Breve storia del FORTRAN

- Il *FORTRAN* (= FORmula TRANslator) nacque nel 1957. Nel 1980 venne rilasciato il *FORTRAN77* (file sorgente "`.for`"): si trattava di un linguaggio ad alto livello molto intuitivo e semplice utile alla realizzazione di algoritmi per calcoli matematici.
- Da molti fisici e matematici veniva preferito al *C* (più rigido nella sintassi e più vicino alla architettura del sistema) anche se presentava alcuni limiti rispetto ad esso.
- Nel tempo nuove caratteristiche vennero aggiunte al FORTRAN77 fino ad arrivare al Fortran90 (file sorgente "`.f90`") nel 1991: formato libero, allocazione dinamica della memoria, puntatori, tipi di dati definiti dall'utente, moduli, funzioni ricorsive, funzioni incorporate per la gestione degli array.
- Permettendo una completa compatibilità con il FORTRAN77 molti sviluppatori rimasero legati a tale linguaggio.

Sintassi

- A differenza del C il F90 non è *case-sensitive*.
- Come il C anche il F90 presenta un formato libero (con alcune limitazioni):
  - le righe possono estendersi al massimo per 132 caratteri;
  - è possibile andare a capo (per un massimo di 39 volte) terminando la riga con "&" (un altro segno "&" ad inizio della riga successiva permette una migliore indentazione);
  - non è necessario terminare le istruzioni con il "`;`" ma questo può essere utilizzato per seprare due istruzioni sulla stessa riga;
  - i commenti devono essere preceduti dal simbolo "`!`" e terminano a fine riga (sono consentiti anche i commenti in linea);
  - i blocchi di istruzioni non sono racchiusi dalle parentesi graffe "{}" ma dala istruzione specifica di termine del costrutto.

Struttura di un programma

- A differenza del C, il F90 non richiede l'inclusione esterna di librerie (come `math.h` o `stdio.h`).
- Il F90 prevede l'utilizzo di subroutine o funzioni; i prototipi di funzioni del C sono sostituiti dai costrutti `INTERFACE` (non sempre obbligatori).
- Il programma inizia con il costrutto `PROGRAM <nome-programma>` (il nome del programma può essere costituito da lettere, underscore e cifre per un massimo di 31 caratteri) e termina con il costrutto `END [PROGRAM [<nome-programma>]]` (ove tra parentesi quadre sono raccolte informazioni ausiliarie non obbligatorie).
- Il programma si apre con una sezione dichiarativa (inclusione di moduli, interfaccia a funzioni o subroutine, definizione di variabili, definizione di tipi).
- Segue la sezione esecutiva all'interno della quale il flusso di istruzioni viene gestito tramite costrutti analoghi a quelli del C.

Struttura di un programma

- C:

```
#include <stdio.h> /* inclusione di librerie */
#define n 10       /* definizione di macro */
main()             /* funzione principale */
{  <dichiarazioni>
   <istruzioni>    }
```

- F90:

```
PROGRAM <nome-programma>
  USE <nome-modulo>
  IMPLICIT NONE
  <dichiarazioni>
  <istruzioni>
END PROGRAM <nome-programma>
```

Dichiarazioni

- Si consiglia l'impiego del comando `IMPLICIT NONE`: una volta invocato, le variabili di programma devono essere dichiarate esplicitamente (come nel C).
- I nomi di variabili devono cominciare con una lettera e possono contenere lettere, underscore e cifre per un massimo di 31 caratteri.
- *Nota*: Poiché il F90 non è case-sensitive le variabili A e a non sono distinguibili.
- Il costrutto per la dichiarazione di una variabile è:
  `TipoVariabile[,Opzioni]::<nome>[=<valore>]`
- *Esempio*:
  `Real :: VariabileReale=3.1`
  definisce una variabile reale di nome "`VariabileReale`" con valore iniziale assegnato di 3.1.

Tipi di variabili

C	F90	Significato
`int`	`INTEGER`	variabile intera
`float`	`REAL`	variabile reale in singola precisione
`double`	`DOUBLE PRECISION`	variabile reale in doppia precisione
`char`	`CHARACTER`	variabile carattere (stringa)
-	`LOGICAL`	variabile booleana ("`.TRUE.`" o "`.FALSE.`")
-	`COMPLEX`	variabile complessa ($x + iy$)

Opzioni

Opzione	Significato
`PARAMETER`	il valore assegnato alla variabile nella dichiarazione non è modificabile nel seguito
`DIMENSION(n,m, ...)`	la variabile è dichiarata come array di dimensioni n,m, ... (sono possibili anche altri metodi per la dichiarazione di array)
`(LEN=n)`	può seguire solo una dichiarazione character e sta ad indicare la lunghezza della stringa
`(KIND=n)`	può seguire qualsiasi tipo di variabile consentendo di modificare la precisione e il range di validità della stessa

Esempi

```
INTEGER                    :: i,j,k
REAL                       :: x,y,z,raggio
DOUBLEPRECISION, PARAMETER :: pi=3.14159
REAL,DIMENSION(10)         :: array1,array_2
CHARACTER(LEN=10)          :: NomeCognome
CHARACTER(LEN=*),PARAMETER :: citta="Bologna"
LOGICAL                    :: veroOfalso
```

Operatori matematici

- Gli operatori di post/pre - incremento/decremento (`++x, x++, --x, x--`) e di auto-riassegnazione (come `x+=1`) del C non trovano corrispondenza diretta in F90.
- Per le altre operazioni più comuni vale la seguente tabella di conversione:

C	F90	Significato
`x+y`	`x+y`	addizione
`x-y`	`x-y`	sottrazione
`x*y`	`x*y`	moltiplicazione
`x/y`	`x/y`	divisione
`powf(x,y)`	`x**y`	elevamento a potenza
`fabs(x)`	`ABS(x)`	valore assoluto

Operatori logici

- Gli operatori logici del C trovano corrispondenza anche nel F90.
- Per ragioni di compatibilità con il F77, il F90 prevede anche l'utilizzo di due forme di operatori per il confronto di variabili

C	F90	C	F90
`==`	`==` oppure `.EQ.`	`&&`	`.AND.`
`!=`	`/=` oppure `.NE.`	`||`	`.OR.`
`<`	`<` oppure `.LT.`	`!`	`.NOT.`
`<=`	`<=` oppure `.LE.`		
`>`	`>` oppure `.GT.`		
`>=`	`>=` oppure `.GE.`		

Esempi

```
REAL              :: x,y,z=3
INTEGER,PARAMETER :: n=2
z=x**n+ABS(y)-z   !commento: z viene sovrascritta;
                  ! viene utilizzato n=2 per l'ele-
                  ! vamento a potenza


LOGICAL :: test
REAL    :: x,y
test=(ABS(x)>=ABS(y))&  !commento: test risulta
     &.AND.(x*y<0)      ! .TRUE. se sono verificate
                        ! entrambe le condizioni
```


Costrutti per il controllo di flusso

- La maggior parte dei costrutti del C trovano corrispondenza con costrutti analoghi del F90.
- Mentre in C un blocco di istruzioni (ad esempio, condizionato dalla verifica di un `if` o all'interno di un ciclo `for`) è racchiuso da parentesi graffe "{","}", in F90 il costrutto è costituito da una istruzione di apertura ed una di chiusura.
- Per un maggior ordine formale, ad ogni costrutto del F90 può essere assegnato un nome simbolico in apertura che per essere poi richiamato in chiusura.

```
[<nome-simbolico>:] nomeCOSTRUTTO [<opzioni>]
    <istruzioni>
    ...
END nomeCOSTRUTTO [<nome-simbolico>]
```

Ciclo `for` - `DO...END DO`

- C: ciclo `for` per `index` da `start` a `end` con passo `increment`

```
for(index=start;i<=end;i+=increment)
{
  <istruzioni>
  ...
}
```

- F90: ciclo `DO` per `i` da `1` a `j` con passo `k`

```
[<nome>:]DO index=start,end[,increment]
  <istruzioni>
  ...
END DO [<nome>]
```

È necessario specificare l'incremento se diverso da 1.

Ciclo `do...while` - `DO WHILE...END DO`

- C: ciclo `do while` ripetuto sino a che è valida una espressione logica

```
do
{
  <istruzioni>
  ...
}while(<espressione-logica>)
```

- F90: il costrutto è analogo

```
[<nome>:]DO WHILE(<espressione-logica>)
  <istruzioni>
  ...
END DO [<nome>]
```

Esempi

```
DO i=0,10,2
  x=i*x
END DO cicloDO

cicloOUT: DO iOUT=24,k*j,-1
  cicloIN:  DO iIN=k,k*j,j/k
              x=x+1
            END DO cicloIN
          END DO cicloOUT
```


Esempi

```
DO WHILE (a.GE.b)
  a=a/2.
END DO

cicloOUT: DO WHILE((x.LT.100).AND.(j.GT.5))
  cicloIN:  DO i=0,10
              x=x+i
            END DO cicloIN
            j=j+1
          END DO cicloOUT
```

Costrutto `if` - `IF...END IF`

- C: esecuzione delle istruzioni subordinata alla verifica di una espressione logica

```
if(<espressione-logica>)
{
  <istruzioni>
}
[else if(<espressione-logica>)
{
  <istruzioni>
}
...]
[else
{
  <istruzioni>
}]
```


- F90: il costrutto è analogo

```
[<nome>:]IF(<espressione-logica>) THEN
  <istruzioni>
[ELSEIF(<espressione-logica>) THEN [<nome>]
  <istruzioni>
...]
[ELSE [<nome>]
  <istruzioni>]
ENDIF[<nome>]
```

Costrutto `switch` - `SELECT CASE...END SELECT`

- C: esecuzione delle istruzioni subordinata al valore di una variabile intera o carattere

```
switch(<variabile>)
{
  case <valore1>:
    <istruzioni>
    break;
  [case <valore2>:
    <istruzioni>
    break;
    ...]
  default:
    <istruzioni>
}
```


- F90: il costrutto è analogo

```
[<nome>:]SELECT CASE (<variabile>)
  CASE (<valore1A>[,<valore1B>,...]) [<nome>]
  <istruzioni>
  [CASE (<valore2A>[,<valore2B>,...]) [<nome>]
  <istruzioni>
  ...]
  CASE DEFAULT [<nome>]
  <istruzioni>
END SELECT [<nome>]
```

Esempi

```
outa: IF(a.NE.0) THEN
        b=b/a
        IF(b.NE.0) THEN
          c=c/b
        ELSE
          c=0
        ENDIF
      ELSEIF(b.GT.0) THEN outa
        b=-b
      ELSE outa
        b=b*c
      ENDIF outa
```


Esempi

```
SELECT CASE (num)
  CASE(3,6)
    ! equivalente a IF(num==3.OR.num==6)
    x=x/num
  CASE(4,6:9)
    ! equivalente a ELSEIF(num==4.OR.(num>5.AND.num<10))
    x=x+num
  CASE DEFAULT
    ! equivalente a ELSE
    x=x*num
END SELECT
  ! equivalente a ENDIF
```

Istruzioni di I/O

- Nel F90 non vi è distinzione tra indirizzo di memoria di una variabile ed il suo valore. Nel momento in cui si desidera acquisire un valore in input ed assegnarlo ad una variabile non è necessario anteporre il simbolo “&” al nome della variabile.
- Come il C, anche il F90 consente di specificare una stringa per la formattazione dell’I/O, ma questa non risulta obbligatoria (in tal caso la stampa o la lettura delle variabili viene eseguita con il formato di default).
- La formattazione dell’I/O non verrà affrontata nel seguito. Eventuali stringhe di testo possono essere inserite in fase di stampa: lo spazio viene utilizzato come unico separatore mentre per andare a capo è sufficiente inserire una nuova istruzione di stampa per ogni linea.
- Eventuali stringhe devono essere racchiuse da virgolette (”) o apici (’).

Istruzioni `scanf()` - `READ*,READ(*,*)`

- C: lettura formattata dallo standard input

```
scanf(<formato>,&<variabile1>[,&<variabile2>,...]);
```

- F90: è possibile leggere dati in formato libero dallo standard input con le istruzioni

```
READ*, <variabile1>[,<variabile2>,...]
READ(*,*) <variabile1>[,<variabile2>,...]
```

Nota: Si ricorda che il carattere “&” permette di andare a capo con una istruzione F90 (si consiglia il ricorso frequente ad esso per migliorare la leggibilità del codice). Le variabili vengono lette in base al tipo con cui sono state dichiarate.

Istruzioni `printf()` - `PRINT*,WRITE(*,*)`

- C: stampa formattata sullo standard output

```
printf(<formato>,<variabile1>[,<variabile2>,...]);
```

- F90: con i comandi `PRINT*` e `WRITE(*,*)` è possibile stampare stringhe e variabili sullo standard output

```
PRINT*, ["<stringa1>",]<variabile1>[&
       &[,"<stringa2>"],<variabile2>,...]
WRITE(*,*) [<stringa1>",]<variabile1>[&
       &[,"<stringa2>"],<variabile2>,...]
```

Nota: Volendo ottenere una migliore precisione nella stampa di una variabile, occorre specificare un formato. In ogni caso, la parte decimale di numeri reali viene arrotondata e non troncata.

Apertura di file per l'IO

- C: è necessario definire un puntatore di tipo `FILE` ed aprire l'unità con `fopen(<file-pointer>,<nome-file>,<modalità>)`; si ricorrere poi ai comandi

```
fscanf(<file-pointer>,<formato>,&<variabile>)
fprintf(<file-pointer>,<formato>,<variabile>)
```

- F90: in modo analogo, tramite il comando `OPEN` si definisce una unità logica (rappresentata da un numero intero) specificando il nome del file a cui si farà riferimento e la modalità di accesso. La lettura e la stampa avvengono con i comandi `READ,WRITE` specificando l'unità di destinazione

```
READ([UNIT=]<unita-logica>,*) <variabile>
WRITE([UNIT=]<unita-logica>,*) <variabile>
```

Nota: È buona norma chiudere le unità logiche non più in uso.

Istruzioni OPEN, CLOSE

```
OPEN([UNIT=]<unita-logica>,FILE=<nome-file>,&
    &ERR=<error-label>,STATUS=<status>,ACTION=<mode>)
CLOSE(<unita-logica>)
```

- <unita-logica>= numero intero di riferimento per l'unità logica;
- <nome-file>= stringa per il nome del file da aprire;
- <error-label>= numero di riferimento della linea a cui il flusso di istruzioni deve essere trasferito in caso di errore nell'apertura;
- <status>= si consiglia di utilizzare 'UNKNOW': il file verrà sovrascritto se già esistente o creato se assente (altre possibilità sono 'OLD','NEW','REPLACE','SCRATCH')
- <mode>= 'READ' per file in sola lettura, 'WRITE' per file in sola scrittura o 'READWRITE' per file in lettura e scrittura.

Esempi e Note

```
READ*, x,y
PRINT*, "x= ",x,"   f(x)= ",y

OPEN(14,FILE='inp.dat',ERR=10,STATUS='UNKNOWN',&
     &ACTION='READ')
OPEN(17,FILE='out.dat',ERR=10,STATUS='UNKNOWN',&
     &ACTION='WRITE')
READ(14,*) x1,x2,x3
WRITE(17,*) "f(x1)= ",y1," f(x2)= ",y2," f(x3)= ",y3
CLOSE(14)
CLOSE(17)
```

Nota: Non può essere aperto più di un file su di una stessa unità logica.

Funzioni e subroutine

- Come il C, il F90 prevede l'utilizzo di funzioni definite dall'utente.
- Le procedure esterne sono distinguibili in
  - SUBROUTINE: sequenza di codice dipendente da parametri e caratterizzata da un nome che può essere richiamata (tramite istruzione "`CALL <nome-subroutine>([<argomenti>])`"); alla subroutine possono essere passati valori ed al suo interno possono risiedere istruzioni che modificano tali valori;
  - FUNCTION: come una subroutine, ma restituisce un solo risultato in corrispondenza del nome di funzione richiamato (non è necessaria alcuna altra istruzione); il valore restituito dipende, in genere, dai parametri passati alla funzione.

Definizione di subroutine e funzioni

- La struttura di una SUBROUTINE può essere schematizzata come segue

```
SUBROUTINE <nome-subroutine>[(<argomenti>)]
  <dichiarazione degli argomenti>
  <dichiarazione di variabili locali>
  ...
  <istruzioni>
END [SUBROUTINE[<nome-subroutine>]]
```

- Per quanto riguarda le FUNCTION si ha qualcosa di analogo

```
<tipo-risultato> FUNCTION <nome-funzione>(<argomenti>)
  <dichiarazione degli argomenti>
  <dichiarazione di variabili locali>
  ...
  <istruzioni>
END [FUNCTION [<nome-funzione>]]
```

Inclusione di subroutine e funzioni

- A differenza dell'ANSI C, il F90 non richiede la definizione di prototipi di procedure, se queste sono interne, cioè definite all'interno del programma principale.
- Per includere procedure interne è sufficiente inserire nel programma principale una sezione delimitata dall'istruzione "`CONTAINS`" all'interno della quale occorre definire tutte le procedure utilizzate dal programma.
- In F90 è possibile definire anche moduli ("`MODULE`") che possono contenere dichiarazioni di variabili ed una sezione `CONTAINS` per le procedure; richiamando un modulo (istruzione "`USE <nome-modulo>`") si rendono visibili al programma principale le variabili e le procedure in esso contenute.

Argomenti visibili ad una procedura

- In generale, quando il controllo del flusso di istruzioni passa ad una procedura, essa non può gestire le variabili in uso nel programma principale se queste non vengono rese "visibili" anche alla procedura.
- In C, ciò è possibile passando esplicitamente le variabili come argomenti di una funzione.
- Anche in F90 una variabile risulta "visibile" alla procedura se viene passata come argomento oppure se viene richiamata all'interno della procedura sfruttando un modulo in cui essa viene dichiarata.
- I nomi simbolici dati alle variabili all'interno del programma e della procedura non devono obbligatoriamente coincidere: ciò consente la massima riusabilità di una procedura.

Scopo di un argomento: INTENT

- In C, una funzione può restituire un solo valore; utilizzando però gli indirizzi di memoria delle variabili, è possibile modificare anche i valori di altre variabili del programma principale.
- In F90, è necessario specificare lo scopo ("INTENT") di ciascuno degli argomenti passati esplicitamente alla procedura.
- L'INTENT di un argomento, viene specificato come opzione nella dichiarazione della variabile e può assumere i valori:
  - INTENT(IN): il valore dell'argomento è accessbile in lettura;
  - INTENT(OUT): l'argomento non è utilizzato sino a che non gli viene assegnato un valore che verrà restiruito al programma princiaple al termine della procedura;
  - INTENT(INOUT): il valore dell'argomento è accessbile in lettura e può essere modificato;

Esempi

```
PROGRAM Main
  IMPLICIT NONE
  REAL :: x1,y1,x2,y2
  READ*, x1,y1,x2,y2
  CALL Simmetric(x1,y1,x2,y2)
  PRINT*, "Distanza=",Distance(x1,y1,x2,y2)
  CONTAINS
    REAL FUNCTION Distance(a,b,c,d)
      REAL, INTENT(IN)::a,b,c,d
      Distance=SQRT((a-c)**2+(b-d)**2)
    END FUNCTION Distance
    SUBROUTINE Simmetric(a,b,c,d)
      REAL, INTENT(IN)::a,b
      REAL, INTENT(INOUT)::c,d
      IF((a*c<0).AND.(b*d<0)) THEN
        c=-c; d=-d
      ENDIF
    END SUBROUTINE Simmetric
  END PROGRAM Main
```

Variabili globali e Moduli

- Sfruttando il comando `USE <nome-modulo>` all'interno di una procedura, è possibile rendere visibili ad essa tutte le variabili dichiarate nel modulo (incluso a sua volta anche ne programma principale).
- Una variabile visibile tramite inclusione di un modulo risulta leggibile e modificabile dalla procedura.
- Tali variabili non devono essere ridefinite all'interno della procedura; pertanto, per esse, non viene specificato alcun `INTENT`. Esse devono necessariamente mantenere lo stesso nome che avevano nel programma principale in quanto rese comuni alla procedura tramite l'inclusione del medesimo modulo.
- Ciò consente di evitare il passaggio di un gran numero di argomenti alla procedura nel caso essa sia assai complessa, ma ne rendono l'utilizzo strettamente vincolato al programma in uso limitandone la riusabilità.

Struttura di un Modulo

```
MODULE <nome-modulo>
  <inclusoine di altri moduli>
  [IMPLICIT NONE]
  <dichiarazione di variabili>
  CONTAINS
    <definizione di procedure>
END [MODULE [<nome-modulo>]]
```

Esempi

```
MODULE Dati
  IMPLICIT NONE
  REAL :: x1,y1
  CONTAINS
    REAL FUNCTION Distance(a,b)
      REAL, INTENT(IN)::a,b
      Distance=SQRT((x1-a)**2-(y1-b)**2)
    END FUNCTION Distance
END MODULE Dati

PROGRAM Main
  USE Dati
  IMPLICIT NONE
  REAL :: x2,y2
  READ*, x1,y1,x2,y2
  PRINT*, "Distanza=",Distance(x2,y2)
END PROGRAM Main
```