquiry
select type(q)   ¡ q => p%element
 type is (t_p)   ¡ derived type, intrinsic type
 print *,q%t_pos, q%x + q%px ¡ touch class(*) variable
 class is (t_pos) ...; ¡ type is (real) ¡ derived&intrinsic type
 class default ...;
end select
call q%sub(1.0);call q%sub(1.0,.true.,p1);deallocate(q, p1)
type, abstract :: t_abst   ¡ abstract type forces complete
 procedure(t_f), deferred :: func
end type t_abst
type, extends(t_abst) :: t_type
 contains
  procedure :: func => my_func
end type t_type
```