CINECA

Introduzione alle Procedure

Introduction to Fortran for Scientific Computing

Angela Acocella, Tommaso Gorni, Moreno Guernelli, Lorenzo Varrassi, Caterina Caravita

a.acocella@cineca.it

November 12th 2025

COSA VIENE PRESENTATO

- Le procedure di un programma Fortran (*Subroutines and Functions*)
- I moduli nel Fortran90 (*Modules*)
- Interfaccia esplicita (*Host/Use association*)
- Interfaccia implicita (Blocchi di interfaccia)
- *«Scope» e «scoping units»*

Procedure

Un programma Fortran può essere suddiviso in singole sottounità che corrispondono a parti logiche distinte, ognuna delle quali contiene una serie di semplici istruzioni: chiamiamo queste unità procedure (o sottoprogrammi)
Una procedura ben programmata permette di:

- testare indipendentemente le single sottounità di codice
- rendere riutilizzabile parte del codice
- isolare variabili per evitare di propagare errori nelle fasi di *debugging* e/o *refactoring* di un codice

Una procedura può essere:

- esterna, codificabile come unità di programma separata, che può essere compilata, testata e sottoposta a *debugging* in modo indipendente da tutte le altre procedure del programma

- interna: inserita all'interno di un'altra unità di programma. Se un'unità di programma contiene una procedura, viene definita *host unit*. Le procedure interne non possono contenere ulteriori procedure

Sottoprogrammi e Moduli

Il Fortran prevede due tipi di sottoprogrammi o procedure (**SUBROUTINES** e **FUNCTIONS**) e una tipologia di unità di programma separata (**MODULES**):

SUBROUTINES: sono procedure che vengono richiamate nominandole attraverso una istruzione **CALL** e possono restituire più valori tramite una lista di argomenti

FUNCTIONS: sono procedure che vengono richiamate **nominandole** utilizzando un'espressione, e restituiscono un solo valore in funzione di uno o più parametri. La chiamata avviene tramite un'istruzione di assegnazione

MODULES: sono unità di programma separate che contengono definizioni di variabili, definizioni di *derived-data types*, sottoprogrammi e blocchi di interfaccia, da condividere con altre unità di programma; i moduli vengono richiamati mediante l'istruzione **USE** inserita all'interno delle altre unità

Sottoprogrammi e Moduli

Un codice Fortran deve **SEMPRE** possedere UN SOLO PROGRAMMA PRINCIPALE, chiamato ***main program***. Si possono invece definire un numero illimitato di procedure e unità di programmazione separate (subroutines/functions/modules)

Le procedure (subroutines/functions)vengono inserite ed introdotte in maniera diversa a seconda che siano interne o esterne:

- procedura interna (contenuta all'interno dell'unità di programmazione) è introdotta dalla keyword `CONTAINS`, utilizzata per indicare l'inizio della procedura: `CONTAINS` separa qualunque procedura interna dalla *host unit*. **Una procedura interna ha accesso a tutte le entità dichiarate nella *host unit***, (inclusa la possibilità di chiamare i suoi ulteriori sottoprogrammi interni)

- procedura esterna: (presente all'esterno del programma principale) viene inserita dopo la keyword `END PROGRAM`, o eventualmente in un file a parte (librerie, es: `LAPACK/BLAS`)

I moduli (modules)sono unità di programmazione separata dal *main program* e da tutti i sottoprogrammi

Organizzazione del programma

Schema delle diverse unità e sottounità di programma Fortran

```
PROGRAM, FUNCTION, SUBROUTINE, MODULE nome
[USE (module name)]
IMPLICIT NONE
Dichiarazioni
Istruzioni eseguibili
[CONTAINS]
   [procedure (SUBROUTINE, FUNCTION)]
END (PROGRAM, FUNCTION, SUBROUTINE, MODULE nome)
```

Good Programming Practice: suddividere un programma di grandi dimensioni in sottounità indipendenti per agevolare *debugging*, *testing* e *refactoring*

Good Programming Practice: contrassegnare sempre l'unità che inizia e finisce con il suo nome

Subroutine

Sintassi per la SUBROUTINE:

```
SUBROUTINE nome (lista-argomenti-fittizi)
  (dichiarazioni variabili)
  (istruzioni)
  [RETURN]
END SUBROUTINE nome
```

Chiamata della SUBROUTINE:

```
PROGRAM principale
  ...
  CALL nome(lista-argomenti-effettivi)
  ...
END PROGRAM principale
```

Subroutine

Sintassi per la SUBROUTINE:

```
SUBROUTINE nome (lista-argomenti-fittizi)
  (dichiarazioni variabili)
  (istruzioni)
  [RETURN]
END SUBROUTINE nome
```

Chiamata della SUBROUTINE:

```
PROGRAM principale
  ...
  CALL nome(lista-argomenti-effettivi)
  ...
END PROGRAM principale
```

Esempio:

```
SUBROUTINE somma(a,b,c)
REAL :: a,b,c
c = a + b

END SUBROUTINE somma
```

```
PROGRAM principale
   ...
   CALL somma(a,b,c)
   ...
END PROGRAM principale
```

Subroutine

- Una *subroutine* è definita dal suo nome, univoco: il nome segue le convenzioni Fortran standard (può contenere caratteri alfanumerici e l'underscore, ma il primo carattere deve essere sempre alfabetico). Il nome è facoltativo nell'istruzione **`END SUBROUTINE`**

- Le variabili da utilizzare all'interno della *subroutine* sono specificate utilizzando una lista di argomenti, racchiusa tra parentesi, che segue il nome della *subroutine*. L'elenco degli argomenti contiene la lista delle variabili semplici e/o degli array che vengono scambiati tra la subroutine e il programma principale. Queste variabili sono in realtà argomenti fittizi (*dummy arguments*), poiché la *subroutine* non alloca memoria per esse

- L'ordine e il tipo degli argomenti fittizi deve corrispondere all'ordine e al tipo di argomenti effettivi indicati nell'istruzione **`CALL`** del programma principale - **`CALL`** nome (lista-argomenti-effettivi / *actual arguments*) – *argument association by-reference*

- Si possono usare nomi diversi per gli argomenti fittizi ed effettivi, ma quando possibile è preferibile usare lo stesso nome in modo da rendere più leggibile il codice

Subroutine

- Le variabili che vengono dichiarate ed utilizzate **SOLO** all'interno della *subroutine* e che non sono accessibili attraverso la procedura `CALL` sono definite e trattate come **variabili locali**

- Uscendo dalla subroutine, il valore delle variabili locali viene perso. Se si prevede di usare ancora tali valori alla successiva invocazione della *subroutine*, si possono preservare con l'attributo `SAVE`, che va aggiunto in fase di dichiarazione

- Può essere utile stabilire più punti di uscita dalla *subroutine*, per esempio in funzione di un risultato ottenuto al suo interno. Per uscire dalla *subroutine* e tornare al programma chiamante si usa l'istruzione `RETURN`

- Per testare una subroutine è necessario scrivere un *test program*, ovvero, un semplice programma che richiama la *subroutine* con un insieme di dati sufficiente a testare la funzionalità del sottoprogramma

L'attributo INTENT

In Fortran gli argomenti tra le unità di programma vengono passati *by-reference*, (*argument association by-reference*) ovvero, mediante il loro indirizzo di memoria per risparmiare spazio di memoria

Gli argomenti fittizi (lista degli argomenti del sottoprogramma) e gli argomenti effettivi (lista degli argomenti dell'unità chiamante) condividono le stesse posizioni di memoria: quello che viene passato alla subroutine sono i puntatori alle posizioni di memoria che contengono gli argomento effettivi

Questa è un'ottima funzionalità, ma può anche essere fonte di errori e indurre *side-effects,* i.e., effetti collaterali, indesiderati. Poiché gli argomenti fittizi ed effettivi puntano alla stessa locazione di memoria, la modifica di un argomento fittizio modifica anche l'argomento effettivo:

- modifiche involontarie, se la subroutine cambia un argomento senza volerlo, la variabile originale nella chiamante viene modificata
- alias e conflitti, se lo stesso argomento effettivo è passato a più argomenti fittizi, una modifica può influenzare gli altri
- difficile debug, effetti collaterali nascosti rendono il codice più difficile da leggere, testare e mantenere

L'attributo INTENT

Per questa ragione, Fortran permette di assegnare una corretta funzionalità agli argomenti fittizi utilizzando l'attributo `INTENT`, che istruisce il compilatore sull'effettivo utilizzo di una determinata variabile

L'attributo `INTENT` può essere:

`INTENT(IN)`	L'argomento fittizio è usato solo per passare dati di input alla subroutine
`INTENT(OUT)`	L'argomento fittizio è usato solo per restituire un risultato al programma chiamante
`INTENT(INOUT)`	L'argomento fittizio è usato sia per passare dati di input che di output al programma chiamante

L'attributo INTENT

Per questa ragione, Fortran permette di assegnare una corretta funzionalità agli argomenti fittizi utilizzando l'attributo `INTENT`, che istruisce il compilatore sull'effettivo utilizzo di una determinata variabile

Se viene accidentalmente modificata una variabile di input nella subroutine, il compilatore restituisce **un errore**:

Example in S.J. Chapman, Chap. 7

```
sub1.f90(7): error #6780: A dummy argument with the INTENT(IN) attribute
shall not be defined nor become undefined.  [INPUT]
input = -1.
^
compilation aborted for sub1.f90 (code 1)
```

L'attributo INTENT

Per questa ragione, Fortran permette di assegnare una corretta funzionalità agli argomenti fittizi utilizzando l'attributo `INTENT`, che istruisce il compilatore sull'effettivo utilizzo di una determinata variabile

L'attributo `INTENT` può essere:

`INTENT(IN)`

`INTENT(OUT)`

`INTENT(INOUT)`

Esempio:

```
SUBROUTINE Somma(a,b,c)
    IMPLICIT NONE
    REAL, INTENT(OUT) :: c
    REAL, INTENT(IN) :: a, b
    c = a + b
    RETURN
END SUBROUTINE Somma
```

L'attributo INTENT

Per questa ragione, Fortran permette di assegnare una corretta funzionalità agli argomenti fittizi utilizzando l'attributo `INTENT`, che istruisce il compilatore sull'effettivo utilizzo di una determinata variabile

L'attributo `INTENT` può essere:

`INTENT(IN)`

`INTENT(OUT)`

`INTENT(INOUT)`

Esempio:

```
SUBROUTINE Sommatoria(a,b,c)
   IMPLICIT NONE
   REAL, INTENT(INOUT) :: c
   REAL, INTENT(IN) :: a, b
   c = c + a*b
   RETURN
END SUBROUTINE Sommatoria
```

Function

Sintassi per la FUNCTION:

```
tipo FUNCTION nome ([lista-argomenti-fittizi])
  (dichiarazioni variabili)
  (istruzioni)
  nome = valore di ritorno
END FUNCTION nome
```

Chiamata della FUNCTION:

```
PROGRAM principale
  tipo :: nome
  tipo :: variabile
  ...
  variabile = nome([lista-argomenti-effettivi])
  ...
END PROGRAM principale
```

L'elenco degli argomenti della funzione può essere vuoto se la funzione esegue istruzioni di calcolo senza necessità di ricevere argomenti di input

Le parentesi tonde sono richieste anche se la lista è vuota per chiarezza, e sono obbligatorie in chiamata della *function*

Function

Sintassi per la FUNCTION:

```
tipo FUNCTION nome ([lista-argomenti-fittizi])
  (dichiarazioni variabili)
  (istruzioni)
  nome = valore di ritorno
END FUNCTION nome
```

Chiamata della FUNCTION:

```
PROGRAM principale
  tipo :: nome
  tipo :: variabile
  ...
  variabile = nome([lista-argomenti-effettivi])
  ...
END PROGRAM principale
```

Esempio:

```
REAL FUNCTION somma(a,b)
REAL :: a, b

  somma = a + b
END FUNCTION somma



PROGRAM principale
  REAL :: somma
  REAL :: r
  ...
  r = somma(a,b)
  ...
END PROGRAM principale
```

Function

Sintassi per la FUNCTION:

```
tipo FUNCTION nome ([lista-argomenti-fittizi])
  (dichiarazioni variabili)
  (istruzioni)
  nome = valore di ritorno
END FUNCTION nome
```

Chiamata della FUNCTION:

```
PROGRAM principale
  tipo :: nome
  tipo :: variabile
  ...
  variabile = nome([lista-argomenti-effettivi])
  ...
END PROGRAM principale
```

Esempio:

```
REAL FUNCTION somma(a,b)
REAL, INTENT(IN) :: a, b

  somma = a + b
END FUNCTION somma



PROGRAM principale
  REAL :: somma
  REAL :: r
  ...
  r = somma(a,b)
  ...
END PROGRAM principale
```

Function

- Una *function* è una procedura Fortran il cui risultato può essere un numero, un valore logico, un array, una stringa. Il nome segue le convenzioni Fortran standard (può contenere caratteri alfanumerici e l'underscore, ma il primo carattere deve essere sempre alfabetico). Il nome è facoltativo nell'istruzione **`END FUNCTION`**. Le parentesi tonde dopo il nome sono obbligatorie, la lista di argomenti è facoltativa

- Una *function* è definita in maniera simile ad una *subroutine*; al contrario di una subroutine si deve specificare il tipo di variabile che deve essere restituita dalla funzione (`tipo`)

  → Il `tipo` di una function può essere definito prima della keyword **`function`**, oppure, ALTERNATIVAMENTE, all'interno del corpo del sottoprogramma

```
tipo function nome([lista_argomenti])


oppure


function nome([lista_argomenti])
tipo :: nome
```

Function

- Una *function* è una procedura Fortran il cui risultato può essere un numero, un valore logico, un array, una stringa. Il nome segue le convenzioni Fortran standard (può contenere caratteri alfanumerici e l'underscore, ma il primo carattere deve essere sempre alfabetico). Il nome è facoltativo nell'istruzione **`END FUNCTION`**. Le parentesi tonde dopo il nome sono obbligatorie, la lista di argomenti è facoltativa

- Una *function* è definita in maniera simile ad una *subroutine*; al contrario di una subroutine si deve specificare il tipo di variabile che deve essere restituita dalla funzione (`tipo`)

Il valore di ritorno della *function* deve essere definito prima dell'uscita dalla funzione

```
nome = valore di ritorno
```

```
tipo function nome([lista_argomenti])


oppure


function nome([lista_argomenti])
tipo :: nome
```

Function

Regole nella definizione del tipo di una *function*:

1. Se la **FUNCTION** è interna, il `tipo` si definisce solo nella sua intestazione, prima della keyword **FUNCTION**: l'unità chiamante conosce il tipo di valore restituito dalla funzione
2. Se la **FUNCTION** è esterna e **NON** è usato `IMPLICIT NONE` nell'unità chiamante, il `tipo` può essere dedotto implicitamente (usando le regole **A-H o O-Z** → REAL; **I-N** → INTEGER), oppure specificato prima di **FUNCTION**
3. Se la **FUNCTION** è esterna e nell'unità chiamante è presente `IMPLICIT NONE`, il tipo deve essere dichiarato **SIA** nella definizione della funzione **SIA** nell'unità chiamante

1
```
PROGRAM main
  IMPLICIT NONE
  PRINT*, f(2.0)
CONTAINS
  REAL FUNCTION f(x)
    REAL, INTENT(IN) :: x
    f = 2*x
  END FUNCTION f
END PROGRAM main
```

2
```
PROGRAM main
  PRINT*, f(2.0)
END PROGRAM main

REAL FUNCTION f(x)
  REAL, INTENT(IN) :: x
  f = 2*x
END FUNCTION f
```

3
```
PROGRAM main
  IMPLICIT NONE
  REAL :: f
  PRINT*, f(2.0)
END PROGRAM main

REAL FUNCTION f(x)
  IMPLICIT NONE
  REAL, INTENT(IN) :: x
  f = 2*x
END FUNCTION f
```

Procedure interne

Le **procedure interne** sono definite dopo l'istruzione **CONTAINS**

- Sono parte di una unità di programma e accedono alle variabili dell'unità di cui fanno parte

- La procedura interna ha una INTERFACCIA ESPLICITA, ovvero, la procedura eredita automaticamente le definizioni degli entità dichiarate nell'unità chiamante, a meno che l'unità interna ridefinisca esplicitamente tali entità

- Questa eredità è chiamata *host association*

```
PROGRAM test_internal
IMPLICIT NONE
REAL, PARAMETER :: PI = 3.141592
! PI è dichiarata solo nella host unit
REAL :: theta
! Posso non dichiarare la function nel main program
perché è un sottoprogramma interno

WRITE (*,*)'Enter angle in degrees:; READ (*,*) theta
WRITE (*,'(A,F10.4)') 'The secant is ', secant(theta)

CONTAINS

REAL FUNCTION secant(angle_in_degrees)
REAL :: angle_in_degrees
secant = 1. / cos( angle_in_degrees * PI / 180. )
END FUNCTION secant

END PROGRAM test_internal
```

Procedure esterne

Le **procedure esterne** sono definite all'esterno del programma, dei moduli e di ogni altra procedura

- Sono dotate di un proprio spazio di memoria esclusivo
- Si trovano in un file separato o nello stesso file del programma e dei moduli dopo l'istruzione di chiusura (**END …**)
- **NON** sono precedute dall'istruzione **CONTAINS**
- Hanno un' INTEFACCIA IMPLICITA

```
PROGRAM Esempio
   . . .
   CALL Esterna(...)
   . . .

END PROGRAM Esempio


SUBROUTINE Esterna(...)
   . . .
  RETURN
END SUBROUTINE Esterna
```

Procedure esterne

Le **procedure esterne** sono definite all'esterno del programma, dei moduli e di ogni altra procedura

- Sono dotate di un proprio spazio di memoria esclusivo
- Si trovano in un file separato o nello stesso file del programma e dei moduli dopo l'istruzione di chiusura (**END …**)
- **NON** sono precedute dall'istruzione **CONTAINS**
- Hanno un' INTEFACCIA IMPLICITA

```
PROGRAM test_internal
IMPLICIT NONE
REAL, PARAMETER :: PI = 3.141592
! PI è dichiarata nella host unit
REAL :: theta
REAL :: secant
! Devo dichiarare la function nel main program perché la
function è esterna + IMPLICIT NONE

WRITE (*,*)'Enter angle in degrees:; READ (*,*) theta
WRITE (*,'(A,F10.4)') 'The secant is ', secant(theta)

END PROGRAM test_internal

REAL FUNCTION secant(angle_in_degrees)
REAL :: angle_in_degrees
REAL, PARAMETER :: PI = 3.141592
! PI è dichiarata ANCHE nella function

secant = 1. / cos( angle_in_degrees * PI / 180. )
END FUNCTION secant
```

Procedure esterne

Le **procedure esterne** sono definite all'esterno del programma, dei moduli e di ogni altra procedura

- Sono dotate di un proprio spazio di memoria esclusivo
- Si trovano in un file separato o nello stesso file del programma e dei moduli dopo l'istruzione di chiusura (END …)
- **NON** sono precedute dall'istruzione CONTAINS
- Hanno un' INTEFACCIA IMPLICITA

```
PROGRAM test_internal

REAL, PARAMETER :: PI = 3.141592 ! dichiarata nella host unit
REAL :: theta
! Posso non dichiara la function nel main program anche se è esterna perché NON C'E' IMPLICIT NONE

WRITE (*,*)'Enter angle in degrees:; READ (*,*) theta
WRITE (*,'(A,F10.4)') 'The secant is ', secant(theta)

END PROGRAM test_internal

REAL FUNCTION secant(angle_in_degrees)
REAL :: angle_in_degrees
REAL, PARAMETER :: PI = 3.141592
! PI è dichiarata ANCHE nella function

secant = 1. / cos( angle_in_degrees * PI / 180. )
END FUNCTION secant
```

Come compilare con sottoprogrammi esterni?

Esempio: abbiamo un main program (**mainProg.f90**) e una subroutine esterna **sub1.f90**

sub1.f90

```
subroutine sub1(x)

implicit none
    real, intent(inout) :: x

    x = x**2

end subroutine sub1
```

mainProg.f90

```
program mainProg
    implicit none
    real :: y
    y = 3.0
    call sub1(y)
    print *, y
end program mainProg
```

Come compilare con sottoprogrammi esterni?

Compilazione separata: approccio per ridurre i tempi di compilazione in progetti grandi

1. si compilano il main program e la subroutine (o function), producendo i rispettivi files oggetto (***.o**)

```
$ gfortran -c sub1.f90 mainProg.f90
$ ls
$ mainProg.f90  mainProg.o sub1.f90  sub1.o
```

2. eseguire il linking per creare un unico eseguibile (**a.out**):

```
$ gfortran sub1.o mainProg.o
$ ls
$ a.out mainProg.f90  mainProg.o sub1.f90  sub1.o
```

Come compilare con sottoprogrammi esterni?

Compilazione separata: approccio per ridurre i tempi di compilazione in progetti grandi

1. si compilano il main program e la subroutine (o function), producendo i rispettivi files oggetto (***.o**)

```
$ gfortran -c sub1.f90 mainProg.f90
$ ls
$ mainProg.f90  mainProg.o sub1.f90  sub1.o
```

2. eseguire il linking per creare un unico eseguibile (**a.out**):

```
$ gfortran sub1.o mainProg.o
$ ls
$ a.out mainProg.f90  mainProg.o sub1.f90  sub1.o
```

Compilazione in un unico comando:

```
$ gfortran sub1.f90 mainProg.f90 –o executable.exe
$ ls
$ executable.exe mainProg.f90  sub1.f90
```

Esercizi - Introduzione alle procedure

Si scriva una subroutine che calcoli l'ipotenusa di un triangolo rettangolo

Si scrivano una serie di subroutine riutilizzabili in grado di determinare le proprietà statistiche di un insieme di dati di numeri reali in un array. L'insieme delle subroutine dovrebbe includere:

a) una subroutine per determinare il valore massimo in un set di dati e la sua posizione
b) una subroutine per determinare il valore minimo in un set di dati e la sua posizione

Moduli

I programmi, le *subroutines* e le *functions* Fortran possono scambiarsi dati attraverso i MODULI

- I moduli sono un'unità di programma compilata separatamente dal programma principale che contiene le definizioni e i valori iniziali dei dati da condividere: i.e., variabili semplici, arrays, tipi derivati e loro operazioni, sottoprogrammi e blocchi di interfaccia. Ogni unità di programma che **USA** un modulo ha interamente accesso a tutti i dati dichiarati nel modulo

Moduli

I programmi, le *subroutines* e le *functions* Fortran possono scambiarsi dati attraverso i MODULI

- I moduli sono un'unità di programma compilata separatamente dal programma principale che contiene le definizioni e i valori iniziali dei dati da condividere: i.e., variabili semplici, arrays, tipi derivati e loro operazioni, sottoprogrammi e blocchi di interfaccia. Ogni unità di programma che **USA** un modulo ha interamente accesso a tutti i dati dichiarati nel modulo

I moduli sono uno strumento per:

- definire i dati e loro strutture;
- definire tutte le operazioni operanti sui suddetti dati e strutture;
- definire altre procedure (sottoprogrammi) utilizzate dal programma (*module procedures*)

Moduli

I programmi, le *subroutines* e le *functions* Fortran possono scambiarsi dati attraverso i MODULI

- I moduli sono un'unità di programma compilata separatamente dal programma principale che contiene le definizioni e i valori iniziali dei dati da condividere: i.e., variabili semplici, arrays, tipi derivati e loro operazioni, sottoprogrammi e blocchi di interfaccia. Ogni unità di programma che **USA** un modulo ha interamente accesso a tutti i dati dichiarati nel modulo

I moduli sono uno strumento per:

  - definire i dati e loro strutture;
  - definire tutte le operazioni operanti sui suddetti dati e strutture;
  - definire altre procedure (sottoprogrammi) utilizzate dal programma (*module procedures*)

- L'utilizzo del modulo permette di rendere dati e procedure accessibili al resto del programma, ma consente anche di proteggere i dati e le parti di codice che non necessitano di manipolazioni esterne (*data hiding*). Le librerie Fortran spesso consistono di un insieme di moduli

Moduli

Sintassi per il MODULO:

```
MODULE nome
IMPLICIT NONE
[SAVE]
(dichiarazione dati da condividere)
END MODULE nome
```

Chiamata al MODULO:

```
PROGRAM nome
USE nome (del MODULO)
IMPLICIT NONE
(dichiarazioni variabili locali)
(istruzioni)
END PROGRAM nome
```

Moduli

Sintassi per il MODULO:

```
MODULE nome
IMPLICIT NONE
[SAVE]
(dichiarazione dati da condividere)
END MODULE nome
```

Chiamata al MODULO:

```
PROGRAM nome
USE nome (del MODULO)
IMPLICIT NONE
(dichiarazioni variabili locali)
(istruzioni)
END PROGRAM nome
```

Esempio:

```
MODULE Globale
IMPLICIT NONE
SAVE
REAL :: r, s
END MODULE Globale
```

```
PROGRAM Principale
USE Globale
IMPLICIT NONE
REAL :: c
c = r + s
write (*,*) "Risultato: ", c
END PROGRAM Principale
```

Moduli

Il nome di un modulo segue le convenzioni Fortran standard (può contenere caratteri alfanumerici e l'underscore, ma il primo carattere deve essere sempre alfabetico). Il nome è facoltativo nell'istruzione **`END MODULE`**

L'espressione `SAVE` prima della dichiarazione delle variabili garantisce che i dati dichiarati nel modulo saranno preservati tra le diverse procedure: dovrebbe SEMPRE essere inserito nei moduli che dichiarano dati da condividere con altre unità di programma

Il programma principale e qualunque procedura o altro modulo possono accedere a ciò che è definito in un modulo utilizzando la sintassi:

```
USE nome (del MODULO)
```
(***USE association***)

da usare prima di ogni altra istruzione, compreso IMPLICIT NONE

Un modulo può: i) essere usato da PIU' sottoprogrammi contemporaneamente, ii) contenere ALTRI SOTTOPROGRAMMI, iii) accedere ad ALTRI MODULI mediante *USE association*, ma NON può eseguire **`USE`** di se stesso

Moduli

Un tipico esempio di dati da condividere attraverso un modulo sono le definizioni di precisione dei valori interi e reali che vogliamo vengano utilizzati tra le diverse unità di programma e/o per accertarci che la precisione dei dati non cambi quando il programma viene spostato da un computer ad un altro

Un problema rilevante che si può incontrare durante il trasferimento di un programma Fortran da un computer a un altro è che le definizioni di *single* e *double precision* non siano uniformi: il numero di bit associato a ciascun tipo di variabile reale dipende dal produttore del computer

Per evitare incongruenze, si utilizzano delle funzioni intrinseche per definire la precisione selezionando univocamente il tipo corretto di valore reale da utilizzare quando il programma viene spostato tra computers

`SELECTED_INT_KIND(r):`
tipo di numero intero con r cifre significative

`SELECTED_REAL_KIND(p,r)`
tipo di numero in virgola mobile con p cifre di precisione ed esponente compreso tra ± r

Moduli

Inserendo le diverse definizioni di precisione di interi e reali all'interno di un modulo, mediante *use association* possiamo facilmente condividerli tra le unità di programma

```
MODULE numeric_kinds
- from M. Metcalf, J. Reid, and M. Cohen, "Modern Fortran Explained"

! named constants for 4-, 2-, and 1-byte integers:
integer, parameter :: &
i4b = selected_int_kind(9), &
i2b = selected_int_kind(4), &
i1b = selected_int_kind(2)

! and for single, double and quadruple precision reals:
integer, parameter :: &
sp = kind(1.0), &
dp = selected_real_kind(2*precision(1.0_sp)), &
qp = selected_real_kind(2*precision(1.0_dp))
! I suffissi _sp/_dp sono modifier del literal numerico che rappresenta
la precisione
END MODULE numeric_kinds
```

SELECTED_INT_KIND(r):
Returns a value of the kind type parameter of an integer data type that represents all integer values n with $-10^R < n < 10^R$

SELECTED_REAL_KIND(p,r) returns the kind value of a real data type with decimal precision of at least P digits, exponent range of at least R

PRECISION: returns the decimal precision

KIND(x): determines the kind imteger numbers of x associated with single- and double-precision variables **on a particular processor**

Single precision → 6–7 cifre decimali
Double precision → circa 16 cifre decimali
Quad precision → circa 34 cifre cifre decimali

Moduli

Inserendo le diverse definizioni di precisione di interi e reali all'interno di un modulo, mediante *use association* possiamo facilmente condividerli tra le unità di programma

```
MODULE numeric_kinds
- from M. Metcalf, J. Reid, and M. Cohen, "Modern Fortran Explained"

! named constants for 4-, 2-, and 1-byte integers:
integer, parameter :: &
i4b = selected_int_kind(9), &
i2b = selected_int_kind(4), &
i1b = selected_int_kind(2)

! and for single, double and quadruple precision reals:
integer, parameter :: &
sp = kind(1.0), &
dp = selected_real_kind(2*precision(1.0_sp)), &
qp = selected_real_kind(2*precision(1.0_dp))
END MODULE numeric_kinds
```

SELECTED_INT_KIND(r):
Returns a value of the kind type parameter of an integer data type that represents all integer values n with $-10^R < n < 10^R$

SELECTED_REAL_KIND(p,r) returns the kind value of a real data type with decimal precision of at least P digits, exponent range of at least R

PRECISION: returns the decimal precision

KIND(x): determines the kind imteger numbers of x associated with single- and double-precision variables **on a particular processor**

Single precision → 6–7 cifre decimali
Double precision → circa 16 cifre decimali
Quad precision → circa 34 cifre cifre decimali

Good Programming Practice: usare i moduli per passare una grande quantità di dati (o dati di grandi dimensioni) tra le procedure all'interno di un programma. Includere sempre l'espressione SAVE per assicurarsi che i dati di un modulo non vengano modificati da altre procedure

Moduli

Oltre ai dati, i moduli possono contenere complete procedure (*subroutines/functions*), che vengono chiamate procedure di modulo (*module procedures*)

Le procedure di modulo sono compilate come parte del modulo, e vengono rese disponibili mediante *USE association* al modulo, come per tutte le altre variabili del modulo

Le procedure incluse nel modulo devono seguire la sezione dichiarativa del modulo e devono essere precedute dalla keyword **`CONTAINS`**. La keyword **`CONTAINS`** informa il compilatore che le procedure sono incluse nel modulo

Una procedure compilata all'interno di un modulo e utilizzata mediante *USE association* rappresenta un' INTERFACCIA ESPLICITA: ovvero, tutti i dettagli degli argomenti della procedura sono noti al compilatore. Quando le procedure non sono all'interno di un modulo (o di una *host unit*) hanno, di default, un' INTERFACCIA IMPLICITA

Moduli

Sintassi per le MODULE PROCEDURES:

```
MODULE name
    (dichiarazione dati da condividere)
CONTAINS
     MODULE PROCEDURES
END MODULE name
```

Esempio:

```
MODULE Globale
IMPLICIT NONE
REAL :: r, s
CONTAINS
REAL FUNCTION Calcola()
  IMPLICIT NONE
  READ(*,*) r, s
  Calcola = r * s
  RETURN
END FUNCTION Calcola
END MODULE Globale
PROGRAM Principale
USE Globale
IMPLICIT NONE
REAL :: c
c = Calcola()
write (*,*) "Risultato: ", c
END PROGRAM Principale
```

MODULE PROCEDURE

Restringere l'accesso al contenuto di un modulo

Nei moduli esiste la possibilità di accedere solo ad alcuni dati e/o procedure, utilizzando l'attributo **ONLY**:

```
USE module-name [, ONLY: { data/procedure-names } ]
```

Di default, le entità all'interno del modulo sono PUBBLICHE (**PUBLIC**), cioè accessibili a qualunque unità che esegua *USE association* a quel modulo

Ma, all'interno di un modulo, Fortran permette di specificare che alcune entità siano invece PRIVATE (**PRIVATE**) e non possono essere accessibili da altre unità di programma

Le entità private devono essere esplicitamente dichiarate usando la keyword riservata **PRIVATE**

Restringere l'accesso al contenuto di un modulo

In generale, è una buona pratica limitare l'accesso di dati e/o procedure in un modulo alle sole unità di programma che devono accedervi (*data hiding*)
Questo permette ai programmi di essere più modulari, più semplici da mantenere e da comprendere

Per controllare l'accessibilità si utilizzano gli attributi o dichiarazioni:

PRIVATE : il dato NON sarà disponibile ad unità di programma esterne al modulo (ma le procedure all'interno del modulo possono accedere a quel dato)

PUBLIC : il dato SARA' disponibile ad unità di programma esterne al modulo (**DEFAULT**)

PROTECTED : il dato sarà disponibile come *read-only* alle unità esterne: se, ad esempio, abbiamo definito una variabile **x** **PROTECTED** in un modulo e proviamo in un'altra unità di programmazione a modificarla, il compilatore ci restituisce il seguente errore:

```
Error: Variable 'x' is PROTECTED and can not appear in a variable definition context (assignment) at (1)
```

Restringere l'accesso al contenuto di un modulo

Sintassi per gestire l'accesso del contenuto di un modulo:

- Inserire la tipologia di accesso come attributo di una variabile (**non utilizzabile** per le procedure):

```
INTEGER, PRIVATE   :: count
REAL,    PUBLIC    :: voltage
REAL,    PROTECTED :: intensity
```

- Inserire la tipologia di accesso con un'istruzione cumulativa per un gruppo di elementi, da inserire prima che vengano dichiarati (**utilizzabile** per le procedure):

```
PUBLIC    :: list of public items
PRIVATE   :: list of private items
PROTECTED :: list of protected items
```

- L'istruzione **PRIVATE** può anche essere utilizzata senza un elenco di elementi: se un modulo contiene un'istruzione **PRIVATE** senza un elenco di elementi allora ogni elemento di dati e procedura nel modulo è privato. Deve essere inserita in intestazione

- Al contrario, se vogliamo specificare che tutti gli elementi siano pubblici devono essere elencati esplicitamente in un'istruzione **PUBLIC** seguita dalla lista che vogliamo sia pubblica

Come compilare i moduli?

Esempio: abbiamo un main program (**mainProg.f90**) che usa il modulo **testModule.f90**
(from https://fortranwiki.org/)

testModule.f90

```
module mathModule

  implicit none
  private
  real, public, parameter :: &
    pi = 3.1415 , &
    e = 2.7183 , &
    gamma = 0.57722

end module mathModule
```

mainProg.f90

```
program testing

  use mathModule
  implicit none

  print *, "gamma:", gamma

end program testing
```

Come compilare i moduli?

Compilazione separata:

1. si compilano il main program e il modulo, producendo i rispettivi files oggetto (***.o**) e il file del modulo (namemodule.mod – the module intermediate file): Il compilatore ha bisogno di accedere a questi file per poter leggere interfacce e variabili per compilare effettivamente il codice sorgente che utilizza i vari moduli

```
$ gfortran -c testModule.f90 mainProg.f90
$ ls
$ mainProg.f90  mainProg.o testModule.f90  testModule.o  mathmodule.mod
```

2. si esegue il linking per creare un unico eseguibile (**a.out**):

```
$ gfortran testModule.o mainProg.o
$ ls
$ a.out mainProg.f90  mainProg.o testModule.f90  testModule.o  mathmodule.mod
```

Come compilare i moduli?

In alternativa, **compilazione in un unico comando:**

```
$ gfortran testModule.f90 mainProg.f90 -o executable.exe
$ ls
$ executable.exe mainProg.f90 testModule.f90 mathmodule.mod
```

NOTA: I compilatori supportano opzioni come `-I` per indicare dove si trovano i files intermedi del modulo: `gfortran -c mainprogram.f90 -I $DIR_where_mod_files_are`

Come compilare i moduli?

In alternativa, **compilazione in un unico comando:**

```
$ gfortran testModule.f90 mainProg.f90 -o executable.exe
$ ls
$ executable.exe mainProg.f90 testModule.f90 mathmodule.mod
```

NOTA: I compilatori supportano opzioni come `-I` per indicare dove si trovano i files intermedi del modulo: `gfortran -c mainprogram.f90 -I $DIR_where_mod_files_are`

NOTA: L'ordine dei file di solito non conta se i moduli sono indipendenti, ma … ⚠️ **Attenzione!**

→ **nella compilazione separata (-c)**, l'ordine è essenziale: un modulo deve essere compilato prima di qualsiasi file che lo usa, perché il file `.mod` deve già esistere. Nel comando unico l'ordine non è obbligatorio, ma consigliato

→ **se ci sono moduli interdipendenti** (*use* a cascata), alcuni compilatori possono richiedere un ordine logico per risolvere tipi e interfacce: il file che contiene il modulo **deve comparire prima** di chi lo usa, così il compilatore trova il .mod

→ **per progetti grandi,** usare un Makefile o un sistema di build per gestire automaticamente l'ordine dei moduli evita errori imprevisti

Nota sui moduli Fortran

👉 I file binari **.mod** non seguono uno standard comune: ogni compilatore Fortran (come `gfortran` o `ifort`) li genera in un formato proprietario e incompatibile con gli altri. Di conseguenza, una libreria compilata con un determinato compilatore non può essere utilizzabile con un altro senza essere ricompilata, rendendo **complessa la distribuzione e il riutilizzo di librerie Fortran precompilate**

👉 In Fortran non esiste ancora un sistema davvero efficace per condividere moduli e librerie con la comunità o per contribuire facilmente al linguaggio. Anche strumenti moderni come `fpm` (`Fortran Package Manager`) cercano di risolvere il problema, ma la gestione delle dipendenze tra moduli diventa rapidamente complessa: **è spesso difficile riutilizzare un modulo scritto in un progetto all'interno di un altro**

Esercizi - Introduzione alle procedure

Si scriva un programma che utilizzi alternativamente le variabili sp, dp e qp del modulo *numeric_kinds* per calcolare l'area di un cerchio a partire da un raggio intero

```
MODULE numeric_kinds
- from M. Metcalf, J. Reid, and M. Cohen, "Modern Fortran Explained"
! named constants for 4-, 2-, and 1-byte integers:

integer, parameter :: &
i4b = selected_int_kind(9), &
i2b = selected_int_kind(4), &
i1b = selected_int_kind(2)

! and for single, double and quadruple precision reals:
integer, parameter :: &
sp = kind(1.0), &
dp = selected_real_kind(2*precision(1.0_sp)), &
qp = selected_real_kind(2*precision(1.0_dp))

END MODULE numeric_kinds
```

NOTA:
Si usi nel codice l'istruzione `REAL(x, kind)` che converte un numero intero in un tipo reale di precisione specificata

Interfaccia esplicita e implicita

Quando un programma richiama una sua procedura interna (*host association*), oppure quando un'unità di programma utilizza l'istruzione *USE association* ad un modulo, il compilatore è in grado di conoscere tutti i dati e le proprietà dei sottoprogrammi interni, o dei moduli e delle loro procedure, ovvero, possiamo dire che il compilatore riconosce l'INTERFACCIA della *program unit*. In questo caso, diciamo che l'INTERFACCIA è ESPLICITA

Quando invece un'unità di programma richiama una procedura esterna, il compilatore di norma non ha la possibilità di accedere a quella parte di codice: in questo caso, l'INTERFACCIA è IMPLICITA

Un modo per utilizzare un'interfaccia implicita (quindi il contenuto di un'unità di programma esterna) in modo che il compilatore ne riconosca dati e proprietà, è utilizzare un BLOCCO DI INTERFACCIA da inserire nella sezione di intestazione dell'unità chiamante

Blocco di Interfaccia

Un blocco di interfaccia è una copia esatta della sezione dichiarativa di un sottoprogramma, compreso la specifica di tipo dei suoi argomenti

Viene dichiarato con l'istruzione **INTERFACE**, seguita da una serie di corpi di interfaccia (i.e, sottoprogrammi) e chiusa con l'istruzione **END INTERFACE**

```
interface
    interface-body1
    interface-body2
     ……
end interface
```

Ogni corpo di interfaccia è costituito:

- dall'istruzione **SUBROUTINE** o **FUNCTION** iniziale della procedura esterna corrispondente,
- le istruzioni di specifica del tipo associate con i suoi argomenti e
- un'istruzione **END SUBROUTINE** o **END FUNCTION**

Il blocco di interfaccia specifica tutte le caratteristiche dell'interfaccia di una procedura esterna e il compilatore Fortran utilizza le informazioni nel blocco di interfaccia per eseguire i controlli di coerenza e per applicare funzionalità avanzate

Blocco di Interfaccia

Un blocco di interfaccia è una copia esatta della sezione dichiarativa di un sottoprogramma, compreso la specifica di tipo dei suoi argomenti.

Viene dichiarato con l'istruzione **INTERFACE**, seguita da una serie di corpi di interfaccia (i.e, sottoprogrammi) e chiusa con l'istruzione **END INTERFACE**

Ogni corpo di interfaccia è costituito:

- dall'istruzione **SUBROUTINE** o **FUNCTION** iniziale della procedura esterna corrispondente,
- le istruzioni di specifica del tipo associate con i suoi argomenti e
- un'istruzione **END SUBROUTINE** o **END FUNCTION**

Esempio:

```
INTERFACE
    FUNCTION prova (a)
        REAL, INTENT(IN) :: a
        REAL :: prova
    END FUNCTION prova
END INTERFACE
```

Il blocco di interfaccia specifica tutte le caratteristiche dell'interfaccia di una procedura esterna e il compilatore Fortran utilizza le informazioni nel blocco di interfaccia per eseguire i controlli di coerenza e per applicare funzionalità avanzate

Blocco di Interfaccia

IMPORTANTE:

In un blocco di interfaccia:

- possono essere aggiunte altre specifiche (come variabili locali), ma NON POSSONO ESSERE INSERITE:
  - altre procedure interne e
  - istruzioni di dati o formati diversi da quelli contenuti nel sottoprogramma

- non è importante rispettare l'ordine delle istruzioni

- il modo più conveniente di utilizzare un blocco di interfaccia per procedure esterne è di inserirlo in un modulo

- l'interfaccia viene posizionata nella sezione di intestazione dell'unità di programma richiamante insieme a tutte le dichiarazioni di tipo

Moduli con Blocco di Interfaccia

I blocchi di interfaccia risultano molto utili quando è necessario fornire interfacce esplicite a procedure compilate separatamente (esterne), scritte in versioni precedenti di Fortran o in altre linguaggi come C/C++

In questo caso, la scrittura di un blocco di interfaccia all'interno di un modulo consente ai programmi scritti in Fortran moderno di controllarne tutte le entità, e contemporaneamente ai programmi più vecchi o non Fortran di continuare a utilizzare le procedure invariate

Moduli con Blocco di Interfaccia

I blocchi di interfaccia risultano molto utili quando è necessario fornire interfacce esplicite a procedure compilate separatamente (esterne), scritte in versioni precedenti di Fortran o in altre linguaggi come C/C++

In questo caso, la scrittura di un blocco di interfaccia all'interno di un modulo consente ai programmi scritti in Fortran moderno di controllarne tutte le entità, e contemporaneamente ai programmi più vecchi o non Fortran di continuare a utilizzare le procedure invariate

Un modulo con un blocco di interfaccia è definito dalla seguente sintassi:

```
MODULE nome
    (dichiarazione dati da condividere)
    INTERFACE blocks
...
(insert other procedure interfaces here)
...
    END INTERFACE
END MODULE nome
```

A differenza delle procedure del modulo, NON esiste alcuna istruzione **CONTAINS** quando nei moduli sono inserite interfacce a sottoprogrammi esterni

Moduli con Blocco di Interfaccia

I blocchi di interfaccia risultano molto utili quando è necessario fornire interfacce esplicite a procedure compilate separatamente (esterne), scritte in versioni precedenti di Fortran o in altre linguaggi come C/C++

In questo caso, la scrittura di un blocco di interfaccia all'interno di un modulo consente ai programmi scritti in Fortran moderno di controllarne tutte le entità, e contemporaneamente ai programmi più vecchi o non Fortran di continuare a utilizzare le procedure invariate

Un modulo con un blocco di interfaccia è definito dalla seguente sintassi:

```
MODULE interface_definitions
(dichiarazione dati da condividere)
INTERFACE
 SUBROUTINE sort (array, n)
 IMPLICIT NONE
 REAL, DIMENSION(:), INTENT(INOUT) :: array
 INTEGER, INTENT(IN) :: n
 END SUBROUTINE sort
 ...
 (insert other procedure interfaces here)
 ...
  END INTERFACE
END MODULE interface_definitions
```

A differenza delle procedure del modulo, NON esiste alcuna istruzione **CONTAINS** quando nei moduli sono inserite interfacce a sottoprogrammi esterni

Moduli con Blocco di Interfaccia

Nota: per potersi interfacciare con il C/C++, il blocco di interfaccia viene spesso utilizzato insieme all'attributo `BIND(C)` e al modulo standard `ISO_C_BINDING`

Questa combinazione permette al compilatore di conoscere la convenzione di chiamata, i tipi di dato e i nomi simbolici delle routine esterne, rendendo possibile l'utilizzo di **librerie scritte in linguaggio C** (o C++ con linkage C - `BIND(C)` - all'interno di programmi Fortran, in modo sicuro e portabile

```
module c_interfaces
  use iso_c_binding
  implicit none

  interface
    subroutine c_fun(x) bind(C, name="c_fun")
      import :: c_double ! importa il tipo c_double dal modulo iso_c_binding
      real(c_double), value :: x
    end subroutine c_fun
  end interface

end module c_interfaces
```

Come regolarsi con l'uso dei blocchi di interfaccia?

- Quando possibile, meglio evitare i blocchi di interfaccia semplicemente inserendo tutte le procedure nei moduli e accedendo ai moduli appropriati tramite *use-association*
- Un modulo non deve specificare l'interfaccia di una procedura in esso già contenuta (l'interfaccia è già esplicita in un modulo e disponibile mediante *USE* association)
- Il blocco di interfaccia è utile per fornire interfacce esplicite a procedure compilate separatamente e scritte in versioni precedenti di Fortran o in altro linguaggi come C/C++
- Un modo semplice per creare le interfacce per una vasta libreria di vecchie *subroutines* o *functions,* e renderle a disposizione di tutte le unità di programma chiamanti, è inserirle in un modulo mediante blocco di interfaccia
- Ogni interfaccia (`INTERFACE`) in Fortran è una unità di visibilità separata (*scoping unit):* se il sottoprogramma usa variabili fittizie. devono essere dichiarate dentro il blocco `INTERFACE`, anche se nel programma principale esistono già variabili con lo stesso nome

Esercizi - Introduzione alle procedure

Esercizio 4. Si scriva un programma che chiami la subroutine esterna sort per ordinare gli elementi di un array, attraverso l'utilizzo di un blocco di interfaccia inserito:

a) all'interno del main program
b) all'interno di un modulo

```
SUBROUTINE sort (arr, n)
! Purpose:
! To sort real array "arr" into ascending order using a selection
! sort.
IMPLICIT NONE
INTEGER, INTENT(IN) :: n ! Number of values
REAL, DIMENSION(n), INTENT(INOUT) :: arr ! Array to be sorted
INTEGER :: i ! Loop index
INTEGER :: iptr ! Pointer to smallest value
INTEGER :: j ! Loop index
REAL :: temp ! Temp variable for swaps
! Sort the array
outer: DO i = 1, n-1
! Find the minimum value in arr(I) through arr(N)
iptr = i
inner: DO j = i+1, n
minval: IF ( arr(j) < arr(iptr) ) THEN
iptr = j
END IF minval
END DO inner
! iptr now points to the minimum value, so swap arr(iptr)
! with arr(i) if i /= iptr.
swap: IF ( i /= iptr ) THEN
temp = arr(i)
arr(i) = arr(iptr)
arr(iptr) = temp
END IF swap
END DO outer
END SUBROUTINE sort
```

«Scope» e «scoping units»

Abbiamo imparato che:

- in un programma Fortran è opportuno suddividere il codice in procedure, considerate unità di programmazione indipendenti (*subroutines o functions*)
- alcune di esse possono essere annidate all'interno di altre (procedure interne)

«Scope» e «scoping units»

Abbiamo imparato che:

- in un programma Fortran è opportuno suddividere il codice in procedure, considerate unità di programmazione indipendenti (*subroutines o functions*)
- alcune di esse possono essere annidate all'interno di altre (procedure interne)

… ma occorre considerare che:

- In ciascuna procedura, la **visibilità** delle entità (variabili, procedure, tipi derivati, interfacce, ecc.) dipende **sia dal punto in cui esse sono definite** (ad esempio se all'interno di una procedura, come argomento fittizio, o in un modulo usato), sia **dal modo in cui sono dichiarate** (ad esempio tramite `INTENT`, `PUBLIC/PRIVATE`, `SAVE`, ecc.)
- tale visibilità, detta ***scope*** o **ambito**, determina **l'insieme delle entità accessibili e modificabili** all'interno di una determinata unità di programma
- ogni unità può quindi essere vista come un **contesto di visibilità, ovvero, una *scoping unit*,** che racchiude le entità valide in quello specifico livello di definizione

«Scope» e «scoping units»

Le *scoping units* sono:

- Il programma principale
- Una procedura interna o esterna
- Un modulo
- Un blocco di interfaccia
- Un tipo derivato, *derived-data type*

Scoping Unit	Cosa contiene / *Scope* (visibilità)	Visibile da…
Programma principale	Variabili globali, procedure interne	Solo dal suo ***scope;*** accessibili da procedure interne
Subroutine / Function	Variabili fittizie, variabili locali	Solo dal suo ***scope;*** accessibili da procedure interne
Modulo (`MODULE`)	Tipi, procedure `PUBLIC/PRIVATE`, variabili, costanti	Altro codice può accedervi tramite `USE`
Interfaccia (`INTERFACE`)	Dichiarazioni di procedure, variabili fittizie	Solo nell'unità in cui l'interfaccia è definita o visibile
Tipo derivato (`TYPE`)	Componenti del tipo	Accessibili solo tramite variabili di quel tipo
Blocchi (`BLOCK`)	Variabili locali al blocco	Solo dentro il blocco

«Scope» e «scoping units»

Le visibilità/scope possono essere:

Visibilità globale (*global scope*):

- nomi di entità definite nel *main program*, o in un modulo o in procedure esterne: devono essere **univoci** all'interno dello stesso ambito globale (ad esempio, nomi del *main program*, procedure esterne, moduli e delle loro entità)

Visibilità locale (*local scope*):

- nomi definiti all'interno di una procedura interna (*subroutine/function*): il nome di un'entità locale **deve essere univoco** all'interno della stessa *scoping unit*, ma il suo nome può essere riutilizzato in un'altra *scoping unit* senza creare conflitti

Visibilità di blocco (*block scope*):

- nomi di variabili definite all'interno di un blocco (ad esempio `DO`, `IF`, `SELECT CASE`, `WHERE`) o di un costrutto specifico o blocco di interfaccia: i nomi di queste variabili sono visibili solo all'interno del blocco e indipendenti dalle variabili della *scoping unit* che li contiene

«Scope» e «scoping units»

- Se una *scoping unit* contiene completamente una o più *scoping units*, allora viene chiamata *host unit*: una *host unit* NON HA ACCESSO alle variabili locali di una procedura interna che contiene

- La *scoping unit* interna eredita automaticamente le definizioni delle entità dichiarate nella *host unit*, a meno che la *scoping unit* interna ridefinisca esplicitamente tali entità (vedi le variabili x e i in esempio di slide 65): questa eredità è chiamata *host association* (non si estende alle interfacce)

- Se la procedura interna utilizza un nome di variabile definito nella *host unit* senza ridefinirla, modifiche a quella variabile nella procedura interna passeranno anche all'unità *host* (vedi le variabili y e j in esempio di slide 65)

- Al contrario, se la procedura interna ridefinisce un nome di variabile utilizzato nell'unità *host*, attribuendo così a quella variabile un *local scope*, le modifiche ad essa apportate non influiranno sul valore della variabile con lo stesso nome nell'unità *host* (vedi le variabili x e i in esempio di slide 65)

«Scope» e «scoping units» Fortran for Scientists and Engineers» S.J.Chapman

```
MODULE module_example
IMPLICIT NONE
REAL :: x = 100.
REAL :: y = 200.
END MODULE
```

```
PROGRAM scoping_test
USE module_example
IMPLICIT NONE
INTEGER :: i = 1, j = 2
WRITE (*,'(A25,2I7,2F7.1)') ' Beginning:', i, j, x, y
CALL sub1 ( i, j )
WRITE (*,'(A25,2I7,2F7.1)') ' After sub1:', i, j, x, y
CALL sub2
WRITE (*,'(A25,2I7,2F7.1)') ' After sub2:', i, j, x, y

CONTAINS

SUBROUTINE sub2
REAL :: x
x = 1000.
y = 2000.
WRITE (*,'(A25,2F7.1)') ' In sub2:', x, y
END SUBROUTINE sub2

END PROGRAM scoping_test
```

```
SUBROUTINE sub1 (i,j)
IMPLICIT NONE
INTEGER, INTENT(INOUT) :: i, j
INTEGER, DIMENSION(5) :: array
WRITE (*,'(A25,2I7)') 'In sub1 before sub2:', i, j
CALL sub2
WRITE (*,'(A25,2I7)') 'In sub1 after sub2:', i, j
array = [ (1000*i, i=1,5) ]
WRITE (*,'(A25,7I7)') 'After array def in sub2:', i, j,array

CONTAINS

SUBROUTINE sub2
INTEGER :: i
i = 1000
j = 2000
WRITE (*,'(A25,2I7)') 'In sub1 in sub2:', i, j
END SUBROUTINE sub2

END SUBROUTINE sub1
```

NOTA: sub1 richiama la sua **sub2** interna: solo se non esistesse una subroutine locale con quel nome, il compilatore cercherebbe altrove name hiding

«Scope» e «scoping units» Fortran for Scientists and Engineers» S.J.Chapman

1. Quali sono le *scoping units* all'interno di questo programma?
2. Quali *scoping units* ospitano altre unità?
3. Quali oggetti in questo programma hanno un *global scope*?
4. Quali oggetti in questo programma hanno *local scope*?
5. Quali oggetti in questo programma sono ereditati dalla *host association*?
6. Quali oggetti in questo programma sono resi disponibili dall'associazione USE?
7. Prova ad eseguire il programma e a verificare cosa succede.

«Scope» e «scoping units» Fortran for Scientists and Engineers» S.J.Chapman

```
MODULE module_example
IMPLICIT NONE
REAL :: x = 100.
REAL :: y = 200.
END MODULE
```

SCOPING UNIT

```
PROGRAM scoping_test
USE module_example
IMPLICIT NONE
INTEGER :: i = 1, j = 2
WRITE (*,'(A25,2I7,2F7.1)') ' Beginning:', i, j, x, y
CALL sub1 ( i, j )
WRITE (*,'(A25,2I7,2F7.1)') ' After sub1:', i, j, x, y
CALL sub2
WRITE (*,'(A25,2I7,2F7.1)') ' After sub2:', i, j, x, y

CONTAINS

SUBROUTINE sub2
REAL :: x
x = 1000.
y = 2000.
WRITE (*,'(A25,2F7.1)') ' In sub2:', x, y
END SUBROUTINE sub2

END PROGRAM scoping_test
```

SCOPING UNIT

SCOPING UNIT

```
SUBROUTINE sub1 (i,j)
IMPLICIT NONE
INTEGER, INTENT(INOUT) :: i, j
INTEGER, DIMENSION(5) :: array
WRITE (*,'(A25,2I7)') 'In sub1 before sub2:', i, j
CALL sub2
WRITE (*,'(A25,2I7)') 'In sub1 after sub2:', i, j
array = [ (1000*i, i=1,5) ]
WRITE (*,'(A25,7I7)') 'After array def in sub2:', i, j,array

CONTAINS

SUBROUTINE sub2
INTEGER :: i
i = 1000
j = 2000
WRITE (*,'(A25,2I7)') 'In sub1 in sub2:', i, j
END SUBROUTINE sub2

END SUBROUTINE sub1
```

SCOPING UNIT

SCOPING UNIT

📝 **Possiamo definire due *subroutines* con lo stesso nome (sub2)**, *purché non appartengano allo stesso ambito di visibilità, scope* **!!**

«Scope» e «scoping units» Fortran for Scientists and Engineers» S.J.Chapman

```
MODULE module_example
IMPLICIT NONE
REAL :: x = 100.
REAL :: y = 200.
END MODULE
```

```
PROGRAM scoping_test
USE module_example
IMPLICIT NONE
INTEGER :: i = 1, j = 2
WRITE (*,'(A25,2I7,2F7.1)') ' Beginning:', i, j, x, y
CALL sub1 ( i, j )
WRITE (*,'(A25,2I7,2F7.1)') ' After sub1:', i, j, x, y
CALL sub2
WRITE (*,'(A25,2I7,2F7.1)') ' After sub2:', i, j, x, y

CONTAINS

SUBROUTINE sub2
REAL :: x
x = 1000.
y = 2000.
WRITE (*,'(A25,2F7.1)') ' In sub2:', x, y
END SUBROUTINE sub2

END PROGRAM scoping_test
```

HOST UNIT per la subroutine interna sub2

```
SUBROUTINE sub1 (i,j)
IMPLICIT NONE
INTEGER, INTENT(INOUT) :: i, j
INTEGER, DIMENSION(5) :: array
WRITE (*,'(A25,2I7)') 'In sub1 before sub2:', i, j
CALL sub2
WRITE (*,'(A25,2I7)') 'In sub1 after sub2:', i, j
array = [ (1000*i, i=1,5) ]
WRITE (*,'(A25,7I7)') 'After array def in sub2:', i, j,array

CONTAINS

SUBROUTINE sub2
INTEGER :: i
i = 1000
j = 2000
WRITE (*,'(A25,2I7)') 'In sub1 in sub2:', i, j
END SUBROUTINE sub2

END SUBROUTINE sub1
```

HOST UNIT per la subroutine interna sub2

«Scope» e «scoping units» Fortran for Scientists and Engineers» S.J.Chapman

Global scope

```
MODULE module_example
IMPLICIT NONE
REAL :: x = 100.
REAL :: y = 200.
END MODULE
```

Global scope

```
PROGRAM scoping_test
USE module_example
IMPLICIT NONE
INTEGER :: i = 1, j = 2
WRITE (*,'(A25,2I7,2F7.1)') ' Beginning:', i, j, x, y
CALL sub1 ( i, j )
WRITE (*,'(A25,2I7,2F7.1)') ' After sub1:', i, j, x, y
CALL sub2
WRITE (*,'(A25,2I7,2F7.1)') ' After sub2:', i, j, x, y

CONTAINS

SUBROUTINE sub2
REAL :: x
x = 1000.
y = 2000.
WRITE (*,'(A25,2F7.1)') ' In sub2:', x, y
END SUBROUTINE sub2

END PROGRAM scoping_test
```

Global scope

```
SUBROUTINE sub1 (i,j)
IMPLICIT NONE
INTEGER, INTENT(INOUT) :: i, j
INTEGER, DIMENSION(5) :: array
WRITE (*,'(A25,2I7)') 'In sub1 before sub2:', i, j
CALL sub2
WRITE (*,'(A25,2I7)') 'In sub1 after sub2:', i, j
array = [ (1000*i, i=1,5) ]
WRITE (*,'(A25,7I7)') 'After array def in sub2:', i, j,array

CONTAINS

SUBROUTINE sub2
INTEGER :: i
i = 1000
j = 2000
WRITE (*,'(A25,2I7)') 'In sub1 in sub2:', i, j
END SUBROUTINE sub2

END SUBROUTINE sub1
```

«Scope» e «scoping units» Fortran for Scientists and Engineers» S.J.Chapman

```
MODULE module_example
IMPLICIT NONE
REAL :: x = 100.
REAL :: y = 200.
END MODULE
```

```
PROGRAM scoping_test
USE module_example
IMPLICIT NONE
INTEGER :: i = 1, j = 2
WRITE (*,'(A25,2I7,2F7.1)') ' Beginning:', i, j, x, y
CALL sub1 ( i, j )
WRITE (*,'(A25,2I7,2F7.1)') ' After sub1:', i, j, x, y
CALL sub2
WRITE (*,'(A25,2I7,2F7.1)') ' After sub2:', i, j, x, y

CONTAINS

SUBROUTINE sub2                          Local scope
REAL :: x
x = 1000.
y = 2000.
WRITE (*,'(A25,2F7.1)') ' In sub2:', x, y
END SUBROUTINE sub2

END PROGRAM scoping_test
```

```
SUBROUTINE sub1 (i,j)
IMPLICIT NONE
INTEGER, INTENT(INOUT) :: i, j
INTEGER, DIMENSION(5) :: array
WRITE (*,'(A25,2I7)') 'In sub1 before sub2:', i, j
CALL sub2
WRITE (*,'(A25,2I7)') 'In sub1 after sub2:', i, j
array = [ (1000*i, i=1,5) ]
WRITE (*,'(A25,7I7)') 'After array def in sub2:', i, j,array

CONTAINS

SUBROUTINE sub2                          Local scope
INTEGER :: i
i = 1000
j = 2000
WRITE (*,'(A25,2I7)') 'In sub1 in sub2:', i, j
END SUBROUTINE sub2

END SUBROUTINE sub1
```

«Scope» e «scoping units» Fortran for Scientists and Engineers» S.J.Chapman

```
MODULE module_example
IMPLICIT NONE
REAL :: x = 100.
REAL :: y = 200.
END MODULE
```

```
PROGRAM scoping_test
USE module_example
IMPLICIT NONE
INTEGER :: i = 1, j = 2
WRITE (*,'(A25,2I7,2F7.1)') ' Beginning:', i, j, x, y
CALL sub1 ( i, j )
WRITE (*,'(A25,2I7,2F7.1)') ' After sub1:', i, j, x, y
CALL sub2
WRITE (*,'(A25,2I7,2F7.1)') ' After sub2:', i, j, x, y

CONTAINS

SUBROUTINE sub2
REAL :: x
x = 1000.
y = 2000.
WRITE (*,'(A25,2F7.1)') ' In sub2:', x, y
END SUBROUTINE sub2

END PROGRAM scoping_test
```

x e y ereditate per *use-association*

x e y ereditate per *host-association*

```
SUBROUTINE sub1 (i,j)
IMPLICIT NONE
INTEGER, INTENT(INOUT) :: i, j
INTEGER, DIMENSION(5) :: array
WRITE (*,'(A25,2I7)') 'In sub1 before sub2:', i, j
CALL sub2
WRITE (*,'(A25,2I7)') 'In sub1 after sub2:', i, j
array = [ (1000*i, i=1,5) ]
WRITE (*,'(A25,7I7)') 'After array def in sub2:', i, j,array

CONTAINS

SUBROUTINE sub2
INTEGER :: i
i = 1000
j = 2000
WRITE (*,'(A25,2I7)') 'In sub1 in sub2:', i, j
END SUBROUTINE sub2

END SUBROUTINE sub1
```

i e j ereditate per *host-association*

«Scope» e «scoping units» Fortran for Scientists and Engineers» S.J.Chapman

```
MODULE module_example
IMPLICIT NONE
REAL :: x = 100.
REAL :: y = 200.
END MODULE
```

```
PROGRAM scoping_test
USE module_example
IMPLICIT NONE
INTEGER :: i = 1, j = 2
WRITE (*,'(A25,2I7,2F7.1)') ' Beginning:', i, j, x, y
CALL sub1 ( i, j )
WRITE (*,'(A25,2I7,2F7.1)') ' After sub1:', i, j, x, y
CALL sub2
WRITE (*,'(A25,2I7,2F7.1)') ' After sub2:', i, j, x, y

CONTAINS

SUBROUTINE sub2
REAL :: x
x = 1000.
y = 2000.
WRITE (*,'(A25,2F7.1)') ' In sub2:', x, y
END SUBROUTINE sub2

END PROGRAM scoping_test
```

La variabile x: *local scope* in sub2
le modifiche non influiranno in PROGRAM

```
SUBROUTINE sub1 (i,j)
IMPLICIT NONE
INTEGER, INTENT(INOUT) :: i, j
INTEGER, DIMENSION(5) :: array
WRITE (*,'(A25,2I7)') 'In sub1 before sub2:', i, j
CALL sub2
WRITE (*,'(A25,2I7)') 'In sub1 after sub2:', i, j
array = [ (1000*i, i=1,5) ]
WRITE (*,'(A25,7I7)') 'After array def in sub2:', i, j,array

CONTAINS

SUBROUTINE sub2
INTEGER :: i
i = 1000
j = 2000
WRITE (*,'(A25,2I7)') 'In sub1 in sub2:', i, j
END SUBROUTINE sub2

END SUBROUTINE sub1
```

La variabile i: *local scope* in sub2
Le modifiche non influiranno in sub1

«Scope» e «scoping units» Fortran for Scientists and Engineers» S.J.Chapman

```
MODULE module_example
IMPLICIT NONE
REAL :: x = 100.
REAL :: y = 200.
END MODULE
```

```
PROGRAM scoping_test
USE module_example
IMPLICIT NONE
INTEGER :: i = 1, j = 2
WRITE (*,'(A25,2I7,2F7.1)') ' Beginning:', i, j, x, y
CALL sub1 ( i, j )
WRITE (*,'(A25,2I7,2F7.1)') ' After sub1:', i, j, x, y
CALL sub2
WRITE (*,'(A25,2I7,2F7.1)') ' After sub2:', i, j, x, y

CONTAINS

SUBROUTINE sub2
REAL :: x
x = 1000.
y = 2000.
WRITE (*,'(A25,2F7.1)') ' In sub2:', x, y
END SUBROUTINE sub2

END PROGRAM scoping_test
```

La variabile y ereditata per *host-association*:
le modifiche influiranno in PROGRAM

```
SUBROUTINE sub1 (i,j)
IMPLICIT NONE
INTEGER, INTENT(INOUT) :: i, j
INTEGER, DIMENSION(5) :: array
WRITE (*,'(A25,2I7)') 'In sub1 before sub2:', i, j
CALL sub2
WRITE (*,'(A25,2I7)') 'In sub1 after sub2:', i, j
array = [ (1000*i, i=1,5) ]
WRITE (*,'(A25,7I7)') 'After array def in sub2:', i, j,array

CONTAINS

SUBROUTINE sub2
INTEGER :: i
i = 1000
j = 2000
WRITE (*,'(A25,2I7)') 'In sub1 in sub2:', i, j
END SUBROUTINE sub2

END SUBROUTINE sub1
```

La variabile j ereditata per *host-association*:
le modifiche influiranno in sub1

«Scope» e «scoping units» Fortran for Scientists and Engineers» S.J.Chapman

1. Quali sono le *scoping units* all'interno di questo programma?
Ogni modulo, programma principale, procedura interna o esterna

2. Quali *scoping units* ospitano altre unità?
Il programma principale e la subroutine esterna sub1

3. Quali oggetti in questo programma hanno un *global scope*?
(nomi del modulo, del programma principale e della subroutine esterna sub1

4. Quali oggetti in questo programma hanno *local scope*?
Tutti gli altri oggetti dei sottoprogrammi interni, compresi i loro nomi

5. Quali oggetti in questo programma sono ereditati dalla *host association*?
Tutti gli oggetti nelle due subroutine interne, ***ad eccezione di quegli oggetti esplicitamente ridefiniti nella subroutine interna. La variabile x è locale alla prima subroutine interna****, la variabile y è ereditata dal programma principale che è* ***la host unit. La variabile i è locale alla seconda subroutine interna****, mentre j è ereditata dal sub1, che è anche host unit*

6. Quali oggetti in questo programma sono resi disponibili dall'associazione USE?
Le variabili x e y

«Scope» e «scoping units»

Tipo di scope	Dove si applica	Entità visibili	Note / regole principali	Esempio
Global scope	Programma principale, modulo, procedure esterne	Variabili, procedure, tipi derivati definiti a livello di programma/procedura/ modulo	Nominalmente "globali" all'interno dello stesso programma/procedura/modulo. **Nomi devono essere univoci** nel loro ambito globale	Variabile in un modulo con attributo `PUBLIC`
Local scope	Procedure interne ed esterne	Variabili locali, parametri formali, procedure interne	**Nomi devono essere univoci all'interno della *scoping unit*,** possono essere riutilizzati in altre unità	Variabile **`i`** in `SUBROUTINE` **`sub1(i)`**
Block scope	Blocchi delimitati dentro una procedura o programma (`DO, IF, WHERE, SELECT CASE`)	Variabili dichiarate nel blocco	Visibili solo all'interno del blocco. **Possono avere lo stesso nome di variabili locali della *scoping unit* senza conflitti**	`DO j = 1,10; INTEGER :: k; … END DO` → `k` ha block scope
Interface scope	Blocco `INTERFACE … END INTERFACE`	Procedure dichiarate nell'interfaccia, eventuali tipi locali temporanei	Visibile solo all'interno del blocco di interfaccia; serve per fornire un'interfaccia esplicita a procedure esterne	`INTERFACE; SUBROUTINE sub_ext(a); INTEGER :: a; END SUBROUTINE; END INTERFACE`

Referenze

S.J. Chapman, 'Fortran for Scientists and Engineers', Fourth Edition (2018), **Chapter 7 & 13**

M. Metcalf et al., 'Modern Fortran explained', Fifth Edition (2018), **Chapter 5**

Learn — Fortran Programming Language

NOTA: Gli **esempi di programmazione e alcuni esercizi** contenuti in questa lezione si possono ritrovare in **Chapter 7 & Chapter 13** di S.J. Chapman, 'Fortran for Scientists and Engineers', Fourth Edition (2018)

ESERCIZI @:
SCAI Training / Intro Fortran Bologna · GitLab

Esercizi - Introduzione alle procedure

Esercizio 1. Si scriva una subroutine che calcoli l'ipotenusa di un triangolo rettangolo

Esercizio 2. Si scrivano una serie di subroutine riutilizzabili in grado di determinare le proprietà statistiche di un insieme di dati di numeri reali in un array. L'insieme delle subroutine dovrebbe includere:

a) una subroutine per determinare il valore massimo in un set di dati e la sua posizione
b) una subroutine per determinare il valore minimo in un set di dati e la sua posizione
c) una subroutine per determinare la media (media) e la deviazione standard del set di dati.
d) una subroutine per determinare la mediana del set di dati

Esercizio 3. Si scriva un programma che utilizzi alternativamente le variabili sp, dp e qp del modulo *numeric_kinds* visto nella lezione «Introduzione alle procedure» (slide 34) per calcolare l'area di un cerchio

Esercizi - Introduzione alle procedure

Esercizio 4. Si scriva un programma che chiami la subroutine esterna sort per ordinare gli elementi di un array, attraverso l'utilizzo di un blocco di interfaccia inserito:

a) all'interno del main program
b) all'interno di un modulo

```
SUBROUTINE sort (arr, n)
! Purpose:
! To sort real array "arr" into ascending order using a selection
! sort.
IMPLICIT NONE
INTEGER, INTENT(IN) :: n ! Number of values
REAL, DIMENSION(n), INTENT(INOUT) :: arr ! Array to be sorted
INTEGER :: i ! Loop index
INTEGER :: iptr ! Pointer to smallest value
INTEGER :: j ! Loop index
REAL :: temp ! Temp variable for swaps
! Sort the array
outer: DO i = 1, n-1
! Find the minimum value in arr(I) through arr(N)
iptr = i
inner: DO j = i+1, n
minval: IF ( arr(j) < arr(iptr) ) THEN
iptr = j
END IF minval
END DO inner
! iptr now points to the minimum value, so swap arr(iptr)
! with arr(i) if i /= iptr.
swap: IF ( i /= iptr ) THEN
temp = arr(i)
arr(i) = arr(iptr)
arr(iptr) = temp
END IF swap
END DO outer
END SUBROUTINE sort
```

Esercizi - Introduzione alle procedure

Esercizio 5. Si scriva una subroutine esterna che calcoli la gittata di un proiettile in uscita dalla bocca di un cannone, ricevendo come argomenti fittizi la velocità iniziale υ_0 del proiettile, e l'angolo θ iniziale della traiettoria rispetto al terreno e si scriva il testing program per la subroutine.

Utilizzare il fattore di conversione da gradi a radianti:

```
fattore conversione gradi-->radianti = 0.01745329
```

La formula della gittata, è:

```
gittata = 2 v0**2 * COS(theta) * SIN(theta) /g
```

a) Si modifichi la subroutine precedente in modo che stampi la gittata massima e l'angolo corrispondente per il range di angoli da 0 a 90 gradi.

Esercizi - Introduzione alle procedure

Esercizio 6. La derivata di una funzione continua $f(x)$f(x) è definita dall'equazione:

$$\frac{d}{dx}f(x) = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

Che in una funzione campionata, questa definizione diventa:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{\Delta x}$$

Si consideri un vettore vect che contiene nsamp campioni di una funzione, presi con passo dx tra un campione e l'altro. Scrivere una subroutine che calcoli la derivata di questo vettore usando la formula sopra. La subroutine deve verificare che dx sia maggiore di zero, per evitare errori di divisione per zero.Per verificare il corretto funzionamento della subroutine, si deve generare un insieme di dati la cui derivata è nota e confrontare il risultato della subroutine con il valore corretto. Una funzione adatta a questo scopo è sin(x), poiché dalla matematica elementare sappiamo che:

$$\frac{d}{dx}(\sin x) = \cos x$$

Generare un vettore di input contenente 100 valori della funzione sin(x), a partire da x=0, con passo Δx=0.05. Si calcoli la derivata del vettore con la subroutine e si confrontino i valori ottenuti con quelli corretti. Quanto si avvicina la routine al calcolo corretto della derivata?

Esercizi - Introduzione alle procedure

Esercizio 7. Si scriva una subroutine che calcoli la pendenza m e l'intercetta b della retta di regressione lineare che meglio approssima un insieme di dati in input, e calcoli anche il coefficiente di correlazione della regressione.
I punti dati (x,y) saranno passati alla subroutine tramite due array di input, X e Y.
Le formule per la pendenza, l'intercetta e per il coefficiente di correlazione r sono fornite di seguente:

$$m = \frac{(\Sigma xy) - (\Sigma x)\bar{y}}{(\Sigma x^2) - (\Sigma x)\bar{x}}$$

pendenza

$$b = \bar{y} - m\bar{x}$$

intercetta

$$r = \frac{n(\Sigma xy) - (\Sigma x)(\Sigma y)}{\sqrt{[(n\Sigma x^2) - (\Sigma x)^2][(n\Sigma y^2) - (\Sigma y)^2]}}$$

coefficiente di correlazione

GRAZIE

Angela Acocella
a.acocella@cineca.it