Appunti di Fortran

Un piccolo ripasso di un vecchio nuovo linguaggio

Edmondo Giovannozzi - Ricercatore ENEA

Prima parte

Sommario

- Cenni di Storia del FORTRAN.
- Fixed Free Form.
- Costanti e variabili
- Unità di programmazione.
  - moduli e procedure
- Strutture di controllo e non solo.
- Array.
- Memoria dinamica:
  - allocatable e pointer
- Qualcosa sull'input Output

L'argomento è vastissimo! Mi concentro sulle cose più utili e comunque disponibili (adesso o tra breve) sui compilatori.

Storia del FORTRAN

- Fortran (1957)
- Fortran IV (1962)
  - If logico
- Fortran 66 (1966)
  - primo standard industriale
- Fortran 77 (1978)
  - “IF .. THEN, ENDIF” statement, ed altri miglioramenti
  - “MILK-STD-1753” (1978), “DO WHILE”, etc.
- Fortran 90 (1991-1992)
  - Moduli, operazioni vettoriali, interfacce, memoria dinamica, etc.
- Fortran 95 (1997)
  - FORALL, procedure PURE ed ELEMENTAL, etc.
  - ISO technical report “TR-15581”: array ALLOCATABLE all’interno di strutture.
- Fortran 2003 (2004)
  - Tipi estendibili, **programmazione ad oggetti**, etc
- Fortran 2008 (2010)
  - coarray, DO CONCURRENT, opzione NEWUNIT in OPEN (Finalmente!!!), etc.
  - Technical Specification “TS 29113”, Ulteriore interopibilità con il C (2012)
  - Technical Specification “TS 18508”. Additional Parallel Feature in Fortran (parallel Team) (2014-2017)

Formato del Sorgente

- Tipicamente i file scritti usando la Fixed Form terminano in ‘.f’
- Quelli scritti in Free Form terminano in ‘.f90’
- Opportune opzioni del compilatore possono cambiare queste convenzioni.
- i file che devono essere preprocessati terminano tipicamente in ‘.F’ ed in ‘.F90’

Set dei caratteri

- Set dei Caratteri:
  - Eccetto che nei commenti e nelle stringhe non c'è distinzione tra maiuscole e minuscole
  - abcdefghijklmnopqrstuvz0123456789_
  - Spazio, =+-*/( )[],.:;!"'%&<> (hanno un significato nel sorgente)
  - \{}~?^|$#@ (nelle stringhe)
  - Carattere tabulazione non standard (fare attenzione)

Fixed Form

- Spazi non significativi!
- Colonne 7-72: Statement. Multipli statement sulla stessa linea possono essere separati da ‘;’
- Colonne 1-5: Label numeriche.
- Colonna 1: ‘C’ o ‘*’ indica una linea di commento
- Il carattere ‘!’ in qualsiasi posto tranne che nella colonna 6 inizia un commento.
- Colonna 6: Qualsiasi carattere tranne spazio e 0 indica una continuazione del precedente statement (continuation line). Al massimo 19 continuation lines.
- Lo statement ‘END’ non può essere continuato su un’altra linea.
- Colonne 72-Infinito: Commento

Tipicamente i compilatori permettono di avere linee più lunghe di 72 caratteri impostando alcune opzioni.

1	2	3	4	5	6	7	8	9	10	11	12	…	70	71	72	73	74	75	76	77	78	79	80
1	**0**					**a**	**=**	**(**	**1**		**+**		**)**	**+**									
					&		**(**	**5**	**+**	**7**	**)**					**c**	**o**	**m**	**m**	**e**	**n**	**t**	**o**

Free Form

- Gli spazi sono significativi!
  - Non possono apparire all'interno dei nomi delle variabili o delle varie keyword, eccetto dove l'uso permette entrambi i modi
    - tipicamente GOTO o GO TO, ENDIF e END IF, etc.
  - Devono essere usati per separare nomi e keyword.
- La linea è lunga fino a 132 caratteri
  - ma i compilatori permettono linee più lunghe tramite opzioni
- Il carattere '!' indica l'inizio di un commento (fino a fine linea)
- Più statement su di una linea possono essere separati da ';'
- Il carattere '&' alla fine di una linea indica che la linea successiva è la continuazione di quella attuale. La linea successiva può iniziare con una '&'.

Esempio di continuazione

Continuazione semplice:

```
force = mass1 * mass2 / &
radius ** 2
```

Due ampersand:

```
force = mass1 * mass2 / &
   & radius ** 2
```

Le ampersand possono spezzare nomi e stringhe:

```
force = mass1 * mass2 / ra&
   &dius **2
mio_commento = 'Ma guarda cosa tocca scr&
    &ivere'
```

Il baco più famoso

Trovare i due bachi (fixed form):

```
!234567
      program due bachi
      x = 10
      do 10 i = 1. 10
          x = x + 10
 10   continue
      y = 20
      if ( y .gt. 10) then y = 10
      print *, x, y
      end program
```

Cosa scrive il programma a schermo?

Il programma non sarebbe valido se si usasse la Free Form.

Esercizio: scrivete questo esempio compilatelo ed eseguitelo.

Il baco più famoso

Trovare i due bachi (fixed form):

```
!234567
      program due bachi
      x = 10
      do10i = 1.10
          x = x + 10
 10   continue
      y = 20
      if ( y .gt. 10) theny = 10
      print *, x, y
      end program
```

Cosa scrive il programma a schermo?

20.000 20.000

`do10i`

e

`theny`

sono due legittimissimi nomi di variabili

L'uso della Free Form o di `implicit none` avrebbe permesso di scoprire questi bachi durante la compilazione

Indentazione

- Esiste una convenzione di indentare i blocchi nel sorgente (tipo cicli, if, etc.):

```
a = 32
do i= 1, 10
    a = a + i
enddo
if (a>10) then
    if (a>20) then
        a = a-20
    endif
    a = a+3
endif
print *, a
```

- Indentare usando spazi invece che i tab (qualsiasi editor può essere impostato in questo modo)
- L'indentazione tipica sono 4 spazi. Nella fixed form anche 3 spazi vanno bene (perché 3+3=6 e posso iniziare a scrivere nel settimo carattere).

Un esempio

Un esempio per impostare una directory ed il compilatore. Create una directory e metteteci i seguenti file:

Makefile

```
FC = ifort
FFLAGS =

%.exe : %.f90
    $(FC) $(FFLAGS) -o $@ $<

%.exe : %.f
    $(FC) $(FFLAGS) -o $@ $<
```

ciaociao.f90

```
program ciaociao
implicit none

character(20) :: saluto

saluto = 'Ciao! Ciao!'
print *, saltuto

end program
```

Da unix eseguite:

fusc10> **make ciaociao.exe**

fusc10> **ciaociao.exe**

Li trovate su: ~giovan/public/CorsoFortran

Esercizio

- Esercizio: come deve essere scritto un programma per andare bene sia nella free form che nella fixed form? Specialmente per quel che riguarda le linee continuate.
- Trasformate un vostro programma da fixed form a free form.

costanti

reali	doppia precisione	complesse	intere	logiche	Carattere
1.0 1.0E2 -1.	1.0D0 1.0D2 -1.D0	(1.0, 3.4) (-2.e4, 0.) (1, -1)	23 -12345 b'1001' o'732' z'2AF'	.true. .false.	'Ciao come stai' "Bene e tu? " "Come ti dissi ieri: 'Bene! ' ."

BOZ constant. Solo in alcuni casi. Interpretati come bit pattern

C'è la possibilità di specificare esplicitamente il kind di una costante con un _ seguito da un numero o da un parametro costante (ed è questo l'uso consigliato)

```
integer, parameter :: wp = kind(0.0D0)

real(wp) :: adouble = 2.0_wp

real :: asingle = 1.0_4
real(8) :: b = 3.0_8

b = b + a * 3.4_wp
```

Su Linux tipicamente wp = 8

Sconsigliato

Dichiarazioni di variabili

- *tipo [* **(***kind***)** *]][[* **,***attributi,…]* **::** *]* nome [= inizializzazione][, …]
  - *tipo*: **`real`**, **`integer`**, **`complex`**
  - *kind*: tipicamente 4 per singola precisione, od 8 per doppia. Per le costanti carattere in realtà è la lunghezza (ma esiste anche il kind).
  - *attributi*: varii li vedremo in seguito
  - « **::** »: sono obbligatori se sono presenti gli attributi o l'inizializzazione
- Esempio:

```
real pluto
double precision clarabella
integer :: topolino
complex, optional :: minnie
real(kind=8) :: paperone = 1.2345D0, paperoga = 0.78D0
```

E dove sono i **`real*8`** etc?

E' una estensione non standard dove il numero sta per il numero di byte occupati dalla variabile. E' standard solo per le variabili carattere.

implicit

- L'istruzione implicit dichiara il tipo implicito delle variabile basandosi sull'iniziale. La variabile viene dichiarata se viene usata.
- Di default:
  - (A-H,O-Z): Reali, (I-N): Intere
- Esempio:

```
implicit complex (a-z)
```

- **`implicit none`**
  - Tutte le variabili devono essere dichiarate esplicitamente.
  - E' praticamente obbligatorio!

Il rischio di scrivere male il nome di una variabile e quindi inserire un baco estremamente difficile da scoprire è troppo alto. In tutti i nuovi programmi si deve inserire l'istruzione:

```
implicit none
```

Lo stesso programma duebachi.f di prima sarebbe risultato errato usando questa istruzione (e quindi si sarebbe trovati i bachi durante la compilazione)

Variabili carattere

`character*20`	stringa di 20 caratteri di default kind
`character(20)`	stringa di 20 caratteri di default kind
`character(len=20)`	stringa di 20 caratteri di default kind
`integer, parameter :: ck = kind(' ')` `character(kind=ck, len=20)`	stringa di 20 caratteri . Tipicamente : ck = 1
`integer, parameter :: ck = kind(' ')` `character(len=20, kind=ck)`	stringa di 20 caratteri . Tipicamente : ck = 1

Come dummy argument	
`character*(*)`	Stringa senza lunghezza specificata assume quella dell'argomento attuale.
`character(*)`	
`character(len=*)`	

Sottostringa

```
character(20) :: a
a = 'Ciao come stai'
a(2:5) == 'iao '
```

Lunghezza non specificata

```
character(*),parameter :: a = 'Ciao'
character(:), allocatable :: b
b = 'Ciao come stai'
```

Il kind

E' nato l'uso nel Fortran di specificare il tipo delle variabili tramite l'occupazione in byte: **`real*8`** (variabile reale che occupa 8 byte)
Questo non è nello standard ma è molto usato.

Il **kind** tipicamente si rifà a questi numeri, ma attenzione ad il **kind** dei complessi

Estensione	F90
`real*4`	**`real(4)`**
`real*8`	**`real(8)`**
`complex*8`	**`complex(4)`**
`complex*16`	**`complex(8)`**

Il **kind** di un complesso è il **kind** del real sottostante, ma l'occupazione in byte è doppia

Nel fortran Intel e nel più recente gfortran viene supportato il modulo ISO_FORTRAN_ENV che fornisce le costanti **`int32`**, **`int64`**, **`real32`**, etc.

Funzioni (kind)

`kind(x)`	ritorna il kind dell'oggetto x
`kind(1.0)` → 4 `kind(2.0D0)` → 8	

`selected_int_kind(r)`	ritorna il kind di un intero $-10^r < n < 10^r$
`selected_int_kind(5)` → 4 `selected_int_kind(10)` → 8	

`selected_real_kind(p,r)`	ritorna il kind di un reale data la precisione ed il range

p è circa il numero di cifre decimali
r è circa il massimo esponente decimale

	real(4)	real(8)
p	6	15
r	37	307

`range(x)`	ritorna il range dell'oggetto x
`range(1.0)` → 37 `range(2.0D0)` → 307	

`precision(x)`	ritorna la precisione dell'oggetto x
`precision(1.0)` → 6 `precision(2.0D0)` → 15	

Altre funzioni

- **tiny(x)** : il numero più piccolo del tipo x

  `tiny(1.0)` → 1.18E-38

  `tiny(1.0D0)` → 2.23E-308

- **huge(x)** : il numero più grande del tipo x

  `huge(1.0)` → 3.40E38

  `huge(1.0D0)` → 1.79E308

  `huge(1)` → 2147483647

- **epsilon(x)** : numero piccolo rispetto ad 1 del tipo x

  1.0 + **epsilon**(1.0) /= 1.0, 1.0 + **epsilon**(1.0)/2.0 == 1.0

  `epsilon(1.0)` → 1.19E-7

  `epsilon(1.0D0)` → 2.22E-16

Parameter

- Per dichiarare delle variabili costanti (calcolate durante la compilazione).
- Due modi:

  Stile F90

  tipo, **parameter** :: *nome* = *inizializzazione*

  Esempio:

  ```
  integer, parameter :: pluto = 57
  ```

  Stile F77:

  tipo :: *nome*

  parameter (nome = inizializzazione, …)

  Esempio:

  ```
  integer :: pluto
  parameter (pluto = 57)
  ```

- La lunghezza delle variabili carattere può non essere specificata, la si prende dalla stringa di inizializzazione:

  ```
  character(*), parameter :: paperino = 'Ciao casa!'
  ```

tipi definiti dall'utente

- contengono all'interno tipi semplici (interi, reali, carattere, od altri tipi definiti dall'utente)

```
type [[attributi, ..] :: ] nome
   …
end type
```

Esempio:

```
type clarabella
   integer :: pippo
   real :: topolino
end type
```

- Una variabile di un tipo definito dall'utente si dichiara tipicamente:

```
type (clarabella) :: clara
```

- Le sottoparti del tipo si accedono tramite «%»:

```
clara%pippo = 57
print *, clara%topolino
```

- Alla definizione del tipo le parti possono essere inizializzate:

```
type paperino
   integer :: qui = 1
   real :: quo = 4.5
   complex :: qua = (0,-1)
end type
```

structure constructor

```
type Tpoint
   real :: x = 0.0
   real :: y = 0.0
   integer :: order
end type

type (Tpoint) :: point_a, point_b

point_a = Tpoint(1.0, 2.0, 3)
point_b = Tpoint(order=7)
```

Si può assegnare un valore ad una variabile di tipo **Tpoint** usando il nome del tipo come se fosse una sorta di funzione che ritorna il tipo suddetto. Tutti i campi che non hanno una inizializzazione di default devono essere presenti nel costruttore (in questo caso **order**).

Espressioni

- Espressioni generali
- Espressioni di inizializzazione (initialization expression)
  - Valutate al momento della compilazione.
  - Tipicamente per inizializzare le variabili.
  - Anche vettoriali.
- Espressioni di specificazione (specification expression)
  - Valutate al momento dell'ingresso in un sottoprogramma.
  - Tipicamente usate per specificare i limiti degli array all'interno di un sottoprogramma..
  - Possono apparire anche variabili presenti in common od in moduli accessibili.
  - Intere scalari.

Semplici espressioni

Gli operatori di comparazione possono essere scritti in forma nuova più simile ad altri linguaggi ed alla matematica.

	Fortran 77	Fortran 90
uguaglianza	`.eq.`	`==`
disuguaglianza	`.ne.`	`/=`
maggiore	`.gt.`	`>`
maggiore uguale	`.ge.`	`>=`
minore	`.lt.`	`<`
minore uguale	`.le.`	`<=`
eguaglianza logica	`.eqv.`	`.eqv.`
diseguaglianza logica	`.neqv.`	`.neqv.`

⇐ ≠

operatori logici	
.or.	
.and.	
.not.	
vero	.true.
falso	.false.

concatenazione stringhe
//

Ed anche:

elevazione a potenza	**
somma, sottrazione, moltiplicazione, divisione	+,-,*,/

L'ordine di valutazione lo decide il compilatore!
In particolare:
`if (x>0 .and. log(x)>2.0) ….`
è **errata** perché il compilatore potrebbe calcolare **log(x)** prima di x>0.

semplici strutture

Cicli:

```
do i = 1, n
   ...
end do
```

Qualche semplice struttura che useremo negli esempi

`i` deve essere una variabile intera!

Espressioni condizionali:

```
if (condizione1) then
   ...
else if (condizione2) then
   ...
else
   ...
end if
```

```
if (condizione) …
```

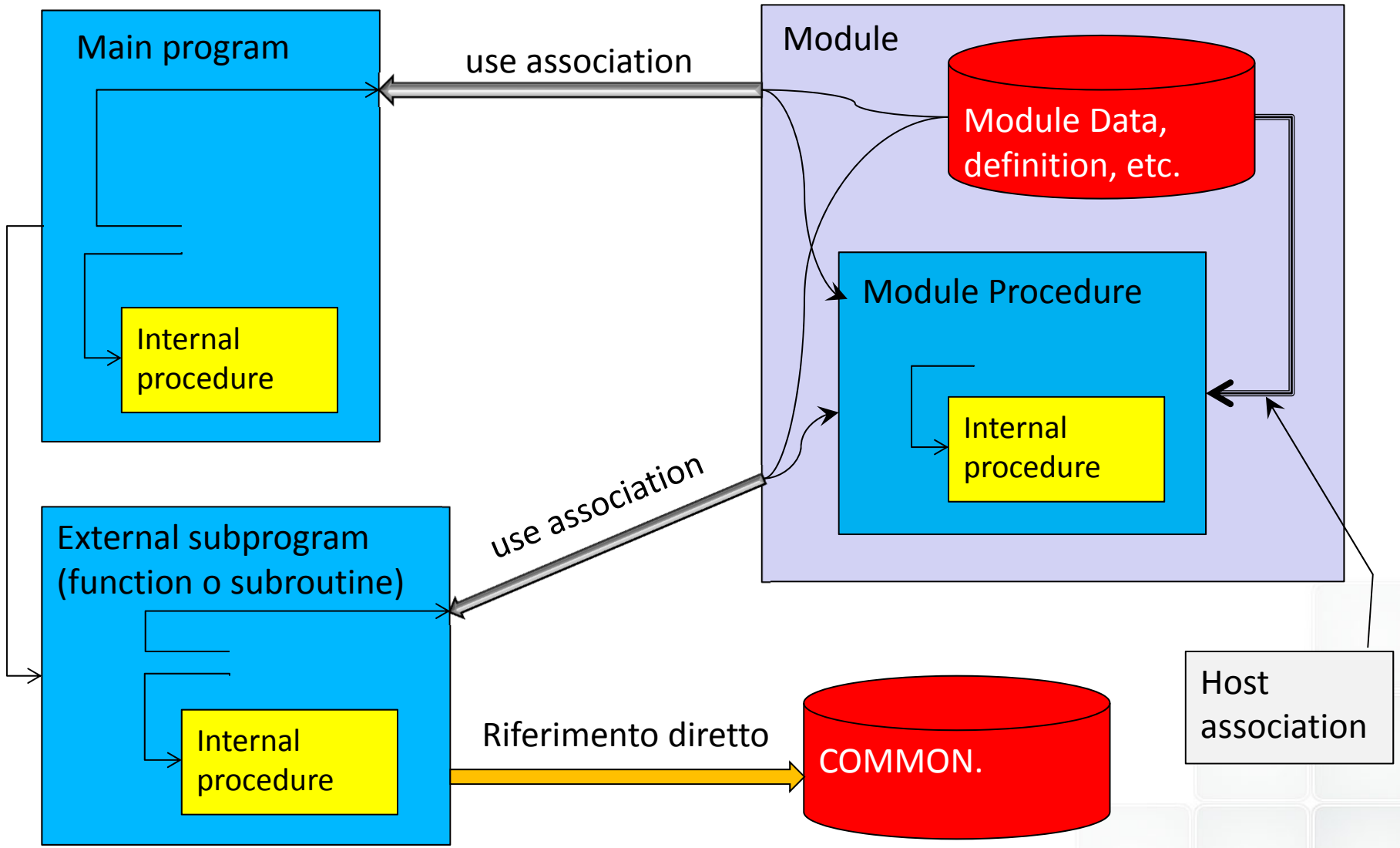
Le parentesi intorno alla condizione sono obbligatorie

```
if (a > 5) a = 5
```

Unità di programmazione

- Main program (PROGRAM)
- External subprograms
  - subroutine e function, come nel F77
- Moduli
  - possono contenere subroutine e function
- Internal procedure
  - interni ad una funzione, subroutine o del main program (sostituti delle funzioni a singola linea)
- Module procedure
  - Subroutine o function che appartengono ad un modulo
- block data (obsoleto)
  - per specificare i dati di un modulo
- common (non sono unità di programmazione)
  - ben noti ma è meglio usare moduli se possibile

Unità di programmazione



Come si specificano

main program
program *nome*
…
end program

Di una subroutine:
subroutine *nome*(*parametr*i,…)
…
end subroutine

Di una funzione
function *nome*(*parametri*) **result(***risultato***)**
…
end function

Di un modulo
module *nome*
…
end module

E' necessario specificare **subroutine** o **function** negli **end** statement all'interno di un modulo. Per gli altri è facoltativo.

Si può specificare negli end statement il nume della unità di programmazione che sta finendo:

```
subroutine pippo
…
end subroutine pippo
```

se nella definizione di una **function result (***risultato***)** non è specificato, la variabile di uscita ha lo stesso nome della funzione.

main program

```
program plutone
use pippo
implicit none

   real :: a = 2
   real :: b = 5

   call add_a(b)
   print *,a, b

contains

   subroutine add_a(b)
     real :: b
     b = b + a
   end subroutine

end program plutone
```

Nome del programma , subroutine o function.
Opzionale per il programma

moduli usati (solo come esempio, il modulo pippo non è realmente usato nel programma)

statement implicit

statement dichiarativi: definizioni di variabili, etc.

Statement esecutivi

Internal subprogram.

la variabile **a** è visibile per host-association.

Gli internal subprogram rimpiazzano le famose **statement-function**

moduli

```
module plutone
use pippo
implicit none

   real :: a = 2
   real :: b = 5

contains

   subroutine add_a(b)
      use minnie
      real :: b
      b = b + a
   end subroutine
end module plutone
```

Nome del modulo

moduli usati nel modulo.

statement implicit

statement dichiarativi: definizioni di variabili, etc.

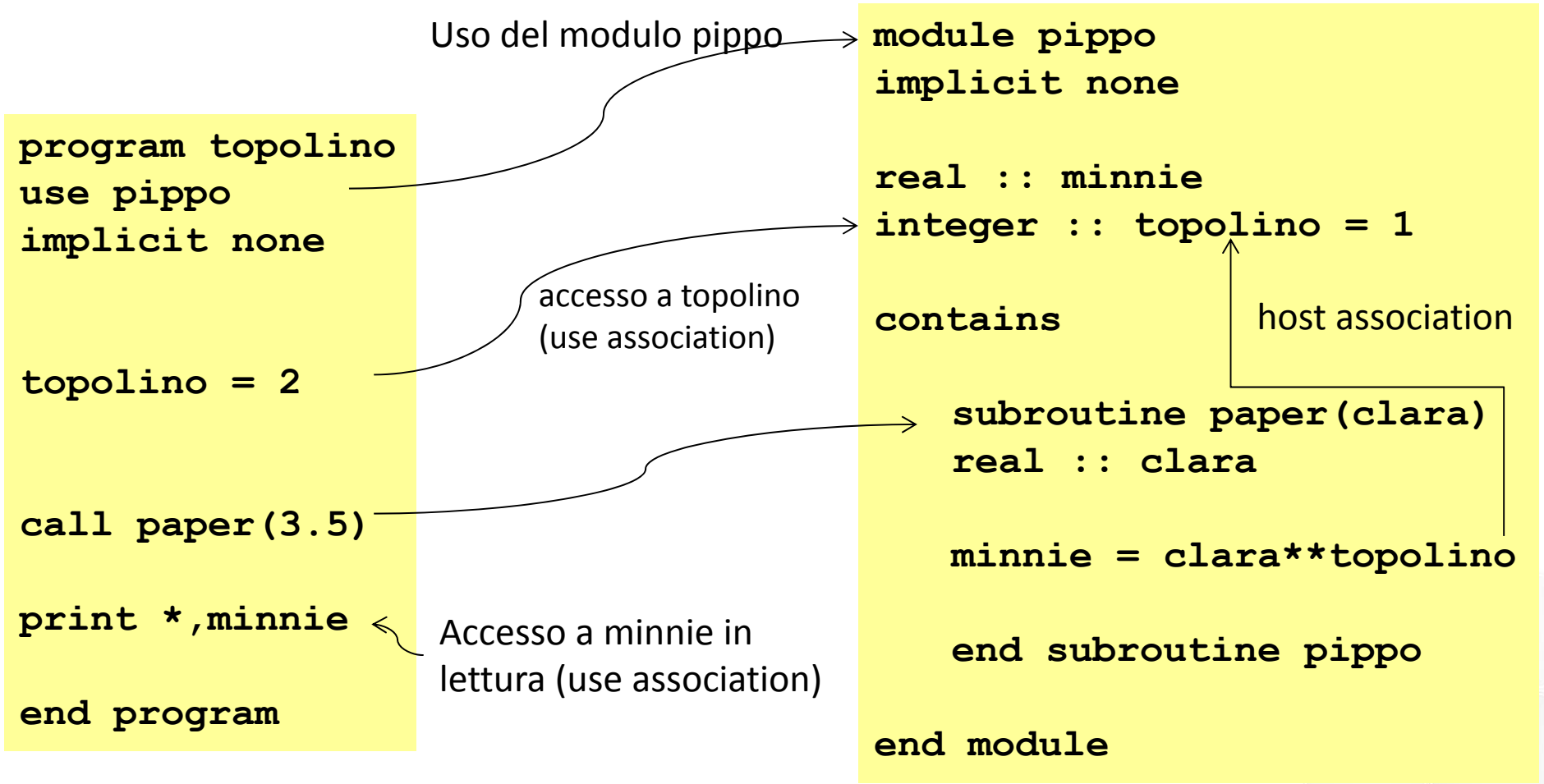
Non ci sono statement esecutivi

Module procedure.

la variabile **a** è visibile per host-association.

Le procedure di un modulo possono a loro volte contenere degli internal subprograms. ed usare ulteriori moduli

usare i moduli



Possibili usi dei moduli

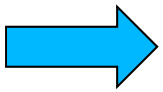
- Contenitori di interfacce per librerie
- Dati globali come sostituzioni dei common
  - tipicamente tramite use association
- Contenitori di dati e subroutine
  - Le subroutine agiscono sui dati del modulo
- Definizioni di tipi definiti dall'utente con le associate subroutine
  - I dati non risiedono nel modulo

sostituizione dei common

Primo step per passare da common a moduli

```
subroutine pippo
common /topo/ a, b, c
…
end subroutine
```

```
subroutine pluto
common /topo/ a, b, c
…
end subroutine
```



```
module topo
    real :: a, b, c
end module topo
```

```
subroutine pippo
use topo
…
end subroutine
```

```
subroutine pluto
use topo
…
end subroutine
```

Attenzione ad OpenMP.
E’ un primo step e non è il modo in cui conviene scrivere un programma nuovo.
Le variabili del modulo possono anche essere vettori dinamici (allocatable) con la loro dimensione calcolata a run-time.

Dati e procedure

```
program gambadilegno
use pluto

call set_topolino(4.5)

minnie = 7

call print_all

end program
```

```
module pluto
  implicit none

  real :: topolino
  integer :: minnie

contains

  subroutine set_topolino(topo)
    real :: topo
    topolino = topo
  end subroutine

  subroutine print_all
    print *,'TOPOLINO:', topolino
    print *,'MINNIE', minnie
  end subroutine

end module
```

Il modulo fornisce variabili sia set di funzioni per operare sulle variabili.

L'accesso diretto alle variabili può essere permesso.

Tipi e procedure

```
program gambadilegno
use pluto

type(trudy) :: tr

call set_topolino(tr, 4.5)

tr%minnie = 7

call print_all(tr)

end program
```

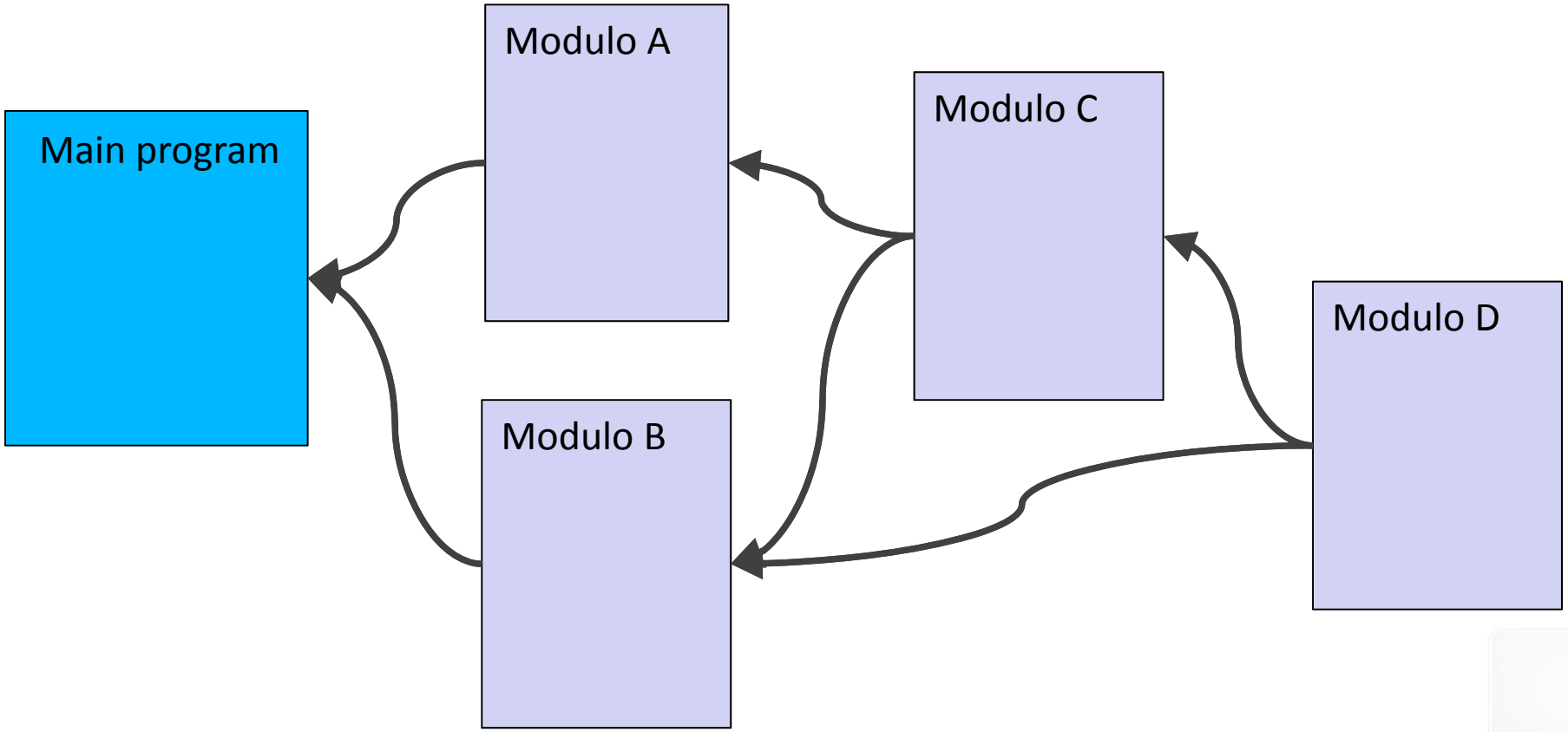
Il modulo fornisce dei tipi definiti dall'utente ed un set di funzioni per operare sui tipi.

L'accesso diretto alle sottoparti delle variabile può essere permesso.

Le funzioni sono naturalmente concorrenti e thread safe.

```
module pluto
  implicit none

  type trudy
    real :: topolino
    integer :: minnie
  end type
contains
  subroutine set_topolino(tr, topo)
    type(trudy) :: tr
    real :: topo
    tr%topolino = topo
  end subroutine

  subroutine print_all(tr)
    type(trudy) :: tr
    print *,'TOPOLINO:', tr%topolino
    print *,'MINNIE', tr%minnie
  end subroutine

end module
```

Moduli: use association



Grafo aciclico nella use association.
Il problema della compilazione a cascata, se per esempio cambia il modulo D è stato risolto con l'uso dei sottomoduli, che però non sono ancora disponibili nei compilatori.

Visibilità private e public

```
module pippo
implicit none

private

real minnie
integer topolino

public topolino, pippo
contains

    subroutine pippo
        …
    end subroutine pippo
    …
end module
```

Tutte le funzioni del modulo avranno implicit none

di default tutte le variabili e le funzioni sono private ovvero non usabili

Solo topolino e pippo sono pubbliche, ovvero chi usa il modulo può accedere a topolino e chiamare pippo. Ma non a minnie.

F2008 variabili protected (si possono leggere direttamente, ma scrivere solo attraverso le procedure del modulo)

use statement

- Uso

  use *[[nature] ::]* name [**,** *renamelist*]

  use *[[nature] ::]* name**, only:** *onlylist*

- *nature*: INTRINSIC, NON_INTRINSIC. Alcuni moduli, sono intrinsic. Permettono alla procedura che li usa di avere accesso ad alcune funzionalità aggiuntive.
- *renamelist*: *nome-locale* **=>** *nome-nel-modulo*
- *onlylist*: nomi delle variabili del modulo rese visibili, con possibile rinominazione.

use statement esempi

```
module pluto
   real :: pippo
   real :: minnie
end module
```

```
program topolina
use pluto, clara => minnie
integer :: minnie
…
end program
```

minnie è stata rinominata **clara**, quindi è possibile definire una variabile **minnie** locale senza che ci sia una sovrapposizione di nomi (vietata quando si usa un modulo)

```
program topolino
use pluto, only :: pippo
integer :: minnie
…
end program
```

solo la variabile **pippo** viene importata dal modulo. Questo come sopra permette di definire una variabile locale **minnie**

subroutine

- definizione:

  [prefisso] **`subroutine`** *nome [(argomenti, ..)]*

  - *prefisso:*
    - **`elemental`** : agisce sui singoli elementi di un vettore.
    - **`pure`** : non ha side-effect
    - **`recursive`** : può richiamare se stessa direttamente od indirettamente. Non può essere elemental.
    - **`impure elemental`** : agisce sui singoli elementi di un vettore in sequenza (FORTRAN 2008)
- Chiamata:
  `call subroutine` *nome [(argomenti, ..)]*
- Si può uscire da una subroutine in anticipo tramite l'istruzione **RETURN**. Questo è valido anche per le funzioni.

function

- definizione:

  [prefisso] [tipo] **`function`** *nome* **`(`***[argomenti, ..]***`)`** *[***`result(`***risultato***`)`***]*

  - *prefisso:*
    - **`elemental`** : agisce sui singoli elementi di un vettore.
    - **`pure`** : non ha side-effect.
    - **`recursive`** : può richiamare se stessa direttamente od indirettamente. Non può essere elemental.
    - **`impure elemental`** : agisce sui singoli elementi di un vettore in sequenza (FORTRAN 2008)
  - *tipo:* **`real`**, **`integer`**, **`character`**, etc. Può esssere specificato in questo punto od anche successivamente.
  - *risultato:* se il risultato non è specificato il risultato avrà lo stesso nome della funzione. E' obbligatorio nel caso la funzione sia direttamente ricorsiva.
- Chiamata:

  … **`=`** … *nome(argomenti, ..)* …
- Esempio:

```
real function topolino(a) result(minnnie)
…
minnie = ….
```

```
function topolino(a)
real :: topolino
…
topolino = ….
```

Qualche esercizio

- Scrivere un modulo con una variabile e due procedure una per impostare il valore della variabile ed una per leggerlo.
- Scrivere un modulo con delle variabili private e notare che il compilatore si rifiuta accedervi direttamente.
- etc.

procedure ricorsive

- Se non viene specificato l'attributo «**recursive**» le procedure (subroutine e function) non possono richiamare se stesse direttamente od indirettamente.
- Se non è strettamente necessario non scrivere funzioni ricorsive. Tipicamente dipende dalla struttura dei dati.
- Una procedura ricorsiva ha all'interno una condizione di terminazione tale che a seconda dei parametri di input non chiamerà se stessa ma uscirà direttamente (in caso contrario si richiamerebbe all'infinito fino a riempire lo stack)

```
recursive function somma(n) result(risultato)
implicit none
integer :: risultato, n

if (n == 0) then
   risultato = 0
else
   risultato = n + somma(n-1)
endif

end function
```

- Esercizio: scrivete una funzione ricorsiva che calcola la sequenza di Fibonacci.

interfaccia implicita (F77)

- L'interfaccia standard del Fortran 77 viene sempre assunta dal programma chiamante se non ha a disposizione una interfaccia esplicita (tramite una dichiarazione o tramite un modulo).
- Nell'interfaccia implicita viene tipicamente passato alla routine l'indirizzo delle variabili associate agli argomenti.
- Inoltre il numero delle variabili è fissato. Non ci sono variabili optional.
- Non si richiede che la procedura sia pura (ci sono costrutti all'interno dei quali possono essere presenti solo procedure pure)
- Non si passano argomenti per «keyword»
- Gli argomenti array sono: **explicit-shape** o **assumed-size** (ci torneremo in seguito, in pratica i modi di passare un array in F77)
- Gli argomenti non possono avere gli attributi **`optional`**, o **`value`**
- Torneremo in seguito sulle interfacce esplicite.

Interfaccia Esplicita

- La procedura chiamante ha esplicitamente accesso alla interfaccia della procedura chiamata.
- Tramite l'istruzione **interface**.
  - l'istruzione **interface** può trovarsi anche in un modulo usato dalla procedura chiamante.
- Tramite use association
  - la procedura chiamata si trova in un modulo usato dal chiamante.
- Tramite host association
  - la procedura chiamata si trova nella stessa unità di programmazione della procedura chiamante.
  - Una procedura in un modulo ha esplicitamente accesso alla interfaccia delle altre procedure in un modulo.
  - Un internal subprogram ha esplicitamente accesso alle interfacce delle altre internal subprogram (come anche la procedura che le contiene).

Corrispondenza tra argomenti dummy e quelli attuali

- per posizione come nel Fortran 77.

```
call        pluto(a,    b,   c)
subbroutine pluto(pri, sec, ter)
```

- Per keyword (richiede una interfaccia esplicita)

```
call pluto(ter=c, pri=a, sec=b)
```

in questo caso l'ordine degli argomenti non ha importanza.

- Mista, devono apparire prima quelli posizionali.

```
call pluto(a, ter=c, sec=b)
```

- L'uso delle keyword è particolarmente utile con gli argomenti opzionali.

attributo save

- Le variabili specificate all'interno delle procedure non sopravvivono alla fine della procedura a meno che:
  - sono state dichiarate con l'attributo «**save**»
  - hanno un «save» implicito perché inizializzate all'atto della dichiarazione
- Esempio:

```
subroutine pippo
real :: a
real, save :: b
real :: c = 3.4
```

variabile temporanea, alla fine della subroutine il suo contenuto non è più valido. Alla chiamata successiva risulta non inizializzata.

save esplicito

save implicito

- Esercizio:

Scrivete un subroutine che scrive a schermo il numero di volte che è stata chiamata

argomenti di una procedura: attributi

- Gli attributi tipici degli argomenti di una procedura sono: **`intent()`**, **`optional`**, **`value`**.
- Attributo **`value`**: l'argomento viene passato per «valore», se la procedura lo modifica questo non ha effetto sulla variabile nel programma chiamante:

```
program nonincr
implicit none
  real :: a = 2.0

  call nonincrementa(a)
  print *,'A', a

contains

  subroutine nonincrementa(b)
    real, value :: b
    b = b + 10.0
    print *,'B', b
  end subroutine

end program
```

- ha bisogno di una interfaccia esplicita (ho risolto mettendola come internal subprogram, ma è bene inserirla all'interno di un modulo)

argomenti opzionali

- Gli argomenti opzionali hanno l'attributo optional.

```
subroutine gambadilegno(trudy)
real, optional :: trudy
```

- possono essere usati solo se presenti
  - la funzione **present**(nome) ritorna **.true.** se l'argomento è presente
  - se non sono presenti possono solo essere passati ad un'altra routine che li dichiara anch'essa optional
- Esempio:

```
subroutine paperone(paperoga)
real, optional :: paperoga
real :: paperino
if (present(paperoga)) then
    paperino = paperoga
endif

call gambadilegno(paperoga)
```

posso farlo perché **paperoga** è presente, comunque una variabile interna di appoggio mi serve

anche **trudy** è optional, perciò gli posso passare **paperoga** anche se non fosse presente

- E' necessaria una interfaccia esplicita

intent degli argomenti

Gli argomenti di una funzione possono avere degli intent: **`in`**, **`out`**, **`inout`**, nessun intent.

- **`in`**:

  L'argomento non deve essere modificato dal sottoprogramma

- **`out`**:

  L'argomento viene reso indefinito all'inizio del sottoprogramma (ad esempio un vettore **`allocatable`** viene deallocato) ed il sottoprogramma lo deve definire in qualche modo. In particolare l'argomento deve essere definibile (ovvero non può essere una espressione).

```
call pluto(5.6)   ! vietato
subroutine pluto(a)
real, intent(out) :: a
```

- **`inout`**:

  In pratica si richiede che l'argomento sia definibile ovvero non sia una espressione.

- nessun intent:

  L'argomento deve essere definibile se lo richiede il flusso del programma.

interface statement

Lo statement interface viene usato per tre diversi scopi:

- Interfaccia esplicita per funzioni esterne ad un modulo (se fossero in un modulo basterebbe usare il modulo)
- Interfaccia generica (per procedure interne ad un modulo)
- Interfaccia astratta

interface

```
program pippo

interface
  subroutine pluto(a, b)
   real :: a, b
  end subroutine
end interface

real :: dd, ff

call pluto(dd, ff)

end program

subroutine pluto(a, b)
real :: a,b
…
end subroutine pluto
```

La routine pluto è esterna d un moduloquindi per accedere ad una sua interfaccia bisogna dichiararla in uno statement interface.

In questo caso non è necessaria visto che l'interfaccia implicita è equivalente a quella esplicita.

Però permette il controllo degli argomenti in ingresso.

E' buona norma che le routine esterne ad un modulo abbiano una interfaccia equivalente a quella implicita (seguire la convenzione del F77)

moduli come contenitori di interfacce di funzioni di libreria

Il modulo definisce una l'interfaccia di una serie di funzioni presenti in una libreria esterna.
Si presume che le funzioni di questa libreria seguano lo standard F77 (come è buona norma).

Definire un modulo di questo tipo può essere utile perché permette il controllo degli argomenti passati alle funzioni della libreria. Esempio:

```
module ftu
interface

   SUBROUTINE FUVAXA(LIO,NSHOT,CHANI,IMODE,X,Y,N,XL,YL,XU,YU,IER)
      INTEGER   LIO,NSHOT,IMODE,N,IER
      CHARACTER CHANI*(*), XL*(*), YL*(*), XU*(*), YU*(*)
      REAL      X(*),Y(*)
   END SUBROUTINE FUVAXA
    …
end interface
end module
```

abstract interface

```
program pippo

abstract interface
  subroutine trudy(a, b)
   real :: a, b
  end subroutine
end interface

procedure(trudy) :: pluto

real :: dd, ff

call pluto(dd, ff)

end program

subroutine pluto(a, b)
real :: a,b
…
end subroutine pluto
```

L'interfaccia è diventata astratta.
trudy è il nome dell'interfaccia astratta e non di una subroutine

definisco una procedura **pluto** che ha l'interfaccia **trudy**.

Questo è utile se uno ha diverse funzioni con la stessa interfaccia e nell'estensione agli oggetti del fortran

interfaccia generica

Permette di associare nomi generici a procedure specifiche, tipicamente all'interno di un modulo.

```
module pippo

interface print
   module procedure print_int
   module procedure print_real
end interface

contains

subroutine print_int(a)
  integer ::a
  print *,a
end subroutine

subroutine print_real(a)
  real ::a
  print *,a
end subroutine

end module
```

```
program gamba
use pippo

integer :: a = 2
real :: b = 1.0

call print(a)
call print(b)

end module
```

Le due procedure devono avere la lista degli argomenti distinguibile l'una dall'altra.

Il nome specifico può essere privato.

Più moduli possono specificare lo stesso nome generico, purché sia sempre soddisfatto il criterio di distinguibilità

overloading degli operatori

In una interfaccia generica si possono inserire anche operatori. Esempio :

```
module pippo

type point
   real :: x, y
end type

interface operator(+)
 module procedure s_p_p
end interface

contains

 function s_p_p(a,b) result(som)
  type(point),intent(in) :: a, b
  type(point) :: som
  som%x = a%x + b%x
  som%y = a%y + b%y
 end function

end module
```

```
program gamba
use pippo

type(point):: a = point(1.0, 2.0)
type(point):: b = point(3.0, 4.0)
type(point):: c

c =  a + b

end program
```

Seguono le stesse regole di precedenza degli operatori oroginarii

Devono avere **intent(in)** specificato.

Possono essere binari od unari.

operatori definiti dall'utente

Gli operatori definiti dall'utente sono «.operatore.». Esempio :

```
module pippo

type point
   real :: x, y
end type

interface operator(.somma.)
 module procedure s_p_p
end interface

contains

 function s_p_p(a,b) result(som)
  type(point),intent(in) :: a, b
  type(point) :: som
  som%x = a%x + b%x
  som%y = a%y + b%y
 end function

end module
```

```
program gamba
use pippo

type(point):: a = point(1.0, 2.0)
type(point):: b = point(3.0, 4.0)
type(point):: c

c =  a .somma. b

end program
```

Si dividono in unari e binari.
Quelli unari hanno la precedenza maggiore di tutti gli operatori. Quelli binari inferiore a tutti (anche di quelli intriseci come +, -, .or. , etc).

Devono avere **intent(in)** specificato.

Assegnazione definita dall'utente

Gli operatori definiti dall'utente sono «.operatore.». Esempio :

```
module pippo

type point
   real :: x, y
end type

interface assignment(=)
 module procedure setp
end interface

contains

 subroutine setp(p, a)
  type(point),intent(out) :: p
  real,intent(in) :: a
  p%x = a
  p%y = a
 end subroutine

end module
```

```
program gamba
use pippo

type(point):: a

a = 3.5

end program
```

Equivale a chiamare la subroutine:

```
call setp(p, (a) )
```

Attenzione alla parentesi sul secondo argomento che lo trasforma in una espressione.

Il primo argomento deve avere **intent(inout)** od **intent(out)**. Il secondo **intent(in)**.

Strutture di controllo

do construct

```
do i = 1, n
    …
end do
```

```
do while (…)
…
end do
```

```
do concurrent (…)
    …
end do
```

if construct

```
if (…) then
    …
end if
```

```
if (…) …
```

case construct

```
select case
    …
end select
```

branching

```
goto
```

```
if (..) 1,2,3
```

```
goto (…) …
```

select type construct

```
select type (…)
    …
end select
```

Le seguenti non sono strutture di controllo, ma assegnazioni vettoriali

where

```
where (…)
    …
end where
```

forall

```
forall (…)
    …
end forall
```

Anche queste non sono strutture di controllo

associate

```
associate (…)
    …
end associate
```

block*

```
block
    …
end block
```

*non ancora disponibile sui compilatori

do construct

Ciclo semplice:

```
do it = start, N, step
    …
end do
```

Si esegue il ciclo incrementando **it** a passi di **step**.
la variabile di ciclo **it** non deve essere modificata.
La variabile **it** deve essere intera.

Non block do:

```
do 10 i=1,N
    …
10  istruzione
```

Sconsigliato per nuovi programmi.

Ciclo while:

```
do while (condizione)
    …
end do
```

Si esegue il ciclo finche la condizione rimane vera.

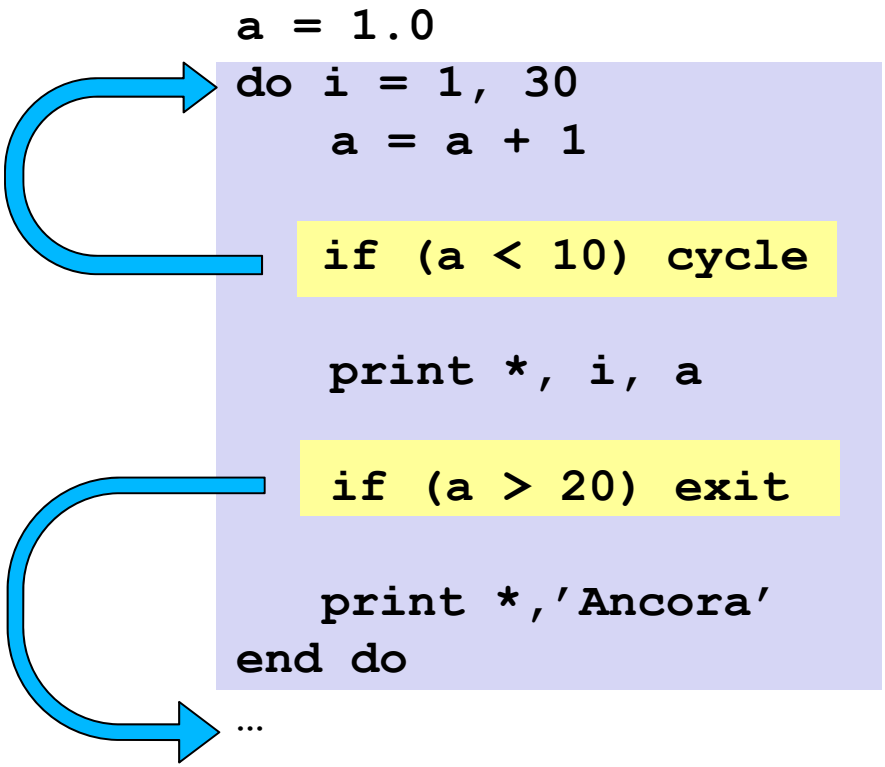
Ciclo infinito:

```
do
    …
end do
```

Il ciclo verrà eseguito all'infinito.

Uscita da un ciclo

EXIT esce da un ciclo
CYCLE vai alla successiva iterazione



```
a = 1.0
do i = 1, 30
   a = a + 1

   if (a < 10) cycle

   print *, i, a

   if (a > 20) exit

   print *,'Ancora'
end do
...
```

Queste due istruzioni permettono di uscire anche da un ciclo infinito.

Generalmente escono dal ciclo più interno a meno che non si specifichi il nome del ciclo (?).

nome di un ciclo

Ad un ciclo (e nel Fortran 2008 anche ad altri costrutti) può essere assegnato un nome.
Il nome è separato dal **do** da «:»

```
calcola_psi: do i = 1, N
   …
enddo calcola_psi
```

E' opzionale, ma se è presente deve essere uguale al nome del ciclo

Permette di uscire da cicli annidati se si usa con EXIT o CYCLE

```
primo: do i = 1, N
  …
  secondo: do j=1, M
       …
       if (…) cycle primo
       …
       if (…) exit primo
    …
    enddo secondo
enddo primo
```

Se non si specificasse il nome del ciclo si andrebbe alla successiva iterazione o si terminerebbe il ciclo più interno ovvero «**secondo**»

Esercizio

Il seguente programma legge dei numeri dalla tastiera:

```
program sommapos
implicit none
real :: somma, value
somma = 0
do
    print *,'Nuovo:'
    read(*,*) value
    …
    somma = somma + value
end do
print *, somma
end
```

Completatelo dove ci sono i tre puntini in modo che esegua la somma solo dei numeri positivi, ed esca quando il valore inserito è nullo.

Usate EXIT e CYCLE

select case

```
select case (case-expr)
case (case-value-list)
    …
case (case-value-list)
    …
case default
    …
end select
```

Il valore della *case-expr* è il **case-index**.
Il **case-index** deve essere di tipo discreto: intero, carattere, logico

non deve stare necessariamente all'ultimo posto. E non è obbligatorio

case-value-list
lista separata da virgole di espressioni di inizializzazione (calcolate durante la compilazione):

- value
- range (*start*:*stop*, *start*: , :*stop*)

```
select case (numero)
case (:-1)
   print *,'minore di zero'
case (0,3,5:7)
   print *,'o 0 o 3 o 5,6,7'
case(8:)
   print *,' >= 8 '
case default
   print *,'Quelli che restano'
   print *,' 1, 2, 4'
end select
```

Branching

- **goto** statement
  `goto` *label*
- **continue** statement (non fa nulla ma spesso usato con le label)
  label `continue`
- computed **goto** statement (obsolescente)
  `goto (`*label-list*`)` *scalar-int-expression*
- arithmetic **if** statement (obsolescente)
  `if (`*scalar-numeric-expression*`)` *label_minore_0, label_uguale_0, label_maggiore_0*

label è un numero intero posto prima di uno statement.

```
if (numero < 0) goto 10
….

10 continue
print *,'Errore  numero  deve essere > 0'
```

GO TO

- Servono i goto?
  - No, c'è un teorema che lo dimostra.
- In alcuni casi possono essere utili (e per questo non sono obsoleti).
- Uso tipico:
  - Gestire condizioni di errore

    goto in avanti, verso la fine

    delle routine .
- Evitare:

  i goto per fare cicli

```
Apri-file
if (file non aperto) goto 10
…
leggi-record
if (record non letto) goto 20
…
return
20  print *,'Errore lettura'
chiudi-file
10 print *,'Errore nel file…'
ier = -1
return
```

Esercizio

C'è un file composto di righe, il primo carattere della riga è: '+', '*', ' ', '0':

- Se '+' inizia una sezione somma (con un moltiplicatore).
- Se '*' inizia una sezione moltiplicazione (con un sommatore).
- ' ' continuazione della sezione precedente.
- '0' fine del file

Voglio calcolare (per l'esempio dato):

- Per la sezione moltiplicazione: 3*2 + 5
- Per la sezione somma: (3+4+7)*2
- Il numero di sezioni è arbitrario, ed anche il numero di righe di una sezione.
- C'è un modulo con una funzione

  `call readnext(ch, num)` dove appunto ch vale *,+, ,0. e num è il numero (che può essere il sommatore o moltiplicatore se la riga inizia con '*' e '+', mentre è il numero che va sommato o moltiplicato nelle sezioni '+' e '*'.
- Dimenticavo, non usate goto.

Esempio:

```
* 5
  3
  2
+ 2
  3
  4
  7
+ 3
  5
* 3
  7
  2
* 2
* 2
```

array

- Fixed size

```
real :: a(10), b(0:9), c(-1:8)
real, dimension(-10:5) :: d
```

I primi hanno tutti ***size*** 10, l'indice di norma inizia da 1, ma se specificato può iniziare da un altro numero. Inoltre hanno ***rank*** 1

```
real :: aa(10,15)
```

Array di ***rank*** 2. In memoria la dimensione che corre più velocemente è la prima, ovvero sono memorizzati per colonna (quasi). Ha due dimensioni di ***extent***: 10 e 15

Il numero massimo di dimensioni è 7, portato a 15 nel fortran 2008.

- Elementi di un array

Si accede agli elementi di un array specificando una serie di indici:

```
aa(5, 11)
```

- Sezione di un array. Si ottiene specificando un range per uno o più indici:

```
aa(4:7,2)      ! Array di rank 1 e size 4
aa(4:7:2, 3:10)    ! Array di rank 2 e shape [2, 8]
```

- range:

indice-iniziale:*indice-finale*:*stride*

Forme tipo **a(:5)** tutti gli elementi fino al 5 o **a(3:)** dal terzo in poi o **a(:)** tutto l'array o **a(::2)** tutto l'array a passi di due sono permesse.

Costruzione di array

- Con elementi fissi:

```
integer :: a(10)
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
a = (/ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 /)
```

```
real :: b(10)
b = [ real :: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Con un implied do

```
a = [(5*i, i=1,10)]
b = [3.2, (0.1*i, i=1, 9)]
```

- Specificare il tipo è utile con gli array di caratteri:

```
character(len=10) :: cc(3)
cc = [character(len=10)  :: 'primo','secondo', 'terzo']
```

Uno sguardo alle sezioni

Immaginiamo di voler implementare un array multidimensionale avendo a disposizione solo un array monodimensionale usando quindi degli indici. Insomma descriviamo l'array **a(8,5)** a partire dall'array **v(40)**

```
a(i,j) = v(i + NR*(j-1))
```

dove i va da 1 a NR e j da 1 a NC, riscriviamola però:

```
a(i,j) = v(1 + 1*(i-1) + NR*(j-1))
```

	NC = 5				
NR = 8	1,1	1,2	1,3	1,4	1,5
	2,1	2,2	2,3	2,4	2,5
	3,1	3,2	3,3	3,4	3,5
	4,1	4,2	4,3	4,4	4,5
	5,1	5,2	5,3	5,4	5,5
	6,1	6,2	6,3	6,4	6,5
	7,1	7,2	7,3	7,4	7,5
	8,1	8,2	8,3	8,4	8,5

Come scriviamo la sezione composta dagli elementi rossi?

```
b = a(2:8:3, 2:4:2)
ibase = 1 + 1*(2-1) + NR*(2-1) = 10
b(k,l) = v(10 + 3*1*(k-1) + 2*NR*(l-1))
```

Insomma è evidente che qualsiasi elemento di una qualsiasi sezione di un array può essere raggiunto dando il punto d'inizio dell'array, lo stride ed il numero di elementi lungo quella dimensione.
A parte questioni di efficienza non c'è una differenza tra una dimensione e l'altra, tutte possono essere trattate nello stesso modo.

Array con i tipi definiti

- I tipi definiti dall'utente possono contenere degli array ed essere essi stessi array:

```
type (xpoint)
   real :: x(3)
   integer :: info
end type
type(xpoint)   :: p(3)
```

```
p(1)%x(2)   ! scalare
```

- Od anche estrarre delle sezioni:

```
p(1:2)%x(2)
p(1:2)%info
```

! sezione (array non contiguo di reali)

! sezione (array non contiguo di interi)

è illegale invece

```
p(1:2)%x   ! ILLEGALE
p%x    ! ILLEGALE
```

- In una sequenza di designatori (… % … % …) uno al massimo può avere rank maggiore di 0.
- Alla destra di un designatore di rank maggiore di zero non può apparire un componente allocatable, o pointer.

associate construct

Nel Fortran quando si usano molte strutture una d'entro l'altra i nomi possono diventare molto lunghi. Per risolvere questo problema si può usare il costrutto **associate**. Fondamentalmente associa un nome ad una, o più, espressioni complicate.

```
type point
   real :: x, y
end type
```

```
type grid
   type(point) :: points(8)
end type
```

```
type(grid) :: griglia(5)
```

```
do i=1, 5
  do k=1, 8
    associate( x => griglia(i)%points(k)%x , &
               y => griglia(i)%points(k)%y )
      x = i*k
      y = (i-1)*(k-1)
      print *, sqrt( x**2 + y**2 )
    end associate
  end do
end do
end program
```

Array come argomenti dummy

- F77
  - Fixed size. Il numero di elementi è una espressione di specificazione.

```
subroutine pippo(b, n)
integer :: n
real :: b(n)
```

  - Assumed size: (*)

```
subroutine pippo(b)
real :: b(*)
```

  - La routine assume di ricevere l'indirizzo del primo elemento dell'array
- F90
  - Assumed shape (richiede una interfaccia esplicita): (:)

```
subroutine pippo(b)
real :: b(:)
```

  - E' molto probabile che un descrittore sia passato alla subroutine.
  - Solo lo shape viene passato, ma non i limiti inferiori e superiori.

Array come argomenti dummy

Passaggio di una sezione

- Assumed shape

```
call pippo(a(2:10:2))
subroutine pippo(a)
real :: a(:)
```

Probabilmente passo un descrittore e la funzione accede alla sezione tramite il descrittore (ma decide il compilatore cosa fare)

- Explicit size od assumed size «(*)»

```
call pippo(a(2:10:2), size(a(2:10:2)))
subroutine pippo(a, n)
real :: a(n)
```

La routine si aspetta un vettore contiguo (che non ho). Una copia temporanea viene creata (come al solito molto probabilmente, lo decide il compilatore):

```
temp = a(2:10:2)
call pippo(temp, size(temp))
a(2:10:2) = temp
```

sequence association

- Generalmente un argomento attuale deve essere uguale in shape al corrispondente elemento dummy.
- Per compatibilità con il F77 c'è una eccezione (anzi due).

```
program pippo
implicit none
real :: aa(5) = [real:: 10, 20, 30, 40, 50]
real :: bb(2,2) = reshape([real:: 11, 12, 13, 14],[2,2])

call printseq(aa, size(aa))
call printseq(bb, size(bb))

contains
  subroutine printseq(a, n)
  integer :: n
  real :: a(n)

   print*,a

  end subroutine
end
```

La matrice **bb** viene vista come un vettore all'interno della routine printseq

L'argomento dummy è explicit shape o assumed size «(*)»

Alcune funzioni

- **size(array)** numero di elementi
- **size(array, dim)** numero di elementi lungo la dimensione dim
- **shape(source)** shape di un array, ovvero array di interi con gli extent per ogni dimensione:

  real :: a(5,2:8)

  shape(a) == [5, 7]
- **lbound(array, *{*dim*}*)** lower bound
  - Se dim è presente ritorna uno scalare con il lower bound per l'indice della dimensione dim
  - Se dim è assente array di interi
- **ubound(array, *{*dim*}*)** upper bound
  - Come per il lower bound

Array expression

- Gli array possono essere usati nelle espressioni. Tutti gli array devono essere conformabili ovvero avere lo stesso shape. Gli scalari si comportano come uno si aspetta:

```
real :: a(5,10), b(5), c(7,11), d
a(:,7) * b + c(3,2:6)*d
```

Questa espressione equivale a:

```
a(1,7)*b(1) + c(3,2)*d
…
a(5,7)*b(5) + c(3,6)*d
```

- Vector subscript:

Una particolare sezione può essere ottenuta passando un vettore di interi:

```
a([1,3],[5,7,8])
```

a(1,5)	a(1,7)	a(1,8)
a(3,5)	a(3,7)	a(3,8)

Un array con un vector subscript non può essere passato ad una routine che lo dichiara con l'intent «out» od «inout». E comunque non è definibile dalla stessa routine.

Funzioni intrinsiche

- Molte funzioni intrinseche come sin, cos, etc. sono **elemental** (come uno si aspetterebbe):

```
real :: theta(12), co(12), si(12)
real :: dth = 0.2618
theta = [(dth*i,i=0,11)]
co = cos(theta)
si = sin(theta)
```

- Agiscono sui singoli elementi di un array. Accettano in input scalari od array di qualsiasi dimensione.

Array assignment

Esempio:

```
real :: a(10), b(11,5)
a = b(2:11,1)**2
```

L'array che viene definito (**a** in questo caso) deve avere lo stesso shape della espressione (per i vettori allocatable che vedremo in seguito c'è una differenza).

L'espressione viene completamente calcolata prima di eseguire l'assegnazione:

```
integer :: a(5) = [1,2,3,4,5]
a(2:) = a(:4)
```

```
a == [1,1,2,3,4]
```

Produce: `[1,1,2,3,4]` E non è equivalente a:

```
a(2)=a(1); a(3)=a(2); a(4)=a(3); a(5)=a(4)
```

Si possono usare vector substrict ma solo se non contengono indici ripetuti:

```
a([1,3,4]) = …
```

allocatable

- Una variabile può essere dichiarata allocatable, se è un vettore se ne specifica il rank ma non i limit:

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

- La variabile non dispone di memoria associata all'inizio del programma. La memoria viene allocata esplicitamente durante l'esecuzione del programma.
- Se è un vettore al momento dell'allocazione si specificano i limiti:

```
allocate(a, b(5), c(2:7, 8))
```

- Se la variabile esce di scopo:
  - una variabile locale di una routine senza save.
  - una variabile temporanea, etc.

  La variabile viene automaticamente deallocata
- Una variabile allocatable può far parte di un tipo definito dall'utente:

```
type pluto
   real, allocatable :: a(:)
end type
```

- Un argomento dummy di una subroutine può essere allocatable (l'argomento attuale deve essere allocatable).
  - Attenzione se ha intent(out) viene automaticamente deallocato

allocatable

- Una variabile allocatable può stare in due stati:
  - allocata
  - non allocata
    - Tipicamente una variabile inizia la sua vita come non allocata
- Per testare se una variabile è allocata:

```
if (allocated(a)) …
```

- Forma completa di allocate:

```
allocate(var, stat=stat, errmsg=msg, &
         source=source, mold=mold)
```

source: dopo l'allocazione la variabile viene riempita con i valori di source (F2003), non è necessario specificare i limiti di a (F2008)

mold: si alloca una variabile tipo mold, ma non la si riempie (F2008)

- Deallocare:

```
deallocate(a, stat=stat, errmsg=msg)
```

- *stat*: un intero, *msg*: una variabile carattere

Allocatable

- F2003: Riallocazione automatica negli assignment. Generalmente disponibile con una opzione del compilatore perché cambia la semantica.

```
integer, allocatable :: a(:)
a = [1,2,3,4]
a = [10,20,30,40,50,60]
```

La variabile se non ha le dimensioni corrette viene automaticamente deallocata e riallocata

- Spostamento dell'allocazione

pure subroutine move_alloc(from, to)

Esempio:

```
integer, allocatable :: a(:), b(:)
allocate(a(5))
a = 6
call move_alloc(from=a, to=b)
print *,b
```

Sposta l'allocazione da «from» a «to». Questo non comporta copiare i dati.

pointer e target

- Una variabile pointer si può trovare in tre stati:
  - associato
  - disassociato
  - indefinito
- pointer in stato indefinito
  - tipicamente un pointer inizia a vivere in questo stato
  - Un pointer in questo stato può essere usato in pochissime situazioni:
  - Attenzione perché un pointer può diventare indefinito a seconda del flusso delle istruzioni.
- Una variabile dichiarata pointer non ha una memoria propria come gli allocatable:
  - Può essere allocato tramite lo statement allocate. Ma non viene automaticamente deallocato quando esce di scopo (memory leak)

```
real, pointer :: a(:)
allocate(a(7))
```

  - Può essere associato ad una variabile target

```
real, pointer :: pa(:)
real, target :: a(8)
pa => a
```

pointer e target

- Pointer assignment
  - Il pointer può anche essere associato con una sezione del target (e subobject)

```
real, target :: a(8)
real, pointer :: pa(:)
pa(:) => a(2:6:2)
```

  - Per essere puntate da un pointer le variabili devono avere l'attributo TARGET.
- Un pointer può essere reso disassociato:
  - Durante l'inizializzazione con null()

```
real, pointer :: a(:) => null()
```

  - Tramite lo statement nullify

```
real, pointer :: a(:)
nullify(a)
```

Stato di unpointer

- Per controllare lo stato di un pointer funzione:

  associated(pointer, *{*target*}*)

  Ritorna .true. se il pointer è associato (con lo specifico target, se specificato)
- Attenzione non c'è nessun modo di controllare lo stato di un pointer che è indefinito:
  - Se non inizializzato è in stato indefinito
  - Se la memoria a cui punta cessa di essere valida.

Argomenti allocatable e pointer

- Un argomento dummy di una routine può essre definito **pointer**, **allocatable** o **target**.
- Ad un argomento dummy **pointer** o **allocatable** può essere solo associato rispettivamente un argomento attuale **pointer** o **allocatable**.
- L'uso tipico è di allocare l'array nella routine della dimensione necessaria:

```
real, allocatable :: a(:)
…
call pluto(a)

subroutine pluto(a)
real, allocatable :: a(:)
…
allocate(a(n))
…
end subroutine
```

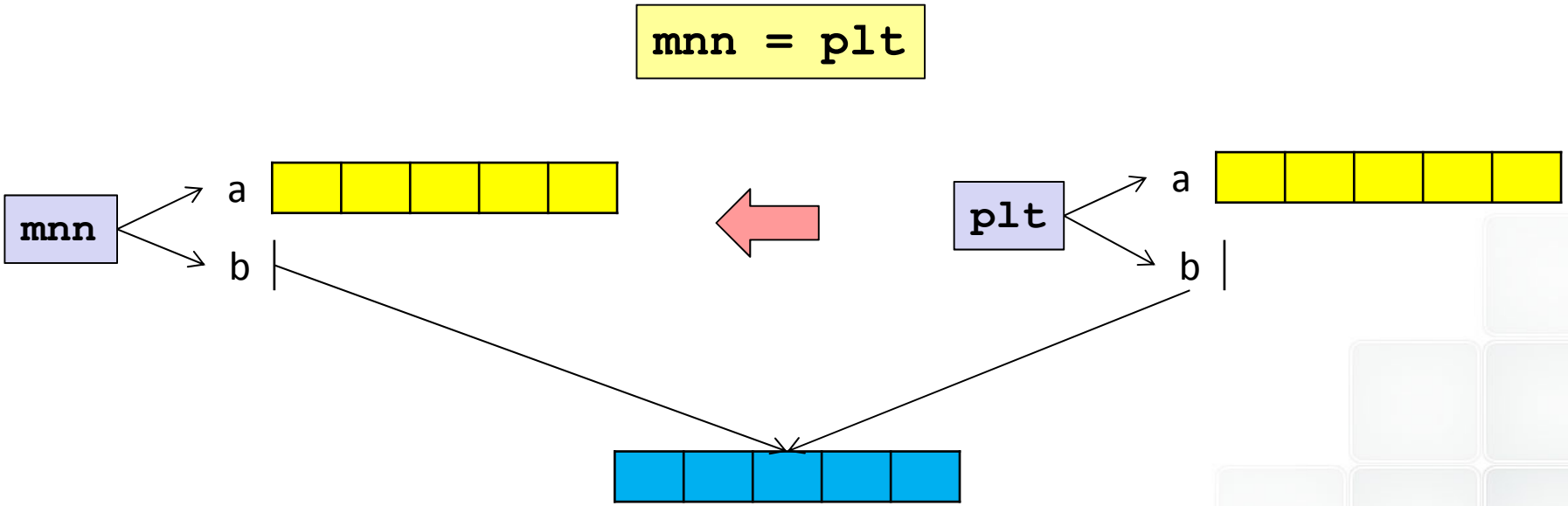
- Le sezioni di un array (**pointer** od **allocatable**) non mantengono l'attributo **pointer** od **allocatable**, e quindi non possono essere associate ad argomenti dummy **pointer** od **allocatable**.
- Per quel che riguarda l'attributo **target** tutte le possibili combinazioni possono essere valide. Ma la cosa può essere parecchio confusa.

Tipi definiti on allocatable e pointer

```
type pluto
   real, allocatable :: a(:)
   real, pointer :: b(:)
end type
```

```
type(pluto) :: plt
type(pluto) :: mnn

allocate(plt%a(5), plt%b(5))
```

L'allocatable viene copiato completamente, nel caso riallocandolo opportunamente.
Il pointer viene copiato ma non la memoria a cui punta.



procedure pure

- Una procedura pura non ha «side effects».
  - Usata:
    - parallel processing (in costrutti potenzialmente paralleli)
    - specification
  - Non scrive su file
  - Gli **intent** degli argomenti devono essere specificati
  - Se una funzione, gli **intent** devono essere **in**. Solo il risultato in uscita cambia.
  - Non può modificare nessuna variabile esterna.
  - Non può avere variabili locali con il save implicito (inizializzate), od esplicito.
- Si dichiara con «pure» davanti:

```
pure subroutine(a, b)
real, intent(in) :: a
real, intent(out) :: b
b = a*2
end subroutine
```

procedure elemental

- Sono pure
  - F2008: con «inpure» davanti possono anche non esserlo.
- Gli argomenti devono essere dichiarati scalari.
- Il risultato delle **function** deve essere dichiarato scalare.
- Non deve essere ricorsiva
- Gli argomenti «dummy» ed il risultato non devono essere né pointer né allocatable.
- Gli argomenti attuali devono essere conformabili (o scalari o se array con lo stesso shape).
- Se un argomento attuale corrisponde ad argomento con intent **out** od **inout**, deve essere un array (ovviamente).
- Deve avere interfaccia esplicita.
- Si dichiarano con «**elemental**».

procedure elemental esempio

```
module pippo
implicit none
real, parameter :: pi = 3.14159265
real, parameter :: mu0 = 4E-7*pi

type vector
    real :: x, y
end type

contains
elemental function magfield(pos, current)
    type(vector),intent(in) :: pos
    real,intent(in) :: current
    type(vector) :: magfield

    real :: r2

    r2 = pos%x**2 + pos%y**2

    magfield%x = -mu0*current/2.0/pi/r2 * pos%y
    magfiled%y =  mu0*current/2.0/pi/r2 * pos%x

end function
end module
```

```
program topo
use pippo
implicit none
integer, parameter :: n = 12

type(vector) :: pos(n)
type(vector) :: b(n)

real :: th(n)
real :: dth = 2*pi/n
real :: r0 = 2.3
real :: current = 1.6
integer :: i

th = [(dth*i,i=0,n-1)]

pos%x = r0*cos(th)
pos%y = r0*sin(th)

b = magfield(pos, current)

end program
```

Funzioni numeriche

- Trigonometriche:
  - sin, cos, tan, sinh, cosh, tanh, asin, acos, atan, atan2
    - atan2(y,x)
- Radice quadrata e logaritmi
  - sqrt, log, log10, exp
- varie
  - abs, ceiling, floor, dim, dprod, mod, modulo, sign
    - dim(x,y) : x-y se (x-y)>0 se no 0
    - dprod(x,y): x*y ma in doppia precisione
- Matriciali:
  - dot_product, matmul, transpose

Aliasing

- Quando una variabile viene acceduta attraverso nomi diversi:
  - Come argomento di una procedura e attraverso un modulo od un common.
  - Attraverso due argomenti della stessa procedura.
- In generale l'aliasing è vietato!
  - E' a cura del programmatore.
  - I **pointer** permettono l'aliasing per costruzione (in alcuni casi). Questo può rendere il codice meno efficiente, è per questo motivo che un **pointer** può puntare solo ad oggetto dichiarato **target**.

Aliasing Esempio

```
integer, target :: a(5) = [1,2,3,4,5]
integer, pointer :: pa(:)
```

```
a(2:) = a(:4)
```
⟹ `a == [1,1,2,3,4]` ⟵ il compilatore provvede a calcolare correttamente la parte destra dell'espressione prima di assegnarla

```
subroutine aliasass(a, b)
integer :: b(:), a(:)

  a(2:) = b(:size(b)-1)

end subroutine
```
⟵ il compilatore non si aspetta aliasing, lo potrebbe trasformare in un semplice loop

```
call aliasass(a,a)
```
⟹ `a == [1,1,1,1,1]` ⟵ **ifort** versione 13. Effettivamente lo ha trasformato in un loop.

```
pa => a
a(2:) = pa(:4)
```
⟹ `a == [1,1,2,3,4]` ⟵ alias legittimo

```
call aliasass(a,pa)
```
⟵ alias illegittimo anche se uso il pointer

```
call aliasass(a,(a))
```
⟵ nessun alias: **(a)** è una espressione

Alias: di peggio

```
integer :: a(5) = [1,2,3,4,5]
```

```
call aliasxxx(a,a(2::2))
```

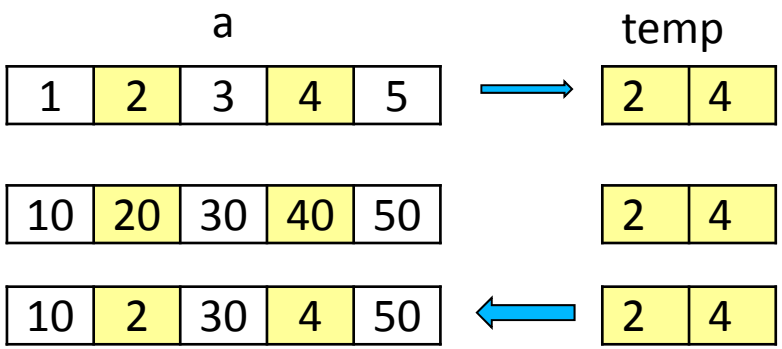
```
a == [10,2,30,4,50]
```

```
subroutine aliasxxx(a, b)
integer :: a(:), b(*)
integer :: i

 a = [(i*10, i=1:size(a))]

end subroutine
```

l'argomento **b** è assumed size ovvero contiguo, per passargli una sezione il compilatore crea una variabile temporanea

```
call aliasxxx(a,a(2::2))
```

```
temp = a(2::2)

call aliasxxx(a,temp)

a(2::2) = temp
```



storage association

- equivalence e common:

  Le variabili hanno uno storage in comune:

```
real :: a, b(5)
integer :: c
equivalence (a, b(3)), (c, b(4))
```

  a e **b(3)** puntano alla stessa locazione di memoria, come **c** e **b(4)**, che viene reinterpretata a seconda del tipo di variabile.

- Solo i seguenti tipi (tra i numerici) possono (secondo lo standard) essere messi in equivalenza tra di loro: default integer, real, double precision, complex, logical

```
integer :: a
integer(2) :: b
equivalence (a, b)  ! Non standard
```

- I common definiscono uno storage che viene usato da variabili in diverse routine (usato prima dei moduli):

```
subroutine pippo
integer :: a
real :: b
common  /topo/ a, b
…
subrotine pluto
integer :: aa
real :: bb
common /topo/ aa, bb
```

Routine Matematiche Addizionali (F2008)

- Error function (x reale)
  - **erf**(x), **erfc**(x), **erfc_scaled**(x) [==exp(x**2)*erfc(x)]
- Iperboliche inverse (x reale o complesso):
  - **acosh**(x), **asinh**(x), **atanh**(x)
- gamma (x reale)
  - **gamma**(x), **log_gamm**a(x) [==log(abs(gamma(x)))]
- Bessel function (x reale)
  - **bessel_j0**(x), **bessel_j1**(x), **bessel_jn**(n,x)
  - **bessel_y0**(x), **bessel_y1**(x), **bessel_yn**(n,x)
  - **bessel_jn**(n1, n2, x), **bessel_yn**(n1, n2, x)
- Altre
  - **hypot**(x, y) [== sqrt(x**2+y**2)] Senza over/under-flow non necessari.
  - **norm2**(x, {dim}): norma L_2 di un array.

Funzioni sui caratteri

len(string)	lunghezza di una variabile carattere
len_trim(string)	lunghezza di una variabile carattere dopo aver rimosso i bianchi finali
adjustl(string)	sposta la stringa a sinistra riposizionando i bianchi iniziali alla fine
adjustr(string)	sposta la stringa a destra riposizionando i bianchi finali all'inizio
index(string, substr, {back})	ricerca substr in string e riporta la posizione iniziale. Dalla fine se back=.true.
repeat(string, ncopies)	Concatena **string ncopies** volte.
scan(string, set, {back})	Ricerca in una stringa uno qualsiasi dei caratteri in set e ne dà la posizione

trim(string)	La stringa ma senza i bianchi finali (utile per i nomi dei file):
	`print *, trim('Ciao come stai    ')//';'`
	`Ciao come stai;`

verify(string, set, {back})	Il primo (o l'ultimo) carattere in **string** che non è in **set**.
achar(i, {kind})	Carattere corrispondente all' i-esima posizione nella tabella ASCII.
char(i, {kind})	Simile ad achar ma nel set di caratteri definito da kind (in pratica uguali).
iachar(c)	Posizione nella tabella ASCII del carattere **c**.
ichar(c)	Come sopra ma nel set di caratteri di **c** (in pratica uguali).

Funzioni di conversione

aimag(z)	Parte immaginaria di un numero complesso
aint(a, {kind})	Troncamento verso lo zero di un numero
anint(a, {kind})	arrotondamento al numero intero più vicino (ma il risultato è reale)
cmplx(x, {y}, {kind})	numero complesso di tipo default se kind non è specificato.
conjg(z)	coniugato di un complesso
dble(a)	conversione in doppia precisione
int(a, {kind})	conversione in un intero (di tipo kind se specificato).
logical(l, {kind})	Conversione tra diversi tipi di logici
nint(a, {kind})	Intero più vicino
real(a, {kind})	Trasformazione in un reale (di default kind se non specificato, eccetto per i complessi). Se l'argomento è complesso la parte reale dell'argomento (del kind del complesso).

Numeri random ed altro

- Numeri random:
  - Una funzione **random_number** genera numeri random:

```
real :: a(10), b(20,20)
call random_number(a)
call random_number(b)
```

  - **random_seed**({size, put, get}) per impostare o recuperare il seed.
- Tempi vari
  - **cpu_time**(time): tempo di cpu.
  - **date_and_time**({date,time, zone, values}): varie informazioni sull'ora locale.

Funzioni per gli Array

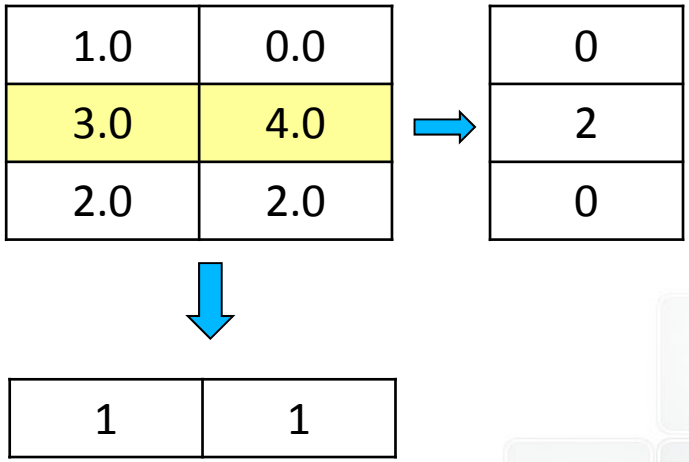
count(mask, {dim})	conta il numero di valori .true. in mask
all(mask, {dim})	ritorna .true. se tutti gli elementi di mask sono .true.
any(mask, {dim})	ritorna .true. se almeno uno dei valori di mask è .true.
sum(array,{dim},{mask})	somma gli elementi dell'array
product(array,{dim},{mask})	prodotto degli elementi dell'array
maxloc, **minloc**(array,{dim},{mask})	posizione del massimo/minimo di un array
maxval, **minval**(array,{dim},{mask})	valore del massimo/minimo di un array
merge(tsource, fsource, mask)	merge di due vettori a seconda di mask
spread(source, dim, ncopies)	replicazione di una array lungo una dimensione aggiuntiva
reshape(source, shape, {pad},{order})	Cambia la dimensione di una array
pack(array, mask, {vector})	Ritorna un array con gli elementi corrispondenti a quelli .true. di mask
unpack(vector, mask, field)	L'incontrario di pack
transfer(source, mold, {size})	Il risultato ha le stesse caratteristiche fisiche di source ma reinterpretato come mold. Può sostituire equivalence.
cshift(array, shift, {dim})	Shift circolare degli elementi di un array

count

```
program pippo
implicit none
real :: a(3,2)

a = reshape([real:: 1.0, 3.0, 2.0, 0.0, 4.0, 2.0],[3,2])

print *, count(a>2.0)
print *, count(a>2.0, dim=1),
print *, count(a>2.0, dim=2)

end
```

Stampa:

```
     2
     1      1
     0      2      0
```



all, any

```
program pippo
implicit none
real :: a(3,2)

a = reshape([real:: 1.0, 3.0, 2.0, 0.0, 4.0, 2.0],[3,2])

print *, all(a>2.0), ' : ', any(a>2.0)
print *, all(a>2.0, dim=1), ' : ', any(a>2.0, dim=1)
print *, all(a>2.0, dim=2), ' : ', any(a>2.0, dim=2)

end
```

Stampa:

F : T

F F : T T

F T F : F T F

1.0	0.0
3.0	4.0
2.0	2.0

all	any
F	F
T	T
F	F

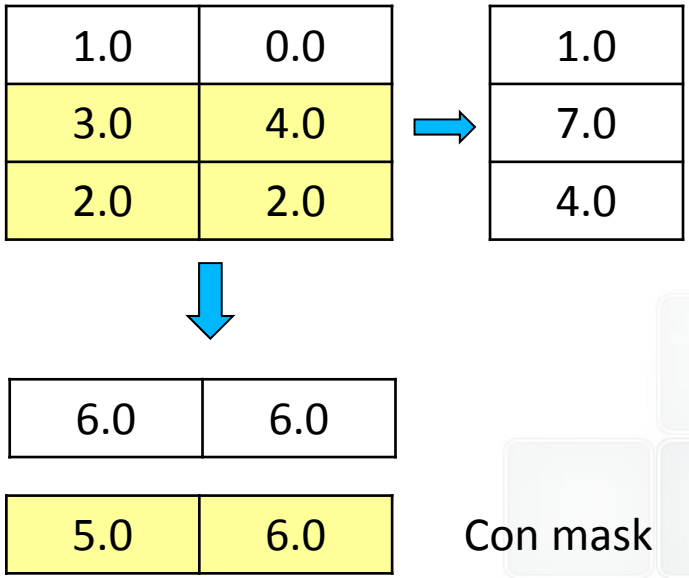
all	F	F
any	T	T

sum, product

```
program pippo
implicit none
real :: a(3,2)

a = reshape([real:: 1.0, 3.0, 2.0, 0.0, 4.0, 2.0],[3,2])

print *, sum(a)
print *, sum(a, dim=1)
print *, sum(a, dim=2)
print *, sum(a, dim=1, mask=a>1.0)

end
```

Stampa:

```
      12.0
      6.0       6.0
      1.0       7.0       4.0
      5.0       6.0
```



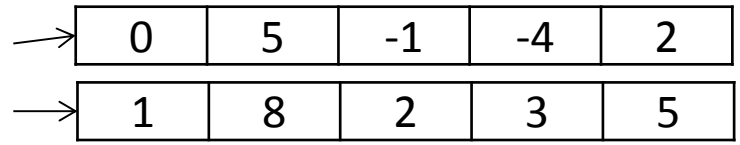
product è simile ma fa il prodotto

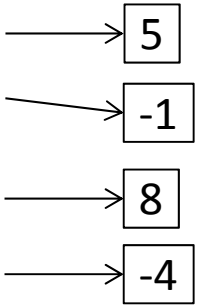
maxval, minval, max, min

Attenzione c'è una sostanziale differenza tra maxval,minval e max, min:

- maxval, minval: valore massimo o minimo di un array
- max, min: valore massimo o minimo tra l'elenco degli argomenti
- Come per sum, etc, l'argomento dim, e mask permettono di specificare una dimensione lungo cui cercare il massimo ed una matrice di maschera.

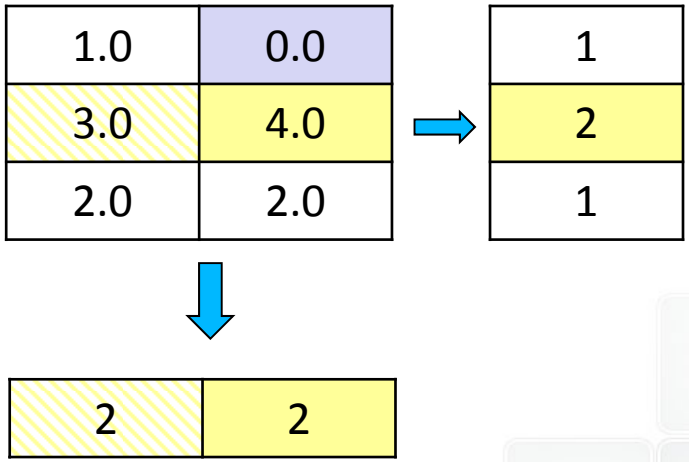
```
program pippo
implicit none
integer :: a(5) = [ 1, 5, -1,  3, 2]
integer :: b(5) = [ 0, 8,  2, -4, 5]

print *, min(a,b)
print *, max(a,b)

print *, maxval(a)
print *, minval(a)

print *, maxval(b)
print *, minval(b)

end
```





minloc, maxloc

```
program pippo
implicit none
real :: a(3,2)

a = reshape([real:: 1.0, 3.0, 2.0, 0.0, 4.0, 2.0],[3,2])

print *, minloc(a)
print *, maxloc(a)

print *, maxloc(a, dim=1)
print *, maxloc(a, dim=2)

end
```



Stampa:

1 2 Un vettore di indici
2 2 Un vettore di indici
2 2
1 2 1

maxloc, minloc

- Attenzione:

```
integer :: a(7) = [1,2,3,4,3,2,1]
```

maxloc(a) → è un vettore di lunghezza 1
maxloc(a, dim=1) → è uno scalare

```
integer :: id
id = maxloc(a) ! Errore
id = maxloc(a, dim=1) ! ok
```

Che si può semplificare in:

```
id = maxloc(a,1)
```

where (Assignment mascherato)

- Assignment vettoriale mascherato

where (*condizione*) *array-assignment*

where (*condizione*)
array-assignments
elsewhere (*condizione2*)
…
elsewhere
…
end where

Si possono usare funzioni non elemental all'interno di *array-assignments*.
A queste viene passato l'intero array.
L'assegnazione soltanto segue le regole del where. (Esercizio).

- Esempio:

```
real :: a(10,20), b(10,20), c(10,20)
…
where (a /= 0.0)
    b = 1.0/a
elsewhere
    b = 0.0
endwhere
…
where (a >= 0) c = sqrt(a)
```

forall

- Assignment parallelo indicizzato:

```
forall (i=1:N, j=1:M, i>j)
   a(i,j) = i+j
   b(j,i) = a(i,j)
end forall
```

- Attenzione, si deve pensare ad una esecuzione parallela. Prima viene completamente eseguito il calcolo di **a** e poi si calcola **b**:

```
forall( i=2:n-1, j=2:m-1)
   a(i,j) = (a(i+1,j)+a(i-1,j)+ &
             a(i,j+1)+a(i,j-1))/4.0
   b(i,j) = 1.0/a(i+1,j+1)
end forall
```

- Solo funzioni pure possono apparire nel corpo del forall (ma anche costrutti where e forall interni).
- Le variabili indice sono interne al forall e non influenzano variabili dal nome uguale presenti esternamente

do concurrent

- Assomiglia al forall ma è un vero ciclo.

```
do concurrent (i=1:n, j=1:m, i/=j)
   …
end do
```

- Solo procedure pure possono apparire all'interno.
- **exit** o **cycle** sono vietati, come **return** per uscire dalle subroutine.
- Fondamentalmente si indica al compilatore che quel particolare ciclo può essere parallelizzato senza problemi (e si guadagnerebbe a farlo).

Input Output «open»

- Aprire un file:

```
integer :: lun  = 57
open(unit=lun, file='michiamocosii.txt')
open(newunit=lun, file='michiamocosii.txt')
```

- I file in fortran sono specificati da una unità logica che è un numero intero.
- Nel F77 si doveva trovarne una libera, nel F2008 si può usare l'opzione **newunit** nella open che sceglie automaticamente una libera.
- Nelle open appaiono molti specificatori.

Alcuni specificatori di open

- **unit**: unità logica, un intero
- **file**: nome del file. i trailing blank non sono considerati
- **action**: cosa si può fare sul file (tipicamente si lascia in bianco a meno che non si voglia solo leggere il file)
  - ‘READ’: si può solo leggere dal file
  - ‘WRITE’: si può solo scrivere
  - ‘READWRITE’: leggere e scrivere
- **form**: Che tipo di formato ha il file
  - ‘UNFORMATTED’: tipicamente binario
  - ‘FORMATTED’: di testo
- **status**:
  - ‘UNKNOWN’: non si specifica se esiste o no (default)
  - ‘OLD’: il file deve esistere
  - ‘NEW’: il file non deve esistere
- **err**: una label dove viene trasferito il controllo se c’è un errore
- **iostat**: una variabile intera che se l’apertura è andata a buon fine ritorna 0

Record oriented

- I file in fortran si dividono record.
- Il record coincide tipicamente con una linea per i file formatted.
- Per i file unformatted la lunghezza del record viene scritta sul file insieme ai dati.
- Questo rende non immediata la lettura da parte di programmi scritti in altri linguaggi di file unformatted scritti da un programma fortran.
- Nel F2003 si possono aprire i file con accesso ‘STREAM’ (specificatore **access**=‘stream’).

scrivere leggere (formattato)

- Scrivere:
  - **print** *formato*,
  - **write**(*lun*, *formato, {***isostat=***ios}*) …
- leggere:
  - **read**(*lun*, *formato*, *{***isostat=***ios}*) …
- dove *formato*:
  - * : libero, segue la lista delle variabili
  - *label*: una label di una istruzione format
  - "(…)": una stringa di formato
- *lun*:
  - una effettiva unità logica di un file aperto
  - *: l'input o l'output di default a seconda se leggo o scrivo
  - una variabile carattere.
- Esempio:

```
print "(4I5)", 12, 13 14 15
character(12) :: str
write(str, *, iostat=ios) 13.5, 7
```

Istruzione format ed un unità di default

- Invece di specificare il formato come una stringa si può usare una label e l'istruzione format:

```
print "(4I5)", 12, 13 14 15
```

equivale a

```
print 10, 12, 13 14 15
10 format(4I5)
```

- Le unità logiche corrispondenti all'input e l'output di default (stdin, stdout, per intenderci) corrispondono alle unità 5, 6.

```
write(*,*) 'Ciao'
write(6,*) 'Ciao'
```

- Si può usare il modulo intrinseco **iso_fortran_env** per saperlo (ifort 13):

```
use, intrinsic :: iso_fortran_env
print *, input_unit  ⟶ 5
print *, output_unit ⟶ 6
print *, error_unit  ⟶ 0
print *, iostat_end  ⟶ -1
```

Aggiunte F2003, F2008

- Per tipi definiti dall'utente si possono definire delle routine che provvedono a formattare gli stessi (ancora non diponibile)
- Input output ricorsivo
- Input output asincrono
- Due funzioni:
  - **is_iostat_end**(ios): .true. se alla fine del file
  - **is_iostat_eor**(ios): .true. se alla fine di un record
- G0 edit descriptor:

```
print '(3G0)', 1, 3, 14.56
```

scrive le variabili dandogli solo lo spazio che serve:

```
1314.5600
```

(In questo caso magari non è una buona idea)

- unlimited format item (*)

```
write(10, '("Array = ", *(I0,:, ", "))') iarray
```

il separatore : indica di finire a scrivere se non ci sono altri dati. Scrive:

```
Array = 12, 3, 124, 58
```

E non è finito qui

- Abbiamo sorvolato sulle variabili carattere.
- Abbiamo sorvolato sull'input-output.
- Supporto delle estensioni dei tipi definiti.
- Programmazione Orientata agli Oggetti.
- Sottomoduli.
- Co-array per il calcolo parallelo.
- Vari moduli intrinseci.
- Interoperabilità con il C.
- Funzioni di manipolazione dei bit avanzate.

- Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Vivamus et magna. Fusce sed sem sed magna suscipit egestas.
- Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Vivamus et magna. Fusce sed sem sed magna suscipit egestas.