Il Linguaggio Fortran 90/95

Raucci Biagio

Indice

Premessa

La storia del linguaggio Fortran inizia nel 1953 quando John Backus, presso l'International Business Machines Corporation, iniziò a sviluppare un sistema di programmazione automatica che si prefiggeva l'ambizioso compito di convertire programmi, scritti secondo una notazione matematica, in istruzioni macchina per l'allora avveniristico sistema IBM 704. Nel 1954 fu così pubblicato il primo rapporto preliminare sul nuovo sistema, chiamato IBM Mathematical FORmula TRANslation System, nome successivamente abbreviato in FORTRAN. Dopo diversi processi di standardizzazione (FORTRAN II, FORTRAN 66, FORTRAN 77, Fortran 90, Fortran 95) oggi il Fortran è ancora il linguaggio più usato per lo sviluppo di applicazioni scientifiche ed ingegneristiche. Non solo, ma dopo le innovazioni introdotte con le ultime azioni di standardizzazione (ISO 1991/ANSI 1992 e ISO/IEC 1997 da cui prima il nome di Fortran 90 e poi Fortran 95) è ritornato ad essere un linguaggio moderno che offre costrutti e metodi tipici dei linguaggi di programmazione più evoluti ed innovativi. Tutto questo nella continuità con il passato; infatti, il precedente Fortran 77 viene visto come un sottoinsieme del Fortran 90/95 in modo da garantire la sopravvivenza dell'enorme patrimonio software esistente.

Le innovazioni principali introdotte dai nuovi standard riguardano essenzialmente i seguenti argomenti:

- Formato libero, svincolato dal vecchio formato fisso, orientato alle colonne, che risentiva ancora dell'impostazione tipica delle schede perforate.
- Costrutti per la definizione di procedure e strutture dati modulari, tesi a fornire una forma semplice e sicura per incapsulare dati e procedure.
- Tipi di dati definiti dall'utente, derivati da uno o più dei tipi di dati predefiniti.
- Operazioni globali su array e definizione di un set di funzioni intrinseche che operano su array.
- Introduzione dei puntatori, che consentono la creazione e la manipolazione di strutture dati dinamiche.
- Miglioramento dei costrutti per la computazione numerica, inclusa la disponibilità di tipi numerici intrinseci parametrizzati tesi a risolvere i problemi di portabilità del codice.

- Tipi di dati intrinseci non numerici parametrizzati, che consentono di supportare più di un set di caratteri contemporaneamente.

In queste note vengono trattate in modo completo gli aspetti più moderni ed efficienti del Fortran 90/95. Sono state, invece, volutamente tralasciate tutte quelle forme divenute obsolete o ufficialmente definite come tali dallo standard e candidate ad essere rimosse dal prossimo lavoro di standardizzazione (come il *formato fisso* di editing, l'`IF` aritmetico, le *funzioni interne a singola istruzione*, la dichiarazione di tipo `CHARACTER*`n, l'istruzione `GOTO` *calcolato*, le *terminazioni di* `DO` *condivise* o su istruzioni diverse da `END DO`) e quelle già eliminate dallo standard del Fortran 95 (costanti di Hollerith, le istruzioni `PAUSE` e `ASSIGN` ed il `GOTO` *assegnato*, l'uso di variabili di ciclo di tipo reale o doppia precisione). Sono state, inoltre, del tutto ignorate dalla trattazione tutte quelle forme appartenenti ad un Fortran primordiale inerenti la gestione dei dati condivisi (blocchi `COMMON`, istruzione `EQUIVALENCE`), la formattazione dei record e la gestione delle memorie di massa (gli specificatori `BLANK`, `ERR` ed `END` e i descrittori `P`, `BZ`, `BN`, `S`, `SP` ed `SS`) o relativi alla dichiarazione o alla inizializzazione di variabili (istruzioni `IMPLICIT` e `DATA`, sottoprogrammi `BLOCK DATA`, lo specificatore di tipo `DOUBLE PRECISION` e le dichiarazioni di array di dimensioni fittizie) e, infine, alcune strutture di controllo ridondanti e superate (istruzione `CONTINUE`, subroutine ad ingressi ed uscite multipli), tutti efficacemente sostituite da forme più moderne ed efficienti. Per la stessa ragione è stato dedicato pochissimo spazio all'*istruzione di salto incondizionato*, all'inclusione di file attraverso l'istruzione `INCLUDE` e alle istruzioni di dichiarazione `PARAMETER` e `DIMENSION` nonché al meccanismo di dichiarazione implicita dei dati, inseriti comunque nella trattazione ma esclusivamente per motivi di completezza.

Capitolo 1

Tipi ed espressioni

1.1 Elementi di base del linguaggio

Il codice sorgente di un programma Fortran consiste in un insieme di *unità di programma* compilate separatamente. Tali unità annoverano un *programma principale* e, opzionalmente, uno o più *sottoprogrammi* (*subroutine* o *funzioni*) e *moduli*. Ciascuna unità di programma, poi, consiste in una serie di *istruzioni* o *proposizioni* (*statement*) e di *linee di commento* opzionali, ma cominciano *sempre* con una istruzione `PROGRAM`, `SUBROUTINE`, `FUNCTION` o `MODULE`, e terminano sempre con una istruzione `END`.

1.1.1 Istruzioni e commenti

Le *istruzioni* Fortran si dividono in *eseguibili* o *non eseguibili*. Le prime descrivono le azioni che un programma può svolgere quando viene eseguito; le seconde forniscono informazioni necessarie affinché un programma venga eseguito correttamente.

Esempi di istruzioni eseguibili sono le istruzioni di *assegnazione* o quelle di *input/output*, come le seguenti:

```
y = SIN(3.14/2.)+2.5
WRITE(*,50) var1, var2
```

Esempi di istruzioni non eseguibili sono, invece, le istruzioni di *dichiarazione* o quelle di *intestazione*:

```
PROGRAM prova
INTEGER, SAVE :: i, j
```

A queste si affiancano poi le *istruzioni per la specifica di formato* (o *istruzioni* `FORMAT`) che forniscono direttive per le istruzioni di input/output. La seguente riga di programma rappresenta un esempio di istruzione di specificazione di formato:

```
100 FORMAT(1X,'Risultati:',2X,I3,3(1X,E9.3))
```

Quale che sia la sua funzione, ciascuna istruzione consiste in una singola riga (o *record*) di programma, opzionalmente seguita da una o più *righe di continuazione*.

Su una singola riga possono apparire anche *istruzioni multiple* separate da un simbolo di punto e virgola ";", come mostrano i seguenti esempi:

```
INTEGER :: i, j, k; LOGICAL :: vero=.TRUE.
i = 0; j = 1; k = k+1
```

Questa possibilità può aiutare a rendere un blocco di istruzioni più compatto ma può ledere in leggibilità se le istruzioni da elencare non sono molto "brevi".

Come anticipato poc'anzi, un'istruzione può anche essere "spezzata" e continuata sulla riga successiva facendo terminare la riga corrente (e, facoltativamente, facendo iniziare la riga successiva) con il carattere di "ampersand" (&), come nel seguente esempio:

```
REAL(KIND=4), ALLOCATABLE(:,:), SAVE :: matrice1, matrice2, &
                                        appoggio, risultato_finale
```

Tranne che negli oggetti di tipo *stringa*, in ciascun punto di un programma la presenza di un punto esclamativo indica che la restante parte della linea è da intendersi come un *commento* e, pertanto, orientata alla lettura del programmatore e non della macchina. I commenti permettono al programmatore di spiegare o *documentare* il funzionamento di un programma. Essi possono apparire in ciascun punto di un programma e sono preceduti dal simbolo di punto esclamativo "!". Ne consegue la possibilità di inserire un commento nella stessa riga di una istruzione, a valle della stessa.

E' sempre buona norma inserire dei commenti in un programma allo scopo di aumentare la leggibilità di un costrutto o per spiegare la funzione di un blocco di istruzioni o di un'intera unità di programma. D'altra parte i commenti vengono semplicemente ignorati dal compilatore per cui anche un loro uso "abbondante" non condizionerebbe in alcun modo le caratteristiche del file eseguibile. Il seguente frammento di programma mostra l'utilizzo dei commenti:

```
PROGRAM abc
!
! Lo scopo del programma è quello di ...
!
  INTEGER :: i, j    ! indici di cicli a conteggio
  REAL :: x          ! soluzione dell'equazione
  ...
  x = LOG(3.)        ! Attenzione: logaritmo naturale
  ...
END PROGRAM abc
```

L'unica eccezione a questa regola è quando il punto esclamativo è presente in una stringa di caratteri:

```
PRINT*, "Il procedimento non converge!!! Diminuire la tolleranza!"
```

1.1.2 Il set di caratteri Fortran

Tutte le istruzioni di un programma Fortran vengono definite utilizzando un gruppo di 84 caratteri. Tali caratteri sono stati scelti perché disponibili su ciascuna macchina e rappresentabili su ciascun display o altro dispositivo di input/output.

Il set di caratteri Fortran consta dei caratteri alfanumerici e di un ristretto gruppo di simboli speciali. I caratteri alfanumerici prevedono le 26 lettere dell'alfabeto inglese, sia nella forma minuscola (`a-z`) che nella forma maiuscola (`A-Z`), le 10 cifre decimali (`0-9`), ed il carattere *underscore* ("_") Tutti i caratteri speciali sono, invece, elencati nella seguente tabella:

Carattere	*Descrizione*	*Carattere*	*Descrizione*
	spazio	=	uguale
+	più	-	meno
*	asterisco	/	"slash"
(	parentesi aperta	)	parentesi chiusa
,	virgola	.	punto
'	apice	"	doppi apici
:	due punti	;	punto e virgola
!	punto esclamativo	&	"ampersand"
%	percentuale	<	minore di
>	maggiore di	$	dollaro
?	punto interrogativo		

Agli ultimi due simboli, `$` e `?`, tuttavia, non è attribuito alcun particolare significato nel linguaggio.

Accanto ai simboli speciali del vocabolario base, sono da considerare anche le *parole chiave* del linguaggio (ad esempio `IF`, `THEN`, `SUBROUTINE`, etc) che da un punto di vita sintattico vanno riguardati come simboli unici. Il linguaggio, inoltre, non prevede *parole riservate* sicché una specifica sequenza di caratteri può essere interpretata, alla luce del contesto sintattico in cui figura, come parola chiave oppure come *nome simbolico* definito dall'utente.

E' da osservare che, a parte che per gli oggetti stringa di caratteri, il Fortran *non* distingue tra le lettere maiuscole e quelle minuscole. La scelta di una delle due possibilità, pertanto, può essere effettuata in base a motivi di gusto o di leggibilità del programma.

Si noti, inoltre, che nel Fortran gli spazi bianchi (*blank*) sono *significativi* per cui essi generalmente *non* sono ammessi nei nomi degli identificatori né nelle costanti numeriche, ma vanno inseriti obbligatoriamente tra due parole consecutive. Sono, invece, opzionali nelle parole chiave "composte", come ad esempio `ELSEIF`, `ENDWHERE`, `INOUT` che possono, pertanto, essere scritte anche come `ELSE IF`, `END WHERE`, `IN OUT`.

1.1.3 Formato di un programma

Il Fortran 90/95 supporta due forme di codice sorgente: il vecchio formato risalente al Fortran 77 (*formato fisso*) e la nuova forma introdotta dal Fortran 90 (*formato libero*). Il primo formato è del tutto obsolescente per cui non verrà trattato in queste note.

Nella forma sorgente libera, le colonne non sono più riservate sicché le istruzioni Fortran possono partire da un punto qualsiasi di una linea. Una linea sorgente può contenere fino ad un massimo di 132 caratteri.

I nomi degli identificatori consistono in stringhe di caratteri con lunghezza variabile da 1 a 31 elementi, l'unica restrizione essendo rappresentata dal fatto che il primo carattere del nome deve essere una lettera.

Come detto in precedenza, il Fortran *non* è un linguaggio *case-sensitive* per cui l'uso di caratteri minuscoli o maiuscoli è, tranne che negli oggetti *stringa*, indifferente. Tuttavia, la scelta di usare, a seconda dei casi, nomi maiuscoli o minuscoli può servire a migliorare la leggibilità di un programma.

I commenti sono rappresentati da tutto quanto segue, su di una linea, un punto esclamativo. Un commento può cominciare in un punto qualsiasi di una riga di programma, con l'effetto che la parte seguente della linea è del tutto ignorata dal compilatore:

```
! Lunghezza di una barra metallica
REAL :: length1 ! lunghezza iniziale in mm (a temperatura ambiente)
REAL :: length2 ! lunghezza finale in mm (dopo il raffreddamento)
```

Una riga di programma può essere prolungata su una o più *linee di continuazione* appendendo un ampersand (`&`) ad ogni riga "da continuare". Anche una riga di continuazione può cominciare con un ampersand, nel qual caso l'istruzione riprende a partire dal primo carattere che segue il carattere `&`; in caso contrario, *tutti* i caratteri sulla linea fanno parte della linea di continuazione, *inclusi* gli spazi bianchi iniziali. Ciò è da tener presente quando le istruzioni continuate coinvolgono stringhe di caratteri. Quanto detto è ben descritto dai seguenti esempi:

```
log_gamma = f + (y-0.5)*log(y) - y + 0.91893853320 +  &
            (((-0.00059523810*z + 0.00079365079)*z -  &
            0.00277777778)*z + 0.08333333333)/y

WRITE(*,*) "Seconda Università degli Studi di Napoli&
            & Dipartimento di Ingegneria Aerospaziale"
```

Una singola riga di programma può contenere istruzioni multiple. In questi casi deve essere utilizzato il simbolo di punto e virgola "`;`" come separatore delle singole istruzioni, come nel seguente esempio:

```
a = 2; b = 7; c = 3
```

Una riga di programma può essere "etichettata" mediante una stringa numerica. Ciò fa in modo che questa riga possa essere "riferita" da una istruzione in un punto qualsiasi della *stessa* unità di programma. Istruzioni etichettate sono, essenzialmente, le istruzioni `FORMAT` ed il "target" delle istruzioni `GOTO`. Le etichette (*label*) consistono in un "campo numerico", ossia una stringa di caratteri numerici. Gli unici vincoli da rispettare sono rappresentati dal fatto che una etichetta non può essere preceduta, sulla sua riga di appartenenza, da alcun carattere diverso dallo spazio bianco, e la sua lunghezza non può eccedere le cinque cifre (in altre parole una etichetta può

essere compresa tra `1` e `99999`). A parte quello di attribuire un "nome" ad una riga, un'etichetta non ha altri significati; in particolare, essa *non* è un numero di riga e *non* ha alcun rapporto con l'ordine in cui vengono eseguite le istruzioni. In altre parole, una riga potrebbe avere l'etichetta `1000` e la successiva avere etichetta `10`: le due righe sarebbero eseguite nello stesso ordine in cui sono scritte, indipendentemente dai numeri delle loro label.

1.1.4 Inclusione di file sorgenti

La sequenza di istruzioni che compongono una unità di programma può essere interrotta allo scopo di inserire il contenuto di un altro file sorgente. Ciò può aver luogo a mezzo della istruzione `INCLUDE` la cui sintassi è:

```
INCLUDE "\textit{nome\_file}"
```

in cui *`nome_file`* è il nome del file (con, eventualmente, il relativo *path*) che deve essere "incluso" nel programma. Questo nome può essere racchiuso, indifferentemente, fra apici singoli o doppi.

Il file incluso viene "raggiunto" dal compilatore quando questo incontra, nell'unità ospite, l'istruzione `INCLUDE`; una volta compilato questo frammento di programma, il compilatore riprende la compilazione dell'unità di programma ospite a partire dalla riga immediatamente successiva all'istruzione `INCLUDE`.

A titolo di esempio, la seguente istruzione inserisce il contenuto del file `MYFILE.DEF` all'interno dell'unità di programma compilata:

```
INCLUDE "MYFILE.DEF"
```

Si noti che il Fortran 90/95 *non* prevede inclusioni *ricorsive*, nel senso che non permette ad un file di includere sé stesso. Sono, tuttavia, consentite inclusioni innestate, fino ad un massimo stabilito dallo specifico compilatore.

1.1.5 I componenti di una istruzione Fortran

Ciascuna unità di programma Fortran si compone di *istruzioni* le cui componenti sono, essenzialmente, *espressioni* ed *assegnazioni*. Ogni espressione può, a sua volta, essere scomposta in unità più elementari che sono *valori* ed *identificatori*, combinati tra loro tramite specifici *operatori*.

Un identificatore può riferirsi ad una delle seguenti entità:

- Una *costante*, che rappresenta un valore che non cambia durante l'esecuzione del programma.
- Una *variabile*, il cui valore è soggetto a cambiare durante il run.
- Un sottoprogramma *subroutine* o *function* oppure un'unità *modulo*.

Ciascun identificatore è caratterizzato da un suo proprio *nome*. In molti casi questi nomi possono essere scelti dal programmatore. La scelta del nome di un identificatore è soggetta ad alcune regole:

- Un nome rappresenta una stringa di caratteri alfanumerici ma può contenere anche il carattere "underscore".
- Ha una lunghezza massima di 31 caratteri.
- Il primo carattere del nome *deve* essere un carattere alfabetico.
- Lettere maiuscole e minuscole sono perfettamente equivalenti.

Dunque, le seguenti stringhe rappresentano nomi validi di un vocabolario Fortran:

```
x
n1
y_33
```

Viceversa, *non* sono validi i seguenti nomi:

```
_x
1n
abc:2
con st
nome_di_variabile_eccessivamente_lungo
```

in quanto i primi due cominciano con un carattere non alfabetico, il terzo contiene il carattere illegale ":", il quarto contiene un carattere *blank*, l'ultimo nome è eccessivamente lungo essendo composto da 38 caratteri.

1.1.6 Struttura di un programma Fortran

Ciascun programma Fortran è formata da un insieme di istruzioni eseguibili e non eseguibili che devono susseguirsi secondo un ordine ben preciso. In particolare, quale che sia la sua "complessità", una qualsiasi unità di programma è sempre divisa in tre sezioni:

1. La *sezione dichiarativa*: si trova all'inizio dell'unità e contiene una serie di istruzioni *non eseguibili* che definiscono il nome e il tipo dell'unità di programma ed il numero ed il nome degli oggetti (variabili e costanti) utilizzati nell'unità.
2. La *sezione esecutiva*: consiste in una serie di istruzioni *eseguibili* che descrivono le azioni svolte dal programma.
3. La *sezione conclusiva*: contiene una o più istruzioni che interrompono l'esecuzione dell'unità di programma.

A chiarimento di ciò si riporta un semplicissimo esempio di programma Fortran:

```
PROGRAM prova

! Scopo: leggere due numeri interi, moltiplicarli tra loro
! e stampare il valore del prodotto

! Sezione Dichiarativa
  IMPLICIT NONE
  INTEGER :: i, j, k    ! Tutte e tre le variabili sono numeri interi

! Sezione esecutiva
! Lettura dei valori da moltiplicare
  READ(*,*) i,j
! Moltiplicazione di i e j
  k = i*j
! Visualizzazione del risultato
  WRITE(*,*) "Risultato: ", k

! Sezione Conclusiva
  STOP
END
```

Ricapitolando, il linguaggio Fortran prevede alcune regole imprescindibili circa l'ordine in cui devono susseguirsi le istruzioni di un programma o di una procedura:

1. Ciascuna unità di programma deve cominciare con una istruzione di intestazione (*heading*). A seconda dei casi questa intestazione può essere `PROGRAM`, `SUBROUTINE`, `FUNCTION` o `MODULE`.

2. Tutte le istruzioni di dichiarazione devono *precedere* la *prima* istruzione eseguibile. Sebbene non sia strettamente necessario, è spesso consigliabile, per ragioni di chiarezza, raggruppare anche tutte le istruzioni `FORMAT`.

3. Le istruzioni eseguibili devono essere ordinate secondo la logica dell'algoritmo alla base del programma.

4. Ciascuna unità di programma deve terminare con una istruzione `END`.

1.2 Costanti e variabili

Una **costante** è un tipo di dato che viene definito prima che un programma sia eseguito ed il suo valore non cambia durante l'esecuzione del programma. Quando il compilatore incontra una costante, ne registra il valore in una locazione di memoria e fa riferimento a questa locazione tutte le volte che la costante viene utilizzata dal programma.

Una **variabile** è, invece, un tipo di dato il cui valore *può* cambiare una o più volte durante l'esecuzione del programma. E' possibile assegnare un valore iniziale a una variabile prima di

eseguire il programma (*inizializzazione della variabile*). Anche le variabili, così come le costanti, sono nomi scelti dall'utente al fine di riferirsi in maniera simbolica a specifiche locazioni di memoria ed ai dati in essi contenuti.

Ogni entità variabile o costante deve avere un nome *unico*; detto nome può essere formato da un massimo di 31 caratteri alfanumerici, compreso il simbolo di *underscore* ("_"), tuttavia il *primo* carattere deve necessariamente essere un carattere alfabetico.

Nomi validi di variabili sono:

```
a
x_0
variabile
var1
nome_molto_lungo
```

Si noti che il linguaggio Fortran, eccetto che per gli oggetti *stringa di caratteri*, *non* è *case sensitive* per cui, ad esempio, `a` e `A` vengono trattati come identificatori dello *stesso* oggetto.

1.3 Istruzioni di dichiarazione di tipo

Per poter usare una variabile o una costante con nome, è necessario dichiararne *esplicitamente* il *tipo* oppure accettarne il tipo *implicito* per *quel nome*. In fase di dichiarazione è, inoltre, possibile specificare altre proprietà per gli oggetti dichiarati.

Una *istruzione di dichiarazione di tipo* ha la forma generale:

tipo [[, *attributi*]::] *lista_di_oggetti*

in cui *tipo* è uno specificatore di tipo di dati (sia esso *intrinseco* o *derivato*), *attributi* è un eventuale insieme di attributi che possono essere assegnati agli oggetti dichiarati, *lista di oggetti* è una lista di nomi di oggetti (variabili o costanti) e nomi di funzioni.

La parola chiave *tipo* può essere una delle seguenti:

```
INTEGER[selettore_di_kind]
REAL[selettore_di_kind]
COMPLEX[selettore_di_kind]
CHARACTER[[lunghezza][selettore_di_kind]]
LOGICAL[selettore_di_kind]
TYPE(nome_di_tipo_derivato)
```

mentre possibili attributi sono:

```
ALLOCATABLE
DIMENSION
EXTERNAL
INTENT
INTRINSIC
```

```
OPTIONAL
PARAMETER
POINTER
PRIVATE
PUBLIC
SAVE
TARGET
```

E' anche possibile dichiarare un attributo *separatamente* come *istruzione*. In ogni caso è possibile specificare ogni attributo soltanto una volta per uno stesso oggetto.

Il simbolo dei due punti ("`::`") è obbligatorio soltanto se l'istruzione di dichiarazione include una lista di attributi oppure se presenta un'espressione di inizializzazione, in tutti gli altri casi il suo uso è opzionale.

Esempi di istruzioni di dichiarazione di tipo sono le seguenti:

```
REAL :: a
LOGICAL, DIMENSION (5,5) :: mask1, mask2
COMPLEX :: cube_root=(-0.5,0.867)
INTEGER, PARAMETER :: short=SELECTED_INT_KIND(4)
REAL(KIND(0.0D0)) :: a1
REAL(KIND=2) :: b=3
COMPLEX(KIND=KIND(0.0D0)) :: c
INTEGER(short) :: i, j, k
TYPE(member) :: george
```

Dunque, una dichiarazione esplicita può specificare degli attributi per gli oggetti della lista, oltre che il loro tipo. Questi attributi descrivono la natura della variabile e in che modo essa sarà usata. In effetti, una istruzione di dichiarazione può essere o *orientata alle entità* (*entity-oriented*) oppure *orientata agli attributi* (*attribute-oriented*). Nel primo caso il tipo di un oggetto e *tutti* i suoi attributi sono specificati in un'*unica* istruzione, nel secondo caso ogni attributo è specificato in una istruzione separata che elenca tutti gli oggetti aventi quell'attributo.

Prima di proseguire nello studio degli elementi fondamentali del linguaggio, è bene chiarire la differenza fra **dichiarazione** e **definizione** di un oggetto:

- Una *dichiarazione* specifica il *tipo*, gli *attributi* ed eventuali altre *proprietà* di un oggetto (tutte quelle elencate in precedenza erano istruzioni di dichiarazione).
- Una *definizione* fornisce il *valore* dell'oggetto.

La seguente istruzione è un ulteriore esempio di istruzione di *dichiarazione esplicita* di tipo:

```
REAL :: x
```

La variabile `x` così dichiarata può essere *definita* in una istruzione di assegnazione come questa:

```
x = 25.44
```

1.3.1 Costanti con nome

Per assegnare un nome ad una costante basta includere l'*attributo* `PARAMETER` nell'istruzione di dichiarazione di tipo:

```
tipo, PARAMETER ::  nome = valore [, nome_2 = valore_2, ...]
```

dove *`tipo`* rappresenta il tipo di appartenenza della costante mentre *`nome`* è il nome con cui ci si riferirà alla costante *`valore`*. Ad esempio, la seguente istruzione assegna il nome `pi` alla costante reale `3.14` e il nome `e` alla costante reale `2.72`:

```
REAL, PARAMETER :: pi=3.14, e=2.72
```

Si noti, in particolare, che se la costante è una *stringa di caratteri* allora *non* è necessario dichiararne la lunghezza, così come esemplificato dalla seguente istruzione:

```
CHARACTER, PARAMETER :: frase="converge"
```

in quanto questa informazione viene automaticamente mutuata dalla lunghezza del lato destro dell'assegnazione.

Oltre all'*attributo* `PARAMETER` il Fortran 90/95 mette a disposizione anche l'equivalente *istruzione* `PARAMETER` la cui sintassi è la seguente:

```
PARAMETER (nome = valore [, nome_2 = valore_2, ...])
```

Un esempio di applicazione dell'istruzione `PARAMETER` è il seguente:

```
INTEGER :: l1, l2, l3
REAL :: x
CHARACTER :: str
PARAMETER(l1=10,l2=20,l3=30,x=1.234,str="finito")
```

che è un esempio di dichiarazione *attribute-oriented*, del tutto equivalente alla seguente possibilità:

```
INTEGER, PARAMETER :: l1=10, l2=20, l3=30
REAL, PARAMETER :: x=1.234
CHARACTER(LEN=6), PARAMETER :: str="finito"
```

che è, invece, *entity-oriented*. Si tratta, comunque, di un retaggio del Fortran 77 qui inserito solo per motivi di completezza ed il cui uso comunque si sconsiglia.

1.3.2 Dichiarazioni esplicite ed implicite

Come visto in precedenza, esempi di istruzioni di dichiarazione esplicita di tipo (anche dette *istruzioni di specificazione di tipo*) sono:

```
REAL :: a, b, val_reale
INTEGER :: i, j, val_intero
COMPLEX :: onda
CHARACTER(LEN=20) :: nome, cognome
LOGICAL :: converge
```

Il Fortran mette a disposizione anche un meccanismo di dichiarazione *implicita* secondo cui se una variabile, una costante o una funzione (*non* intrinseca), *non* è stata esplicitamente dichiarata, allora il suo tipo è mutuato implicitamente dalla *prima* lettera del suo nome secondo la seguente regola:

- Gli oggetti aventi un nome che comincia con le lettere che vanno da "`i`" a "`n`" sono considerati di tipo `INTEGER`.
- Gli oggetti aventi un nome che comincia con le lettere che vanno da "`a`" a "`h`" e da "`o`" a "`z`" sono considerati di tipo `REAL`.

Una specificazione esplicita evita questo meccanismo implicito di dichiarazione. Un altro modo, più sicuro ed elegante, di inibire una dichiarazione implicita degli oggetti è quella di usare l'istruzione:

```
IMPLICIT NONE
```

Questa istruzione *impone* una dichiarazione *esplicita* di ciascuna variabile, costante con nome o funzione definita dall'utente, usate in un programma.

L'uso dell'istruzione `IMPLICIT NONE` è altamente raccomandato in quanto offre non pochi vantaggi. Innanzitutto, esso permette di catturare gran parte degli errori ortografici in fase di compilazione, prima, cioé, che possano verificarsi degli errori difficili da scoprire durante l'esecuzione del programma. Ed inoltre, imponendo al programmatore di includere una lista completa di tutte le variabili nella sezione dichiarativa del programma, semplifica di molto la manutenzione del codice. Se, infatti, un programma deve essere modificato, un controllo della lista specificata nella sezione dichiarativa permette di evitare al programmatore di utilizzare per errore nomi di variabili già definiti.

Si noti che l'istruzione `IMPLICIT NONE` deve precedere ogni istruzione di un programma fatta eccezione per le istruzioni `USE` e `FORMAT` e, chiaramente, per le righe di commento le quali, non essendo compilate, possono essere inserite in un punto qualsiasi del programma. Inoltre, tale istruzione ha effetto soltanto sull'*unità di programma* nella quale viene applicata per cui, in programmi che si articolano in più unità, l'istruzione `IMPLICIT NONE` deve essere specificata in ognuna di esse.

1.3.3 Oggetti di tipo `INTEGER`

Una entità di tipo `INTEGER` è un oggetto atto a contenere un valore appartenente ad un subset dei numeri *interi*. L'istruzione `INTEGER` serve a specificare che una data variabile (o funzione) è di tipo intero. La sintassi dell'istruzione di dichiarazione di un oggetto di tipo intero è:

```
INTEGER [[([KIND=]parametro_di_kind)][, attributi]::]  lista_di_oggetti
```

dove `INTEGER` e `KIND` sono *parole chiave* del Fortran e *`lista_di_oggetti`* è una lista di nomi di variabili, costanti o funzioni (separati da virgole) che saranno utilizzate nella fase esecutiva del programma per immagazzinare dati di tipo intero. Se la specifica del *`parametro_di_kind`* è assente viene assunto il valore del parametro di kind di default.

Ad esempio, la seguente istruzione:

```
INTEGER :: i, j, numero_intero
```

dichiara tre variabili, `i`, `j` e `numero_intero`, di tipo intero. Dichiarazioni un pò più "complesse" sono, invece:

```
INTEGER, DIMENSION(10), POINTER :: giorni, ore
INTEGER(KIND=2), SAVE :: k, limite
```

che rappresentano degli esempi di dichiarazioni *entity-oriented*, mentre le corrispondenti versioni *attribute-oriented* sono:

```
INTEGER :: giorni, ore
INTEGER(KIND=2) :: k, limite
POINTER :: giorni, ore
DIMENSION giorni(10), ore(10)
SAVE :: k, limite
```

Una *costante* intera è rappresentata da un segno "`+`" (opzionale) o da un segno "`-`" che precede un allineamento di cifre decimali, seguito da uno specificatore di kind opzionale. Le costanti intere sono interpretate come valori decimali (ossia in base 10) per default. Esempi di costanti intere con e senza segno sono:

```
123   +123   -123   12_2   -1234567890_4   12_short
```

dove `short` è una *costante con nome* precedentemente definita avente come valore un numero di kind valido.

1.3.4 Oggetti di tipo `REAL`

Il tipo `REAL` approssima gli elementi di un subset dell'insieme dei numeri reali usando due metodi di approssimazione che sono comunemente noti come *singola precisione* e *doppia precisione*. Questi numeri sono anche detti numeri in *virgola mobile* (*floating-point*). L'istruzione `REAL` serve a specificare che un dato oggetto è di tipo reale. La sintassi dell'istruzione di dichiarazione di un oggetto di tipo `REAL` è:

```
REAL [[([KIND=]parametro_di_kind)][, attributi] ::]  lista_di_oggetti
```

dove `REAL` è una *parola chiave* del Fortran e *`lista_di_oggetti`* è una lista di nomi di variabili, costanti o funzioni (separati da virgole) utilizzate nella fase esecutiva del programma per immagazzinare dati di tipo reale. Se la specifica del *`parametro_di_kind`* è assente verrà assunto il valore del parametro di kind di default.

Ad esempio, la seguente istruzione:

```
REAL :: x, y, variabile_reale
```

dichiara tre variabili, `x`, `y` e `variabile_reale`, di tipo reale. Ulteriori esempi, più articolati, di dichiarazioni *entity-oriented* sono:

```
REAL(KIND=high), OPTIONAL :: testval
REAL, SAVE, DIMENSION(10) :: b
```

mentre le stesse dichiarazioni in formato *attribute-oriented* hanno la forma:

```
REAL(KIND=high) :: testval
REAL :: b
DIMENSION b(10)
SAVE  b
OPTIONAL testval
```

Una costante reale approssima il valore di un numero del sottoinsieme reale rappresentabile in macchina. Le due possibili rappresentazioni di una costante reale possono essere *senza* parte esponenziale:

`[`*s*`]`*n*`[`*n*`...][_`*k*`]`

oppure *con* parte esponenziale:

`[`*s*`]`*n*`[`*n*`...]E[`*s*`]`*n*`[`*n*`...][_`*k*`]`
`[`*s*`]`*n*`[`*n*`...]D[`*s*`]`*n*`[`*n*`...]`

in cui il significato dei simboli è il seguente:

- *s* segno della costante; può essere "+" (opzionale) o "-" (obbligatorio).
- *n* cifra decimale (da `0` a `9`).
- *k* parametro di kind (opzionale).

Il punto decimale *deve* essere presente se la costante *non* ha parte esponenziale.

L'esponente `E` denota una costante reale in *singola precisione*, a meno che l'eventuale parametro di kind non specifichi diversamente. L'esponente `D` denota, invece, una costante reale in *doppia precisione*. Entrambi gli esponenti rappresentano potenze di `10` (ad esempio `1.0E+5` rappresenta il numero 1.0×10^5).

Esempi di costanti reali valide sono:

```
3.14159   3.14159_high   -6234.14_4   -.1234   +5.01E3   -2D-3   4.3E2_8
```

Una costante di tipo `REAL` viene registrata in memoria in due parti: la *mantissa* e l'*esponente*. Il numero di bit allocati per la mantissa determina la *precisione* della costante (ossia il numero di cifre significative), mentre il numero di bit allocati per l'esponente determina il *range* della costante (ossia l'intervallo dei valori che possono essere rappresentati).

1.3.5 Oggetti di tipo `COMPLEX`

Una entità di tipo complesso è un oggetto atto ad ospitare un valore appartenente ad un sottoinsieme dei numeri complessi. Esso si compone di due parti: una *parte reale* ed una *parte immaginaria*. Il tipo complesso, quindi, immagazzina un valore complesso in due unità di memoria consecutive atte a contenere, ciascuna, un valore reale, il primo rappresentante la parte reale, il secondo la parte immaginaria del valore complesso.

La sintassi di una istruzione di dichiarazione di oggetti di tipo complesso ha la seguente forma:

```
COMPLEX [[([KIND=]parametro di kind)][, attributi] ::]  lista_di_oggetti
```

Ad esempio, la seguente istruzione dichiara due variabili complesse `v` e `w` aventi parametro di kind pari a `3`.

```
COMPLEX(KIND=3) :: v, w
```

Si noti, tuttavia, che dal momento che un numero complesso rappresenta un'entità "composta" consistente in una coppia di numeri *reali*, il valore del selettore di kind, sia che esso venga specificato in maniera implicita sia che venga espresso in forma esplicita, è applicato a *entrambi* i componenti. Pertanto, le due variabili complesse `v` e `w` precedentemente definite saranno caratterizzate entrambe da una coppia di componenti (la parte reale e la parte immaginaria) reali aventi numero di kind pari a tre. Un ulteriore esempio di dichiarazione (*entity-oriented*) è:

```
COMPLEX(KIND=4), DIMENSION(8) :: cz, cq
```

mentre la sua "versione" *attribute-oriented* è:

```
COMPLEX(KIND=4) :: cz, cq
DIMENSION(8) cz, cq
```

Una costante complessa approssima il valore di un numero complesso pertanto essa è scritta come una coppia di costanti numeriche separate da una virgola e racchiuse tra parentesi. La prima di tali costanti rappresenta la parte reale del valore complesso, la seconda la parte immaginaria. La forma generale di una costante complessa è la seguente:

```
(parte_reale, parte_immaginaria)
```

in cui *parte_reale* e *parte_immaginaria* sono costanti di tipo `REAL`.

Esempi di costanti complesse sono le seguenti:

```
(1.5,7.3)
(1.59E4,-12E-1)
(2.5,6.)
(19.0,2.5E+3)
```

1.3.6 Oggetti di tipo CHARACTER

Una entità di tipo `CHARACTER` è un oggetto atto a contenere un carattere appartenente ad un opportuno insieme, o una "stringa" di tali caratteri. Lo specificatore di tipo è la parola chiave `CHARACTER` e la sintassi dell'istruzione di dichiarazione è la seguente:

`CHARACTER [[LEN=`*`lunghezza`*`][([KIND=]`*`kind`*`)][,` *`attributi`*`] ::]` *`lista`*

dove *`lunghezza`* è un valore intero che specifica il *massimo numero di caratteri* che può essere immagazzinato negli oggetti elencati nella *`lista`*, essendo quest'ultima un elenco di nomi di oggetti (separati da virgole) che saranno utilizzati nella fase esecutiva del programma per immagazzinare dati di tipo carattere.

Ad esempio, le seguenti istruzioni:

```
CHARACTER(LEN=1) :: ch, carattere
CHARACTER(LEN=10) :: str, stringa
```

dichiarano quattro variabili di tipo carattere, di cui le prime due, `ch` e `carattere`, atte ad ospitare un unico carattere, le altre due, `str` e `stringa`, atte a contenere ciascuna una stringa di (al massimo) dieci caratteri.

Una costante di tipo carattere è una sequenza di caratteri racchiusa fra apici (') o doppi apici () che ne rappresentano i *delimitatori*. Una costante di caratteri si può presentare in una delle seguenti forme:

```
'[k_][c...]'
[k_][c...]
```

in cui *`k`* è l'eventuale parametro di kind, mentre *`c`* è un carattere previsto dal sistema di codifica adottato dal processore (ad esempio l'insieme ASCII).

Per rappresentare un apice all'interno di una stringa delimitata da apici è necessario usare *due* apici consecutivi così come per rappresentare un doppio apice all'interno di una stringa delimitata da doppi apici è necessario usare *due* doppi apici consecutivi.

Una stringa di caratteri di *lunghezza nulla* è specificata da due apici (o doppi apici) consecutivi (ossia *non* separati da alcuno spazio bianco).

La *lunghezza* di una *costante stringa* è pari al *numero di caratteri* inseriti fra i delimitatori. A tale proposito si noti che una coppia di apici in una costante stringa delimitata da apici conta come un unico carattere e che una coppia di doppi apici all'interno di una stringa delimitata da doppi apici conta come un unico carattere.

Di seguito sono riportati alcuni esempi di stringhe di caratteri:

Stringa	*Valore*
`'stringa'`	`stringa`
`'1234'`	`1234`
`'l''aereo'`	`l'aereo`
`l'aereo`	`l'aereo`
`Le Stringhe Sono Case Sensitive`	`Le Stringhe Sono Case Sensitive`
`Una coppia di ''doppi'' apici`	`Una coppia di doppi apici`

Se una costante di caratteri eccede la lunghezza utile per una riga di comando (132 colonne), il suo valore deve essere spezzato su più linee, come nel seguente esempio:

```
stringa = "Questa è una delle stringhe più lunghe che &
           &mi viene in mente in questo momento ma ne  &
           &esisteranno sicuramente di molto più lunghe!"
```

C'è un aspetto importante che riuguarda le costanti con nome di tipo `CHARACTER`, ossia il fatto che la sua lunghezza può essere specificata da un asterisco ("*"), essendo in tal modo mutuata direttamente dal suo valore. In tal modo si ovvia alla necessità di dover contare i caratteri della stringa il che rende più semplice modificarne successivamente la definizione. L'istruzione che segue sfrutta proprio questa caratteristica:

```
CHARACTER(LEN=*), PARAMETER :: str="Nessun bisogno di contare i caratteri"
```

Tuttavia questa caratteristica non può essere sfruttata quando un array di caratteri è definito a mezzo di un costruttore di array, dal momento che gli elementi devono essere della stessa lunghezza:

```
CHARACTER(LEN=7), DIMENSION(2), PARAMETER :: aaa=(/"Antonio","Auletta"/)
```

1.3.7 Oggetti di tipo `LOGICAL`

L'istruzione `LOGICAL` specifica che il nome di una data variabile o di una data funzione verrà utilizzato per riferirsi ad un oggetto di tipo "booleano". La sua sintassi è la seguente:

`LOGICAL [[([KIND=]`*parametro di kind*`)][, `*attributi*`] ::]` *lista_di_oggetti*

Esempi di dichiarazioni (*entity-oriented*) di variabili logiche sono:

```
LOGICAL, ALLOCATABLE, DIMENSION(:) :: flag1, flag2
LOGICAL(KIND=byte), SAVE :: doit, dont
```

Le stesse dichiarazioni, in una forma *orientata agli attributi*, appaiono invece nel seguente modo:

```
LOGICAL :: flag1, flag2
LOGICAL(KIND=byte) :: doit, dont
ALLOCATABLE flag1(:), flag2(:)
SAVE doit, dont
```

Una costante di tipo `LOGICAL` può rappresentare unicamente i valori logici "vero" e "falso" e assume una delle seguenti forme:

`.TRUE.[_`*k*`]`
`.FALSE.[_`*k*`]`

in cui *k* è il parametro di kind opzionale. Nel caso in cui il parametro di kind non è esplicitamente dichiarato per esso viene assunto un valore di default.

Le variabili logiche rappresentano raramente il risultato di un programma Fortran. Tuttavia, esse svolgono un ruolo essenziale per il corretto funzionamento di molti programmi. La maggior parte delle *strutture di controllo* dei programmi, infatti, sono governate dai valori assunti da variabili o da espressioni logiche.

1.4 Inizializzazione delle variabili

Fintanto che ad una variabile dichiarata non venga assegnato un valore, tale variabile è detta *non inizializzata*; il valore di una tale variabile *non* è definito dallo standard del linguaggio. Alcuni compilatori impostano automaticamente a zero il valore delle variabili non inizializzate, altri lasciano in memoria i valori che preesistevano nelle locazioni di memoria attualmente occupati dalle variabili, altri ancora generano addirittura un errore al tempo di esecuzione quando una variabile venga utilizzata senza essere inizializzata.

Dal momento che macchine differenti possono gestire le variabili non inizializzate in maniera differente, un programma che funzioni su una macchina potrebbe non funzionare su un'altra macchina (a parte l'evenienza che su una stessa macchina potrebbe funzionare più o meno correttamente a seconda dei dati che i precedenti programmi hanno lasciato nelle locazioni di memoria delle variabili). Per evitare i rischi connessi a tali evenienze, è sempre consigliabile inizializzare tutte le variabili utilizzate in un programma.

Per inizializzare le variabili è possibile adottare una delle seguenti tecniche:

- Inizializzazione mediante un'*istruzione di assegnazione*.
- Inizializzazione mediante un'*istruzione di lettura*.
- Inizializzazione mediante l'*istruzione di dichiarazione di tipo*.

Il seguente programma esemplifica le prime due tecniche:

```
PROGRAM iniz_1
  INTEGER :: i, j
  i = 1          ! inizializzazione tramite assegnazione
  READ*, j       ! inizializzazione tramite istruzione di lettura
  PRINT*, i,j
END PROGRAM
```

La terza tecnica di inizializzazione consiste nello specificare i valori iniziali nell'istruzione di dichiarazione di tipo, e pertanto specifica il valore della variabile già in fase di compilazione contrariamente all'inizializzazione con un'istruzione di assegnazione che, invece, ha effetto soltanto in fase di esecuzione.

La forma generale di un'istruzione di dichiarazione di tipo che inizializzi una variabile è:

```
tipo ::  variabile = valore [, variabile_2 = valore_2, ...]
```

A titolo di esempio, si può considerare il seguente frammento di programma:

```
PROGRAM iniz_2
  INTEGER :: i=1, j=2
  REAL :: x=10
  CHARACTER(LEN=6) :: str="finito"
  LOGICAL :: conv=.FALSE.
  ...
END PROGRAM
```

1.5 Istruzioni di assegnazione ed espressioni

La più semplice fra le istruzioni esecutive del Fortran è l'*istruzione di assegnazione*, la cui sintassi è la seguente:

```
variabile = espressione
```

con il significato che il risultato di *espressione* viene immagazzinato nella locazione di memoria assegnata a *variabile*.

L'esecuzione dell'istruzione produce, nell'ordine:

- il calcolo di *espressione*, il cui risultato viene valutato secondo le regole proprie del tipo dell'espressione;
- la conversione di tipo, se il *tipo* di *espressione* non coincide con quello di *variabile* (questa conversione *automatica* è limitata ai soli casi discussi nel seguito);
- l'immagazzinamento del risultato nella locazione di memoria relativa a *variabile*.

Una espressione Fortran è una qualunque combinazione di operandi ottenuta tramite operatori e parentesi tonde; l'esecuzione delle operazioni indicate dagli operatori dà luogo ad un risultato detto *valore dell'espressione*. Si osservi che anche un singolo operando costituisce una espressione.

Dunque l'istruzione di assegnazione calcola il valore dell'espressione a destra del segno di uguale e *assegna* tale valore alla variabile specificata a sinistra del segno di uguale. Si noti che il simbolo "=" *non* indica uguaglianza secondo il significato comune del termine. Piuttosto un'istruzione di assegnazione va interpretata come: "*registra il valore di* `espressione` *nella locazione di memoria di* `variabile`". E' solo in tal modo che assumono un senso istruzioni di "aggiornamento" del tipo seguente:

```
INTEGER  ::  count=0
count = count+1
```

in cui il significato dell'istruzione di assegnazione è quello di incrementare di un'unità il valore della variabile intera `count` (precedentemente inizializzata a zero) e di immagazzinare il nuovo valore nella locazione di memoria riservata alla stessa variabile `count`.

Si faccia, poi, attenzione al fatto che ogni qualvolta a *variabile* viene assegnato un nuovo valore, il precedente contenuto delle locazioni di memoria corrispondenti a *variabile* viene perduto definitivamente.

Il seguente blocco di istruzioni fornisce un ulteriore, semplice, esempio di istruzioni di assegnazione. Esso ha lo scopo di "invertire" il contenuto delle variabili `a` e `b` facendo uso della variabile "di appoggio" `c`:

```
INTEGER :: a=3, b=5, c
c = a
a = b
b = c
```

Una cosa importante da tenere a mente è che un qualsiasi nome dichiarato con l'attributo `PARAMETER` (ossia, una qualunque costante con nome) è semplicemente un *alias* di un valore ben preciso, e non una variabile, pertanto esso non può in alcun modo essere usato a sinistra di un'istruzione di assegnazione. Nel seguente frammento di programma, pertanto, l'istruzione di assegnazione provoca un messaggio di errore in fase di compilazione:

```
INTEGER, PARAMETER  :: x1=2.54, x2=123.45
x1=3.0*x2          ! Errore: x1 è una costante con nome
```

Qualora occorra assegnare ad una variabile un valore risultante dal calcolo di una espressione di tipo diverso, è necessario ricorrere esplicitamente alla conversione di tipo, secondo lo schema:

```
variabile = CONV(espressione)
```

oppure:

```
variabile_di_appoggio = espressione
variabile = CONV(variabile_di_appoggio)
```

in cui *variabile_di_appoggio* deve avere lo stesso tipo di *espressione* e *CONV* è un nome fittizio per indicare una *funzione di conversione* verso il tipo di *variabile*.

Nel seguito si tratteranno separatamente le specificità delle regole di assegnazione e delle operazioni di cui si compongono le espressioni nei diversi tipi. Prima di proseguire, però, è utile premettere alcune definizioni.

Si chiama *espressione costante* una qualunque espressione in cui ciascuna operazione è di tipo *intrinseco* e ciascun operando fa parte del seguente gruppo (si noti che alcuni dei punti seguenti saranno chiari soltanto proseguendo nella lettura di queste note):

- Una costante
- Un *costruttore di array* i cui elementi siano essi stessi espressioni costanti
- Un *costruttore di struttura* i cui componenti siano essi stessi espressioni costanti
- Una chiamata di funzione intrinseca di elemento, i cui argomenti siano espressioni costanti
- Una chiamata di funzione intrinseca di trasformazione, i cui argomenti siano espressioni costanti
- Una chiamata di fuznione di interrogazione (che non sia `PRESENT`, `ASSOCIATED` o `ALLOCATED`) i cui argomenti siano espressioni costanti o variabili le cui carattersitiche interrogate non siano, però, *presunte*, *definite da un'espressione non costante*, *definite da un'istruzione* `ALLOCATE` o *definite da un'assegnazione di puntatore*.
- Una variabile di *ciclo implicito* i cui limiti ed il cui passo siano espressioni costanti
- Una espressione costante racchiusa in parentesi

Dal momento che il valore di una costante con nome deve essere valutato al tempo di compilazione, le espressioni consentite per la sua definizione sono, in qualche modo, limitate. Una *espressione di inizializzazione* è, infatti, una espressione costante in cui:

- L'operatore di elevamento a potenza deve avere un esponente intero
- Una funzione intrinseca di elemento deve avere argomenti e risultato di tipo `INTEGER` o `CHARACTER`
- Delle funzioni di trasformazione, soltanto le procedure `REPEAT`, `RESHAPE`, `TRANSFER`, `TRIM`, `SELECTED_INT_KIND`, `SELECTED_REAL_KIND` sono consentite

Nella definizione di una costante con nome è possibile utilizzare una espressione di inizializzazione e la costante diventa definita con il valore dell'espressione secondo le regole dell'assegnazione intrinseca. Quanto detto è illustrato dall'esempio che segue:

```
! costante con nome di tipo INTEGER
INTEGER, PARAMETER :: length=10, long=SELECTED_REAL_KIND(12)
! costante con nome di tipo REAL
REAL, PARAMETER :: lsq=length**2
! costante con nome di tipo array
REAL, DIMENSION(3), PARAMETER :: array=(/1.0, 2.0, 3.0/)
! costante con nome di tipo di dati defnito dall'utente
TYPE(miotipo), PARAMETER :: a=miotipo(1.0,5,"casa")
```

1.5.1 Espressioni aritmetiche

Tutti i calcoli effettuati da un programma Fortran vengono specificati attraverso *istruzioni di assegnazione* il cui formato generale è il seguente:

variabile = espressione_aritmetica

L'esecuzione di una istruzione di questo tipo avviene calcolando prima il valore di *espressione aritmetica* e poi memorizzando poi tale valore nella cella identificata dal nome *variabile*.

Si noti, tuttavia, che se il tipo dell'espressione *non* coincide con quello della variabile a primo membro, allora il valore dell'espressione viene convertito al modo seguente:

- Se il tipo di *variabile* è `INTEGER` mentre il tipo di *espressione* è `REAL` allora la parte decimale, incluso il punto, viene rimosso dando luogo ad un risultato *intero*.
- Se il tipo di *variabile* è `REAL` mentre il tipo di *espressione* è `INTEGER` allora al valore (intero) del risultato viene aggiunto il punto decimale dando così luogo ad un valore *reale*.

A titolo di esempio si guardi il seguente frammento di programma:

```
INTEGER :: totale, prezzo, numero
numero = 5
prezzo = 100.65
totale  = numero*prezzo
PRINT*, numero,prezzo,totale
```

Le precedenti istruzioni dichiarano, dapprima, tre variabili di tipo `INTEGER`. Quindi, ha luogo la prima istruzione di assegnazione che salva il valore intero `5` nella variabile `numero`. La seconda istruzione salva il valore approssimato `100.65` nella variabile prezzo. Tuttavia, dal momento che `prezzo` è una variabile intera, il valore reale `100.65` viene dapprima *troncato* all'intero `100` e poi immagazzinato nella variabile. Infine, la terza assegnazione calcola il valore dell'espressione `5*100` e ne salva il risultato (`500`) nella variabile `totale`. Pertanto l'istruzione di scrittura fornirà il seguente risultato:

```
 5   100   500
```

Una espressione aritmetica, utilizzata per ottenere come risultato un valore numerico, può contenere qualsiasi combinazione valida di variabili, costanti e operatori aritmetici. Gli operatori aritmetici standard del Fortran sono:

`+`	Addizione
`-`	Sottrazione
`*`	Moltiplicazione
`/`	Divisione
`**`	Elevamento a potenza

I cinque operatori aritmetici appena elencati sono *operatori binari* in quanto agiscono su una coppia di operandi:

```
a + b
a - b
a * b
a / b
a ** b
```

tuttavia, i simboli `+` e `-` possono agire anche come *operatori unari*:

```
-a
+23
```

Il linguaggio adotta una serie di regole per controllare l'ordine (o *gerarchia*) secondo cui devono essere svolte le operazioni matematiche all'interno di un'espressione, regole peraltro conformi a quelle standard dell'algebra:

1. Dapprima vengono svolte le operazioni collocate all'interno delle parentesi, partendo da quelle più interne.

2. Vengono calcolati gli elevamenti a potenza, procedendo da destra verso sinistra.

3. Vengono calcolate le moltiplicazioni e le divisioni, procedendo da sinistra verso destra.

4. Vengono calcolate le addizioni e le sottrazioni, procedendo da sinistra verso destra.

Aritmetica intera, reale e complessa

Se un operatore aritmetico opera su grandezze tutte dello stesso tipo, il risultato dell'operazione è un valore il cui tipo coincide con quello degli operandi; esso viene determinato eseguendo l'operazione in aritmetica intera, reale (singola o doppia precisione) o complessa a seconda del tipo degli operandi. Così, ad esempio, le operazioni che avvengono tra operandi tutti di tipo intero generano sempre un risultato di tipo intero. Ciò è da tenere bene a mente in presenza di operazioni di divisione. Infatti, se la divisione fra due interi *non* è un intero, il computer tronca automaticamente la parte decimale del risultato, come mostrato dai seguenti esempi:

```
3/2  risultato:  1
1/2  risultato:  0
9/4  risultato:  2
```

Allo stesso modo, le operazioni che avvengono tra operandi tutti di tipo reale generano sempre un risultato di tipo reale:

```
3./2.0  risultato:  1.5
1.1/2.  risultato:  0.55
9./4.   risultato:  2.25
```

Tuttavia, anche le operazioni tra reali hanno le loro peculiarità; infatti, dal momento che i valori memorizzati in un calcolatore hanno una precisione limitata (essendo, necessariamente, limitata l'area di memoria ad essi riservata) la rappresentazione di ciascun valore reale è limitata ad un ben preciso numero di cifre decimali (i dettagli relativi a questa approssimazione sono relativi al *parametro di kind* del valore reale e di questo si parlerà diffusamente nel seguito). Quindi, ad esempio, il valore del rapporto reale `1/3`, espresso matematicamente da un allineamento infinito di cifre decimali (nel caso specifico, da un allineamento periodico), potrebbe avere una rappresentazione di macchina limitata a `0.3333333`. Come conseguenza di questa limitazione alcune quantità che sono "teoricamente" uguali potrebbero risultare diverse quando elaborate da un calcolatore. Ad esempio, l'espressione `3.*(1./3.)`, a causa di inevitabili approssimazioni nella rappresentazione dei dati, potrebbe avere un valore diverso da `1`. Si parla, in questi casi, di *uguaglianza approssimata*, per distinguerla dalla *uguaglianza esatta* tipica dell'aritmetica intera.

Allo stesso modo, operazioni aritmetiche tra operandi di tipo complesso danno come risultato un numero complesso. Di seguito si ricordano le definizioni matematiche delle operazioni elementari fra complessi. Con z_1 e z_2 si rappresentano i numeri complessi $u_1 + iv_1$ e $u_2 + iv_2$, rispettivamente:

Operazione	*Risultato*	
	Parte reale	*Parte immaginaria*
$z_1 + z_2$	$u_1 + u_2$	$v_1 + v_2$
$z_1 - z_2$	$u_1 - u_2$	$v_1 + v_2$
$z_1 * z_2$	$u_1u_2 - v_1v_2$	$u_1v_2 + v_1u_2$
z_1/z_2	$\frac{u_1u_2 + v_1v_2}{\lvert z_2\rvert}$	$\frac{u_2v_1 - v_2u_1}{\lvert z_2\rvert}$
$\lvert z_2\rvert$	$\sqrt{u_2^2 + v_2^2}$	0

Il seguente frammento di programma mostra alcuni esempi di espressioni complesse:

```
COMPLEX, PARAMETER :: z1=(5.,3.), z2=(2.5,-1.1)
COMPLEX :: somma, diff, prod, resul
somma = z1+z2          ! ris: (7.5,1.9)
diff = z1-z2           ! ris: (2.5,4.1)
prod = z1*z2           ! ris: (15.8,2.0)
resul = (z1+z2-(5.5,-0.1))/(2.,2.)  ! ris: (1.,0.)
```

Espressioni aritmetiche miste

Tutte le operazioni aritmetiche viste finora erano applicate ad operandi dello stesso tipo ed il tipo del risultato coincideva con quello degli operandi. E' tuttavia possibile utilizzare in una stessa espressione operandi di tipo diverso dando luogo a quelle che vengono dette *operazioni miste*. Le espressioni che contengono una o più operazioni miste si chiamano *espressioni miste*.

Per quanto concerne il tipo del risultato, vale la regola generale secondo cui se il tipo di due operandi non è lo stesso, quello più "debole" viene automaticamente modificato in quello più "forte" mediante una operazione, detta di *conversione*, che permette il passaggio da una rappresentazione interna ad un'altra. Questo processo di conversione è subordinato alla seguente gerarchia fra i tipi degli operandi:

```
COMPLEX    più forte
  REAL          ↓
INTEGER    più debole
```

Un'operazione che coinvolga sia operandi interi che reali è un esempio di operazione mista. Quando un compilatore incontra una operazione del genere, converte il valore dell'operando intero in valore reale sicché l'operazione viene svolta con operandi reali. A titolo di esempio si possono considerare le seguenti espressioni:

`3/2`	espressione intera	risultato: `1` (intero)
`3./2.`	espressione reale	risultato: `1.5` (reale)
`3./2`	espressione mista	risultato: `1.5` (reale)

Tuttavia, la conversione *automatica* viene effettuata soltanto se l'operando intero e quello reale compaiono nella *stessa* operazione. Di conseguenza, una parte dell'espressione potrebbe essere valutata come una espressione intera e una parte come espressione reale. Ad esempio, la seguente espressione:

```
1/2 + 1./4.
```

fornisce il valore `0.25` (che coincide con il valore del secondo argomento della somma) in quanto la prima operazione, avvenendo fra operandi interi, restituisce un valore intero, nella fattispecie il valore `0`. Se, invece, la prima operazione fosse stata di natura mista, come ad esempio:

```
1./2 + 1./4.
```

avrebbe avuto luogo una conversione automatica tale che il valore (reale) della prima operazione sarebbe stato `0.5` e, dunque, il valore dell'intera espressione sarebbe stato `0.75`, ben diverso da quello fornito dall'espressione precedente.

Sia valori interi che valori reali possono essere combinati con dati di tipo complesso a formare espressioni miste. Tali espressioni vengono valutate convertendo i dati interi e reali a formare dati complessi con parte immaginaria nulla. Così, ad esempio, se `z1` è il numero complesso `(x1,y1)`, ed `r` è un numero reale, l'espressione

```
r * z1
```

viene convertita in:

```
(r,0)*(x1,y1)
```

che verrà valutata come:

```
(r*x1,r*y1)
```

Allo stesso modo, se `n` è un intero, allora l'espressione:

```
n + z1
```

viene convertita in:

```
(REAL(n),0)+(x1,y1)
```

che verrà valutata come:

```
(REAL(n)+x1,y1)
```

Elevamento a potenza

Come regola generale, le espressioni miste dovrebbero sempre essere evitate in quanto subdole e, spesso, non facili da trattare. L'unica eccezione a questa regola è rappresentata dall'elevamento a potenza caratterizzata da una base reale e da un esponente intero. In effetti questa operazione, contrariamente a quanto potrebbe sembrare, *non* è di tipo misto; di ciò ci si può facilmente rendere conto considerando la seguente espressione:

```
INTEGER :: esponente
REAL :: base, risultato
risultato = base**esponente
```

Per calcolare l'espressione `base**esponente` il computer moltiplica il valore di `base` per sé stesso un certo numero di volte (pari al valore di `esponente`). Dunque, l'intera operazione si svolge soltanto fra numeri reali ed è, pertanto, una espressione reale.

Tuttavia è possibile elevare un numero reale anche a potenza reale, come nel seguente esempio:

```
REAL :: x, y, risultato
risultato = y**x
```

Tipicamente in questi casi, per il calcolo del risultato, viene applicata una tecnica indiretta basata sulla seguente uguaglianza:

$$y^x = e^{x \ln y}$$

per cui il valore di `y**x` viene valutato calcolando prima il logaritmo naturale di `y`, quindi moltiplicandolo per `x` e poi elevando e alla potenza del risultato ottenuto. Da quanto detto emerge che l'elevamento a potenza reale deve essere utilizzato soltanto nei casi in cui risulti strettamente necessario in quanto questa procedura risulta più lenta e meno precisa di una serie di moltiplicazioni ordinarie.

Si noti, infine, che mentre è possibile elevare a potenza intera un numero reale negativo, *non* è possibile elevare questo a potenza reale in quanto il logaritmo naturale di un numero negativo non è definito. Pertanto, mentre l'espressione:

```
(-2)**2
```

è perfettamente lecita, l'espressione

```
(-2)**2.0
```

genererebbe, invece, un errore al tempo di esecuzione.

Uso delle funzioni intrinseche

Oltre alle operazioni aritmetiche fondamentali, sono spesso necessarie operazioni più complesse quali, ad esempio, il calcolo dei logaritmi o di funzioni trigonometriche. Tali operazioni vengono eseguite in Fortran usando opportune procedure dette *funzioni intrinseche*. Di queste si parlerà nel dettaglio alla fine del capitolo. Per il momento ci si limiterà a fornire alcune informazioni di carattere generale.

Ciascuna funzione intrinseca opera su uno o più dati, detti *argomenti*, e fornisce un unico risultato che può essere utilizzato scrivendo il nome della funzione seguito da una lista di argomenti racchiusa tra parentesi. Nella maggior parte dei casi è previsto un solo argomento; quando, però, gli argomenti sono più di uno, essi devono essere separati da virgole.

Per utilizzare correttamente una qualunque funzione intrinseca è fondamentale che il numero ed il tipo degli argomenti coincida con quello previsto per la funzione. Si osservi che un argomento di tipo sbagliato *non* viene convertito automaticamente ad un tipo corretto e che, inoltre, errori di questo tipo *non* vengono di solito segnalati dal compilatore e possono generare

risultati imprevedibili. A titolo di esempio, l'espressione `SQRT(2)` è formalmente scorretta in quanto la funzione *radice quadrata*, `SQRT`, non può essere usata con argomenti di tipo intero.

Le seguenti espressioni costituiscono esempi di impiego di funzioni intrinseche:

Espressione	*Significato*
`SQRT(678.36)`	$\sqrt{678.36}$
`EXP(y)`	e^y
`a+SIN(0.5*teta)`	$a + \sin\frac{1}{2}\vartheta$
`0.5*(b+a/SINH(x))+LOG10(3)`	$\frac{1}{2}(b + \frac{a}{\sinh x}) + \log 3$
`b/(ABS(x-y)*eps)`	$b/\lvert x - y\rvert\varepsilon$
`EXP(ABS(h/2.))`	$e^{\lvert h/2\rvert}$
`SQRT(b*c+SQRT(a))`	$\sqrt{bc + \sqrt{a}}$

Come si vede da questi esempi, una qualunque espressione può essere usata come argomento di una funzione intrinseca: in particolare essa può ridursi ad una semplice costante, ad una variabile, oppure può coinvolgere operazioni aritmetiche e/o riferimenti ad altre funzioni intrinseche. E', inoltre, permesso utilizzare come argomento di una funzione intrinseca la funzione stessa.

Se in un'espressione aritmetica compare un riferimento ad una funzione intrinseca, questa viene valutata prima di effettuare qualunque altra operazione. Da ciò segue che le espressioni che compaiono come argomenti di una funzione saranno le prime ad essere valutate e, se esse utilizzano a loro volta una funzione, questa avrà la precedenza.

Il seguente programma mostra un semplice esempio di utilizzo di alcune delle più comuni funzioni intrinseche matematiche: `ATAN2` (per la valutazione della funzione arcotangente) e `SQRT` (per il calcolo della radice quadrata). Il suo scopo è quello di convertire coordinate cartesiane in coordinate sferiche:

```
PROGRAM globo
! Fornisce le coordinate sferiche di un punto note le sue coordinate
! cartesiane e verifica il risultato
   IMPLICIT NONE
   REAL :: r, theta, phi, x, y, z
!  start
   WRITE(*,*) " Inserisci x, y e z: "
   READ(*,*) x,y,z
   r = SQRT(x**2+y**2+z**2)
   theta = ATAN2(y,x)
   phi = ATAN2(SQRT(x**2+y**2),z)
   WRITE(*,*) " Coordinate sferiche: "
   WRITE(*,*) r, theta, phi
   x = r*COS(theta)*SIN(phi)
   y = r*SIN(theta)*SIN(phi)
   z = r*COS(phi)
   WRITE(*,*) " Coordinate cartesiane corrispondenti: "
   WRITE(*,*) x,y,z
```

```
   STOP
END PROGRAM globo
```

Si osservi che alcune funzioni intrinseche permettono di eseguire operazioni di conversione di tipo e possono essere usate per modificare il tipo del risultato dell'espressione che costituisce il loro argomento. Così, ad esempio, la funzione intrinseca `REAL` permette di convertire in reale un valore di tipo intero o di estrarre la parte reale di un valore complesso. Il suo utilizzo può essere esemplificato dalle seguenti espressioni:

Espressione	*Significato*
`REAL(2+1)/2`	`1.5`
`REAL(1/2)*a`	`0.`
`REAL((3.1,-4.0))`	`3.1`

Allo stesso modo, la funzione intrinseca `INT` converte un valore reale o complesso nell'intero corrispondente. Ciò avviene troncando la parte decimale dell'argomento reale o della parte reale dell'argomento complesso.

Espressione	*Significato*
`INT(2.+1.5)/2`	`1`
`INT(3.9)`	`3`
`INT((3.,-4.))`	`3`

In maniera del tutto analoga a quanto visto per le funzioni intrinseche `REAL` e `INT`, la funzione intrinseca `CMPLX(X,Y)` combina gli argomenti `X` e `Y` a formare un dato di tipo complesso. Gli argomenti `X` e `Y` possono essere interi, reali o complessi, ed inoltre possono essere sia *dati scalari* che *array*. L'argomento `Y` è opzionale: se esso viene omesso si assume che abbia valore nullo. Nel caso particolare in cui `X` è un valore di tipo `COMPLEX`, soltanto le componenti di `X` verranno impiegate a formare il valore complesso finale.

Una nota sugli operatori aritmetici

Non tutte le operazioni aritmetiche avvengono sfruttando le stesse risorse di macchina o si caratterizzano per lo stesso tempo di esecuzione. Ad esempio, le operazioni aritmetiche fra interi sono molto più veloci delle corrispondenti operazioni su reali ed enormemente più veloci delle corrispondenti operazioni su complessi. Ma anche fra gli operatori applicabili allo stesso tipo di dati ne esistono alcuni più efficienti ed altri più lenti ed impegnativi. L'operazione di ottimizzazione detta di *strength reduction* ha lo scopo di sostituire una espressione con una equivalente ma che faccia uso di operatori meno "costosi". Del tutto in generale si può dire che l'operazione di somma fra interi è certamente più efficiente dell'operazione di moltiplicazione così come la somma di due reali è più veloce del prodotto misto fra un intero ed un reale. L'operazione di moltiplicazione, poi, è sempre più efficiente di una operazione di divisione ed enormemente più rapida di un elevamento a potenza intera o reale. Di seguito si riportano a titolo di esempio, alcune semplici istruzioni aritmetiche e la loro versione più efficiente (con i e c si sono indicati due generici valori interi, con x un valore reale):

$x \times 2$	$x + x$
$x/2$	$0.5 \times x$
x^2	$x \times x$
$x^{c.5}$	$x^c \times \sqrt{x}$
$(a, 0) + (b, 0)$	$(a + b, 0)$

Una delle applicazioni più importanti di questa tecnica è proprio la possibilità di convertire l'elevamento a potenza in comuni moltiplicazioni, come ad esempio nella valutazione di polinomi sfruttando la seguente identità:

$$a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0 = (a_n x^{n-1} + a_{n-1} x^{n-2} + ... + a_1)\, x + a_0$$

Un'altra operazione assai importante che ha lo scopo di ridurre la complessità di calcolo di un codice numerico è rappresentata dalla possibilità di semplificare le espressioni aritmetiche applicando ad esse opportune regole algebriche. In tal senso, una classe di semplificazioni particolarmente utili si basa sulle identità algebriche. Ad esempio, l'istruzione:

```
x = (y*1.0)/1.
```

può essere efficacemente trasformata in

```
x = y
```

In generale, dunque, è buona norma individuare all'interno del codice delle espressioni "banali" e sostituirle con i corrispondenti valori. Di seguito si elencano alcune delle più comuni identità aritmetiche:

$x \times 0$	$=$	0
$0/x$	$=$	0
$x \times 1$	$=$	x
$x + 0$	$=$	x
$x/1$	$=$	x

E', comunque, da osservare che la maggior parte dei compilatori è in grado di individuare espressioni del tipo precedente e normalmente opera automaticamente questo tipo di semplificazioni.

1.5.2 Espressioni di caratteri

La manipolazione degli oggetti di tipo `CHARACTER` può essere svolta a mezzo di una istruzione di assegnazione il cui formato è:

variabile = espressione_di_caratteri

con il significato che l'istruzione di assegnazione calcola il valore di *espressione_di_caratteri* alla destra del segno di uguale ed assegna tale valore a *variabile* alla sinistra dell'uguale.

L'*espressione_di_caratteri* può essere una qualsiasi combinazione valida di costanti, variabili e operatori di caratteri. Un *operatore di caratteri* è un particolare operatore che elabora dei caratteri e fornisce come risultato ancora dei caratteri. I principali tipi di operatori di caratteri sono le *specifiche di sottostringhe* e l'*operatore di concatenazione*.

L'assegnazione del valore di un'espressione ad una variabile di caratteri merita un'osservazione. Se l'espressione di caratteri è *più corta* della variabile alla quale deve essere assegnata, la parte restante della variabile viene riempita con spazi vuoti (*blank*). Ad esempio, le seguenti espressioni registrano il valore "`casa**`" nella variabile `parola` (con il simbolo "`*`" si è indicato uno spazio bianco):

```
CHARACTER(LEN=6) :: parola
parola = "casa"
```

Viceversa, se l'espressione di caratteri è *più lunga* della variabile alla quale deve essere assegnata, la parte in eccesso della variabile viene "troncata". Ad esempio, le seguenti istruzioni registrano il valore "`case`" nella variabile `parola`:

```
CHARACTER(LEN=4) :: parola
parola = "casetta"
```

Specifiche di sottostringa

Una *specifica di sottostringa* seleziona una "porzione" di una variabile di caratteri e tratta questa variabile come se fosse una variabile stringa indipendente. La porzione di stringa selezionata è chiamata *sottostringa*.

Una sottostringa viene selezionata (*estratta*) "appendendo" al nome della variabile stringa uno *specificatore di lunghezza*. Quest'ultimo ha la forma:

`(`*inizio*`:`*fine*`)`

in cui le costanti intere *inizio* e *fine* indicano la posizione del *primo* e dell'*ultimo* carattere da estrarre. Pertanto, ad esempio, lo specificatore di lunghezza "`(3:5)`" indica che la sottostringa consiste del terzo, del quarto e del quinto carattere della stringa "madre".

Si noti che se la costante *inizio* viene omessa, essa è assunta per default pari ad uno; se, invece, è omessa la costante *fine* il suo valore è assunto pari alla *lunghezza della stringa*; se, infine, *inizio* risulta maggiore di *fine* viene generata una *stringa nulla*.

Ad esempio, data la stringa così dichiarata:

```
CHARACTER(LEN=8) :: stringa="abcdefgh"
```

possibili esempi di sottostringa sono:

```
stringa(2:4)    ! "bcd"
stringa(3:7)    ! "cdefg"
stringa(:4)     ! "abcd"
stringa(7:)     ! "gh"
stringa(5:5)    ! "e"
stringa(6:3)    ! Stringa nulla
```

Si noti che una sottostringa fa riferimento alle *stesse* locazioni di memoria della stringa madre per cui se viene modificato il contenuto della sottostringa cambiano automaticamente anche i caratteri della stringa madre.

Come detto in precedenza, una sottostringa può essere usata come una qualunque variabile stringa "originale" per cui, in particolare, essa può essere usata a sinistra di una espressione di assegnazione. Quanto detto è ben illustrato dal seguente programma:

```
PROGRAM test_sottostringhe
  IMPLICIT NONE
  CHARACTER(LEN=6) :: str1, str2, str3
  str1 = "abcdef"
  str2 = "123456"
  str3 = str1(3:5)
  str2(5:6) = str1(1:2)
  PRINT*, str1
  PRINT*, str2
  PRINT*, str3
END PROGRAM test_sottostringhe
```

il cui output è:

```
abcdef
1234ab
cde
```

Operatore di concatenazione di stringhe

L'*operatore di concatenazione di stringhe* combina due o più stringhe (o sottostringhe) a formare un'unica stringa più grande. Esso è rappresentato dal simbolo di un doppio slash ("`//`"). La forma generale di questo operatore è la seguente:

stringa_1`//`*stringa_2* `[//` *stringa_3* `...]`

in cui, se le stringhe *stringa_1* e *stringa_2* hanno lunghezza *n_1* ed *n_2*, rispettivamente, la lunghezza della stringa finale sarà *n_1* + *n_2*.

Si consideri il seguente frammento di programma:

```
CHARACTER(LEN=7) :: nome1="Donald"
CHARACTER(LEN=5) :: nome2="Duck", nome3="Duffy"
CHARACTER(LEN=11) :: ans1, ans2, ans3
...
ans1 = nome1//nome2
ans2 = nome3//" "//nome2
ans3 = nome2//nome2//nome2
```

A seguito delle operazioni di concatenazione, il contenuto delle variabili `ans1`, `ans2`, `ans3`, sarà il seguente:

```
ans1:  "Donald*Duck"
ans2:  "Duffy*Duck*"
ans3:  "DuckDuckDuc"
```

in cui, come in precedenza, con il simbolo "*" si è voluto indicare graficamente uno spazio vuoto.

Il seguente programma fornisce un esempio un pò più completo delle possibili applicazioni degli strumenti di manipolazione delle stringhe di caratteri:

```
PROGRAM  DateTime
  IMPLICIT NONE
  CHARACTER(LEN=8)  :: date        ! ccyymmdd
  CHARACTER(LEN=4)  :: anno
  CHARACTER(LEN=2)  :: mese, giorno
  CHARACTER(LEN=10) :: time        ! hhmmss.sss
  CHARACTER(LEN=12) :: orario
  CHARACTER(LEN=2)  :: ore, minuti
  CHARACTER(LEN=6)  :: secondi
  CALL DATE_AND_TIME(date,time)
! decompone la variabile "date" in anno, mese e giorno
! date ha forma 'ccyymmdd', dove cc = secolo, yy = anno
! mm = mese e dd = giorno
  anno = date(1:4)
  mese = date(5:6)
  giorno = date(7:8)
  WRITE(*,*) ' Data attuale -> ', date
  WRITE(*,*) '         Anno -> ', anno
  WRITE(*,*) '         Mese -> ', mese
  WRITE(*,*) '       Giorno -> ', giorno
! decompone la variabile "time" in ore, minuti e secondi
! time ha forma 'hhmmss.sss', dove h = ore, m = minuti
! ed s = secondi
  ore = time(1:2)
  minuti = time(3:4)
  secondi = time(5:10)
  orario = ore // ':' // minuti // ':' // secondi
  WRITE(*,*)
  WRITE(*,*) ' Orario attuale -> ', time
  WRITE(*,*) '            Ore -> ', ore
  WRITE(*,*) '         Minuti -> ', minuti
  WRITE(*,*) '        Secondi -> ', secondi
  WRITE(*,*) '         Orario -> ', orario
END PROGRAM  DateTime
```

Un possibile output di questo programma potrebbe essere:

```
Data attuale -> 19970811
        Anno -> 1997
        Mese -> 08
      Giorno -> 11

Orario attuale -> 010717.620
           Ore -> 01
        Minuti -> 07
       Secondi -> 17.620
        Orario -> 01:07:17.620
```

Si noti che si sarebbe potuto "assemblare" la variabile `orario` anche al modo seguente:

```
orario(:2) = ore
orario(3:3) = ':'
orario(4:5) = minuti
orario(6:6) = ':'
orario(7:) = secondi
```

Il programma precedente fa uso della subroutine intrinseca `DATE_AND_TIME` la quale restituisce informazioni circa la data e l'ora in due argomenti stringa. Il primo, `date`, deve avere una lunghezza di almeno otto caratteri ed il valore ritornato ha la forma `ccyymmdd`, in cui `cc` rappresenta il secolo, `yy` l'anno, `mm` il mese, e `dd` il giorno. Se, ad esempio, il giorno corrente fosse l'11 Agosto del 1998, la chiamata alla subroutine restituirebbe la stringa di otto caratteri: `19980811`. Il secondo argomento, `time`, riceve una stringa di 12 caratteri nella forma `hhmmss.sss`, in cui `hh` rappresenta le ore, `mm` i minuti e `ss.sss` i secondi. Così, se questa subroutine venisse invocata all'una e 7 minuti e 17.620 secondi, il valore restituito sarebbe la stringa `010717.620`.

Relazione d'ordine fra stringhe

Il processo mediante cui due operatori stringa vengono confrontati si articola nei seguenti passi:

- Se i due operandi *non* hanno la stessa lunghezza, al più corto dei due vengono idealmente aggiunti spazi bianchi in modo da rendere le stringhe di uguale lunghezza.
- I due operandi vengono confrontati carattere per carattere a partire dal primo (ossia da quello più a sinistra) fino a quando non vengono riscontrati due caratteri differenti oppure fino al raggiungimento della fine delle stringhe.
- Se vengono trovati due caratteri corrispondenti differenti, allora è la relazione tra questi due caratteri che determina la relazione tra i due operandi. In particolare viene considerato "minore" il carattere che viene prima nella *sequenza di collating* del processore. Se non viene riscontrata alcuna differenza le due stringhe vengono considerate uguali.

Così, ad esempio, la relazione:

```
"Adamo" > "Eva"
```

è *falsa* in quanto in qualunque sequenza di collating la lettera `A` precede sempre la `E`, ossia `A` è *minore di* `E`. Allo stesso modo, la relazione:

```
"Eva" < "Evaristo"
```

è *vera* poiché la stringa `Eva` viene prolungata con cinque caratteri vuoti e la relazione fra le due stringhe si riduce alla relazione:

```
" " < "r"
```

e il carattere nullo precede tutti i caratteri alfanumerici.

Si noti, tuttavia, che le relazioni:

```
"ADAMO" < "Adamo"
"X4" < "XA"
"var_1" < "var-1"
```

non sono definite in Fortran ed il valore di ciascuna di esse dipende dalla particolare sequenza di collating implementata dal processore in uso. Questo significa che lo standard del linguaggio *non* definisce alcun ordine relativo tra cifre e lettere, fra lettere minuscole e maiuscole, e fra gli elementi del set di caratteri speciali.

Se, per ragioni di portabilità, fosse richiesta la definizione di un ordine per tutti i caratteri allora è necessario fare uso di un altro metodo di confronto. Questo nuovo metodo si basa sull'utilizzo di quattro *funzioni intrinseche lessicali*: `LGT`, `LGE`, `LLE`, `LLT`, la cui sintassi è:

```
LGT(STRING_A,STRING_B)
LGE(STRING_A,STRING_B)
LLE(STRING_A,STRING_B)
LLT(STRING_A,STRING_B)
```

ed il cui significato è, rispettivamente:

```
STRING_A >  STRING_B
STRING_A >= STRING_B
STRING_A <= STRING_B
STRING_A <  STRING_B
```

sempre e *soltanto* secondo l'ordinamento ASCII. Dunque, queste funzioni restituiscono valore `.TRUE.` o `.FALSE.` come risultato del confronto delle stringhe argomento secondo l'ordinamento della tabella ASCII, sicché il loro risultato è indipendente dal processore e, quindi, completamente portabile.

Così, ad esempio, mentre il valore dell'espressione:

```
"Eva" > "eva"
```

non è univocamente definito dallo standard ma dipende dal particolare processore, l'espressione:

```
LGT("Eva","eva")
```

darà sempre valore `.FALSE.` in quanto nella sequenza di collating ASCII le lettere maiuscole precedono sempre quelle minuscole (l'ordinamento lessicografico ASCII è riportato in Appendice A).

1.5.3 Espressioni logiche

Analogamente alle espressioni matematiche, anche le espressioni logiche sono svolte da un'istruzione di assegnazione, il cui formato è il seguente:

```
variabile_logica = espressione_logica
```

L'istruzione di assegnazione calcola il valore dell'espressione alla destra del segno di uguale e assegna tale valore alla variabile a sinistra del segno di uguale.

L'istruzione a secondo membro può essere formata da una qualsiasi combinazione valida di costanti, variabili e operatori logici. Un *operatore logico* è un operatore che elabora dati di qualsiasi tipo ma fornisce sempre un risultato di tipo logico. Gli operatori logici possono essere di due tipi: *operatori relazionali* e *operatori combinatori*. Di entrambi si forniranno i dettagli nei prossimi paragrafi.

Operatori relazionali

Gli operatori relazionali sono particolari operatori che agiscono su due operandi numerici o su due stringhe e forniscono un risultato di tipo logico. Il risultato dipende dalla *relazione* fra i due valori che vengono confrontati (da cui il nome *relazionale*). Il formato generale di un tale operatore è il seguente:

```
arg_1 op arg_2
```

in cui *arg_1* e *arg_2* sono espressioni aritmetiche, variabili o costanti numeriche o stringhe di caratteri; `op` è, invece, uno dei seguenti operatori relazionali:

Operatore	*Significato*
==	uguale a
/=	diverso da
>	maggiore di
>=	maggiore o uguale di
<	minore di
<=	minore o uguale a

Se la relazione fra *arg_1* e *arg_2* è vera allora `op` fornirà valore `.TRUE.`, altrimenti il risultato sarà `.FALSE.` I seguenti esempi mostrano come funzionano gli operatori relazionali:

```
INTEGER :: i=3, j=4
...
i == j      ! .FALSE.
i /= j      ! .TRUE.
j > 5       ! .FALSE.
"A" < "B"   ! .TRUE.
"a" < "A"   ! .TRUE. (per la sequenza di collating ASCII)
```

Si noti che la ultime due espressioni logiche sono .TRUE. in virtù della relazione d'ordine che esiste fra i caratteri di un qualunque sistema di codifica e della tabella ASCII in particolare.

Nella gerarchia delle operazioni, gli operatori relazionali vengono valutati dopo tutti gli operatori aritmetici; ne consegue che le seguenti due espressioni sono perfettamente equivalenti (entrambe hanno valore .TRUE.):

```
7+3 < 2+11
(7+3) < (2+11)
```

Se il confronto riguarda simultaneamente oggetti di tipo reale ed intero, il valore intero viene convertito in valore reale prima che il confronto abbia luogo. I confronti fra dati numerici e caratteri *non* sono ammessi e generano un errore in fase di compilazione. A titolo di esempio si osservino le seguenti espressioni:

```
4 == 4.     ! .TRUE. Il confronto avverrà tra 4.e 4.
4 < "A"     ! Espressione non valida (errore in fase di compilazione)
```

Operatori combinatori

Gli operatori combinatori sono particolari operatori che agiscono soltanto su operandi di tipo logico, fornendo un risultato anch'esso di tipo logico. Esistono quattro operatori *binari* (.AND., .OR., .EQV. e .NEQV.) e uno *unario* (.NOT.). Il formato generale di un operatore combinatorio è il seguente:

l_1 op *l_2*

in cui *l_1* e *l_2* sono variabili, costanti o espressioni di tipo LOGICAL; op è, invece, uno dei seguenti operatori relazionali:

Operazione	*Funzione*
l_1.AND.*l_2*	AND logico
l_1.OR.*l_2*	OR logico
l_1.EQV.*l_2*	Equivalenza logica
l_1.NEQV.*l_2*	Non equivalenza logica
.NOT.*l_1*	NOT logico

Se la relazione fra *l_1* e *l_2* espressa dall'operatore è vera, l'operatore fornisce il valore .TRUE., altrimenti il risultato è .FALSE.

I risultati degli operatori combinatori per ogni combinazione possibile di *l_1* e *l_2* sono sintetizzati nelle seguenti *tavole di verità*:

l_1	*l_2*	*l_1*.AND.*l_2*	*l_1*.OR.*l_2*	*l_1*.EQV.*l_2*	*l_1*.NEQV.*l_2*
.FALSE.	.FALSE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.
.FALSE.	.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.TRUE.	.TRUE.	.TRUE.	.TRUE.	.TRUE.	.FALSE.

l_1	.NOT. *l_1*
.FALSE.	.TRUE.
.TRUE.	.FALSE.

Nella gerarchia delle operazioni, gli operatori combinatori vengono valutati dopo tutti gli operatori aritmetici e relazionali. Gli operatori logici di un'espressione vengono valutati nel seguente ordine:

1. Tutti gli operatori aritmetici, nell'ordine precedentemente descritto.
2. Tutti gli operatori relazionali (==, /=, >, >=, <, <=), da sinistra verso destra.
3. Tutti gli operatori .NOT., da sinistra verso destra.
4. Tutti gli operatori .AND., da sinistra verso destra.
5. Tutti gli operatori .OR., da sinistra verso destra.
6. Tutti gli operatori .EQV. e .NEQV., da sinistra verso destra.

L'ordine standard degli operatori può essere mutato soltanto con un opportuno uso delle parentesi. Come esempio di quanto detto, si può considerare il seguente set di istruzioni:

```
LOGICAL :: l_1=.TRUE., l_2=.TRUE., l_3=.FALSE.
...
.NOT.l_1              ! .FALSE.
l_1.OR.l_3            ! .TRUE.
l_1.AND.l_3           ! .FALSE.
l_3.EQV.l_2           ! .FALSE.
l_2.NEQV.l_3          ! .TRUE.
l_1.AND.l_2.OR.l_3    ! .TRUE.
l_1.OR.l_2.AND.l_3    ! .FALSE.
.NOT.(l_3.OR.l_1)     ! .FALSE.
```

Come si può notare, gli operatori combinatori vengono valutati dopo tutti gli operatori relazionali; l'operatore .NOT. viene valutato prima degli altri operatori combinatori. Pertanto, le parentesi nell'ultimo esempio sono necessarie, altrimenti l'espressione sarebbe valutata in quest'ordine:

```
(.NOT.l_3).OR.l_1
```

che, invece, dà risultato .TRUE.

Si noti che nel Fortran 90/95 standard non sono ammesse le operazioni combinatorie che riguardano dati numerici o caratteri. Operazioni come queste:

```
4.AND.3     ! Errore di compilazione
```

generano un errore in fase di compilazione.

Si vuole concludere il paragrafo con una considerazione che può tornare spesso utile ai fini di un miglioramento dell'efficienza di un codice di calcolo. Si supponga di dover valutare l'espressione booleana: *l_1*.AND.*l_2*. In casi come questo, chiaramente, se *l_1* risulta .FALSE. non c'è motivo di valutare anche *l_2*. Specialmente se la valutazione delle espressioni logiche è particolarmente onerosa oppure se esse devono essere valutate più volte nel corso dell'esecuzione della stessa unità di programma, assume particolare importanza il valutare inizialmente soltanto *l_1* riservandosi di valutare anche *l_2* se e solo se *l_1* è vera. Questo "trucco" (spesso chiamato "cortocircuito") è, di solito, applicato dal compilatore in maniera automatica, nondimeno è importante riconoscere casi del genere e applicare questa scorciatoia "a mano".

1.6 Introduzione alle operazioni di input/output

Una fase importante della programmazione è quella che consiste nel creare una interfaccia fra l'utente umano e la macchina tramite la quale sia possibile inserire, in fase di esecuzione, i valori dei dati di cui il codice necessita per operare la fase di calcolo, e poter leggere i risultati prodotti dal programma.

Queste operazioni sono note come *operazioni di input/output* o *di lettura/scrittura* o, ancora, *di ingresso/uscita*. Il Fortran 90/95 comprende un insieme eccezionalmente vasto di meccanismi preposti allo scopo. Le operazioni di input/output saranno trattate diffusamente nei capitoli 4 (dedicato alle operazioni cosiddette "interattive", vale a dire orientate ai dispositivi standard quali tastiera e terminale video) ed 8 (dedicato, invece, alla gestione delle memorie di massa). Scopo di questo paragrafo è semplicemente quello di fornire una sommaria introduzione alle forme più semplici di lettura e scrittura al fine di consentire di scrivere già semplici programmi che prevedano un flusso di informazioni da o verso il codice di calcolo, rimandando ai suddetti capitoli una trattazione esaustiva dell'argomento.

Le operazioni di input/output si realizzano in Fortran mediante le operazioni READ, WRITE e PRINT la cui sintassi, nella sua forma più "essenziale", è, rispettivamente:

```
READ(unità_logica,formato) lista_di_variabili
WRITE(unità_logica,formato) lista_di_espressioni
PRINT formato, lista_di_espressioni
```

in cui *unità_logica* individua il dispositivo di lettura/scrittura da utilizzare, mentre *formato* identifica la modalità di interpretazione (*formattazione*) dei dati da fornire in ingresso o prodotti in uscita.

L'effetto dell'istruzione READ è quello di "leggere" dal dispositivo identificato dal valore dell'*unità_logica* una lista di valori (*record di ingresso*) secondo la modalità di interpretazione stabilita dal valore del *formato* e di assegnare detti valori agli elementi della *lista di variabili* nell'ordine in cui essi sono specificati.

L'effetto dell'istruzione WRITE è quello di calcolare il valore della *lista di espressioni* e di "scrivere" detti valori sul dispositivo identificato dal valore dell' *unità_logica* secondo la modalità di formattazione stabilita dal valore del *formato* (*record di uscita*).

L'istruzione `PRINT` si comporta analogamente a `WRITE` salvo per il fatto che il dispositivo di uscita è sempre quello standard del sistema di calcolo utilizzato.

E' possibile semplificare ulteriormente queste istruzioni sostituendo un asterisco ("*") agli argomenti *unità_logica* e *formato*, riferendosi in tal modo ai *dispositivi standard* di ingresso/uscita e formattando i dati con *impostazioni di default* previsti dalla specifica implementazione:

```
READ(*,*) lista_di_variabili
WRITE(*,*) lista_di_espressioni
PRINT* lista_di_espressioni
```

In generale, però, il parametro *formato* avrà come valore una *etichetta* cui è riferita una istruzione `FORMAT` la quale rappresenta una particolare *istruzione non eseguibile* avente lo scopo di descrivere la *specificazione di formato* da utilizzare. Di tutto questo, però si parlerà diffusamente nel seguito.

Come esempio si consideri il seguente frammento di programma:

```
INTEGER :: m, n
...
READ(*,*) m, n
WRITE(*,*) "Il risultato e': ", m + 2*n
```

La prima istruzione eseguibile legge due valori dal dispositivo standard di input (ad esempio, la tastiera) e ne assegna i valori *ordinatamente* alle variabili elencate nella lista di ingresso. Si noti che i valori introdotti devono essere separati da virgole o da spazi bianchi. La seconda istruzione, invece, valuta le espressioni della lista di uscita e ne produce i valori *ordinatamente* sul dispositivo standard di output (ad esempio, il terminale video). Così, se il record di ingresso fosse il seguente:

```
3 4
```

il risultato a video sarebbe:

```
Il risultato e':           11
```

Come si nota, il record di uscita è formattato piuttosto male, con un numero eccessivo di caratteri bianchi fra il primo elemento della lista di uscita (la stringa `Il risultato e':  `) ed il secondo (il valore dell'espressione intera `m+2*n`). Una migliore formattazione si otterrebbe ricorrendo ad una esplicita specificazione di formato, ad esempio, mediante un'istruzione `FORMAT`, come suggerito dal seguente esempio:

```
WRITE(*,100) "Il risultato e': ", m+2*n
100 FORMAT(1X,A17,I2)
```

che serve a strutturare il record di uscita in modo tale che il primo carattere sia uno spazio bianco (`1X`), seguito da una stringa di diciassette caratteri (`A17`) seguita, a sua volta, da un intero di due cifre (`I2`). Il risultato a video apparirebbe, allora, in questo modo:

```
 Il risultato e': 11
```

Analogamente, il seguente frammento di programma:

```
REAL :: pi=3.14
INTEGER :: m=13
...
WRITE(*,*) "Pi greco: ", pi, "Indice m: ", m
```

produce il seguente output:

```
Pi greco:    3.140000    Indice m:           13
```

mentre se l'istruzione di uscita fosse formattata al modo seguente:

```
WRITE(*,100) "Pi greco: ", pi, "Indice m: ", m
100 FORMAT(1X,A10,F4.2,/,1X,A10,I2)
```

il risultato sarebbe:

```
 Pi greco: 3.1
 Indice m: 13
```

In questo caso, infatti, l'istruzione `FORMAT` "edita" il risultato in modo tale che il primo carattere prodotto sia uno spazio bianco, seguito da una stringa di dieci caratteri (`A10`), seguita a sua volta da un valore reale rappresentato con quattro caratteri (incluso il carattere per il punto decimale) di cui due destinati alle cifre decimali (`F4.2`); segue quindi un *avanzamento di record* (`/`) e, di nuovo, su questo nuovo record il primo carattere è uno spazio bianco (`1X`) seguito da una stringa di dieci caratteri (`A10`) seguita, infine, da un valore intero di due cifre (`I2`).

Come detto, però, alle istruzioni di lettura e scrittura formattate sarà dedicato ampio spazio nel capitolo 4.

Le operazioni di lettura/scrittura per dati di tipo `COMPLEX` sono lievemente più complesse dal momento che un dato complesso è, in realtà, una *coppia di valori* di tipo `REAL`. Infatti, la lettura o la scrittura di un valore di tipo complesso implicano la lettura o la scrittura di *due* dati di tipo reale, corrispondenti alle parti reale ed immaginaria del dato complesso (comprensive di parentesi e di virgola come separatore). Il seguente esempio mostra, allo scopo, un semplice esempio di operazioni di lettura e scrittura di dati complessi:

```
PROGRAM aritmetica_complessa
  IMPLICIT NONE
  COMPLEX :: a, b, c
! Lettura di due numeri complessi
  READ(*,'(2F10.3)'), a,b
  c = a*b
! Stampa il loro prodotto
  WRITE(*,*) c
END PROGRAM aritmetica_complessa
```

Se, ad esempio, l'input del programma fosse:

```
(2.500, 8.400)
(.500,  9.600)
```

il corrispondente output sarebbe:

```
(-5.000000,10.00000)
```

1.7 Tipi di dati parametrizzati

Uno dei problemi più seri che affiggono i programmatori che sviluppano software scientifico è rappresentato dalla cosiddetta *portabilità numerica*; con questa dicitura si indica il fatto che la *precisione* ed il *range* di un stesso tipo di dati potrebbe essere differente da processore a processore. Ad esempio, su molte installazioni un valore di tipo `REAL` in singola precisione occupa 32 bit mentre un valore in doppia precisione ne occupa 64, su altre macchine, invece, i valori in singola precisione usano 64 bit mentre quelli in doppia precisione ne usano 128. Questo significa, ad esempio, che ottenere lo stesso risultato che richiede soltanto la singola precisione su un computer potrebbe richiedere l'impiego di dati in doppia precisione su un altro. Questo semplice esempio mostra quanto delicato possa essere il trasferimento di un programma da un computer ad un altro e come i termini *singola precisione* e *doppia precisione* non debbano essere concepiti come concetti definiti in maniera "esatta".

Uno degli obiettivi del Fortran 90/95 è stato quello di superare tale problema di portabilità: il linguaggio, infatti, implementa un meccanismo (*portabile*) di scelta della precisione e supporta tipi che possono essere parametrizzati a mezzo di un valore di *kind*. Il valore di kind viene usato per scegliere un modello di rappresentazione per un tipo e, relativamente ai soli tipi numerici, può essere specificato in termini della precisione richiesta. Un processore può supportare diverse precisioni (o *rappresentazioni*) di dati di tipo `INTEGER`, `REAL` e `COMPLEX`, differenti set di `CHARACTER` (ad esempio, caratteri arabi o cirillici, notazioni musicali etc.) e modi diversi di rappresentare gli oggetti di tipo `LOGICAL`.

In Fortran a ciascuno dei cinque tipi di dati intrinseci (`INTEGER`, `REAL`, `COMPLEX`, `LOGICAL`, `CHARACTER`) è associato un valore intero non negativo chiamato *parametro di kind*. E' previsto che un processore supporti *almeno due* valori di kind per i tipi `REAL` e `COMPLEX` ed *almeno uno* per i tipi `INTEGER`, `LOGICAL` e `CHARACTER`.

Un esempio di "parametrizzazione" di un tipo intrinseco é:

```
INTEGER(KIND=1) :: ik1
REAL(KIND=4) :: rk4
```

I parametri di kind corrispondono alle differenti "precisioni" supportate dal compilatore.

Assegnare esplicitamente un appropriato parametro di kind a ciascun tipo di dato è ancora una soluzione non portabile in quanto uno stesso valore di kind potrà corrispondere, su macchine differenti, a differenti livelli di precisione. Tuttavia, relativamente ai soli tipi numerici, il linguaggio mette a disposizione un meccanismo di selezione del valore di kind sulla base della precisione desiderata. Di ciò si parlerà diffusamente nei prossimi sottoparagrafi nei quali si specializzerà il discorso relativamente a ciascuno dei tipi intrinseci del Fortran.

Parametri di kind per il tipo `REAL`

Il parametro di tipo kind associato ad una variabile reale specifica la *precisione* (in termini di cifre decimali) ed il *range degli esponenti* (secondo la rappresentazione in potenze di 10). Poiché, infatti, i compilatori Fortran prevedono almeno due tipi diversi di variabili reali (*singola precisione* e *doppia precisione*) è necessario specificare quale tipo (*kind*) di variabile si vuole utilizzare: per specificare una di queste due varianti si usa lo specificatore `KIND`.

Il tipo di variante reale è specificato esplicitamente includendone il valore fra parentesi nella istruzione di dichiarazione di tipo subito dopo lo specificatore di tipo, come mostrano le seguenti istruzioni:

```
REAL(KIND=1) :: a, b, c
REAL(KIND=4) :: x
REAL(KIND=8) :: temp
```

Si noti che la parola chiave `KIND=` è opzionale sicché le seguenti istruzioni:

```
REAL(KIND=4) :: var
REAL(4) :: var
```

sono perfettamente equivalenti. Una variabile dichiarata esplicitamente con un parametro di kind si chiama *variabile parametrizzata*. Se non viene specificato alcun parametro `KIND`, viene utilizzato il tipo di variabile reale di default.

Tuttavia, quale sia il significato dei numeri di kind non è possibile saperlo, nel senso che i produttori di compilatori sono liberi di assegnare i propri numeri di kind alle varie dimensioni delle variabili. Ad esempio, in alcuni compilatori un valore reale a 32 bit corrisponde a `KIND=1` e un valore reale a 64 bit corrisponde a `KIND=2`. In altri compilatori, invece, un valore reale a 32 bit corrisponde a `KIND=4` e un valore reale a 64 bit corrisponde a `KIND=8`. Di conseguenza, affinché un programma possa essere eseguito indipendentemente dal processore (*portabilità del software*) è consigliabile assegnare i numeri di kind ad apposite costanti ed utilizzare quelle costanti in tutte le istruzioni di dichiarazione di tipo. Cambiando semplicemente i valori di alcune costanti è possibile eseguire il programma su installazioni differenti. A chiarimento di quanto detto si può considerare il seguente frammento di programma:

```
INTEGER, PARAMETER :: single=4
INTEGER, PARAMETER :: double=8
...
REAL(KIND=single) :: val
REAL(single) :: temp
REAL(KIND=double) :: a, b, c
```

(nell'esempio in esame, evidentemente, i valori `4` e `8` dipendono dallo specifico compilatore utilizzato).

Una tecnica ancora più efficiente è quella di definire i valori dei parametri di kind in un *modulo* e utilizzare questo modulo nelle singole procedure all'interno del programma, come nel seguente esempio:

```
MODULE mymod
   IMPLICIT NONE
   INTEGER, PARAMETER :: single=1
   INTEGER, PARAMETER :: double=2
END MODULE

PROGRAM prova
  USE mymod
  IMPLICIT NONE
  REAL(KIND=single) :: x, y, z
  REAL(KIND=double) :: a, b, c
  ...
  x = 3.0_single + y/z
  a = -1.5_double + b -2.0_double*c
  ...
END PROGRAM prova
```

In tal modo è possibile modificare i numeri di kind di un intero programma intervenendo su un solo file.

Oltre che nelle istruzioni di dichiarazione (di variabile o di costante con nome), è possibile specificare il tipo di kind anche per una costante *letterale*. A tale scopo è sufficiente aggiungere alla costante il carattere *underscore* ("_") seguito dal numero di kind. Esempi validi di costanti reali parametrizzati sono i seguenti:

```
34._4
1234.56789_double
```

Chiaramente, la prima espressione è valida soltanto se `KIND=4` è un tipo di dato reale valido per il processore in uso, la seconda espressione è corretta soltanto se `double` è una costante precedentemente definita con un numero di kind valido. Una costante senza identificatore di tipo kind viene trattata come costante di tipo standard che in molti processori è una costante in singola precisione.

Oltre che nel modo precedente, è possibile dichiarare una costante in doppia precisione nella notazione scientifica utilizzando la lettera "`D`" al posto della "`E`" per specificare l'esponente della costante:

`3.0E0`	costante in singola precisione
`3.0D0`	costante in doppia precisione

Per conoscere i numeri di kind utilizzati da un compilatore è possibile usare la *funzione intrinseca* `KIND` la quale restituisce il numero del parametro di kind assegnato a una costante o una variabile. Il più semplice esempio di utilizzo della funzione `KIND` è quello di determinare i numeri di kind associati alle variabili in singola e in doppia precisione su una particolare installazione:

```
PRINT*, "Il numero di kind per la singola precisione e': ", KIND(0.0E0)
PRINT*, "Il numero di kind per la doppia precisione e': ", KIND(0.0D0)
```

Se queste istruzioni venissero eseguite su un PC con processore Intel Pentium III che usi il compilatore Digital Visual Fortran 6.0, l'output sarebbe:

```
Il numero di kind per la singola precisione e': 4
Il numero di kind per la doppia precisione e': 8
```

Evidentemente, passando ad un altro processore oppure utilizzando un diverso compilatore il risultato potrebbe essere differente.

Per poter eseguire un programma in maniera corretta su processori differenti è necessario un meccanismo in grado di selezionare *automaticamente* il parametro appropriato per il valore reale quando il programma venga trasferito fra due sistemi differenti. Tale meccanismo è rappresentato dalla *funzione intrinseca* `SELECTED_REAL_KIND`: quando viene eseguita, essa restituisce il numero del parametro di kind del *più piccolo* valore reale che soddisfa il *range* e la *precisione* richiesti da un particolare processore. La forma generale di questa funzione è:

numero_di_kind = SELECTED_REAL_KIND(p=*precisione*, r=*range*)

dove ***precisione*** rappresenta il numero di cifre decimali della precisione richiesta e ***range*** è l'intervallo richiesto per gli esponenti delle potenze di 10. I due argomenti ***precisione*** e ***range*** sono facoltativi nel senso che è possibile specificare uno o entrambi gli argomenti per determinare le caratteristiche richieste per il valore reale. Inoltre, i termini `p=` e `r=` sono facoltativi, purché gli argomenti ***precisione*** e ***range*** siano specificati in questa sequenza; il termine `p=` può essere omesso se viene specificata soltanto la precisione.

Più precisamente, la funzione `SELECTED_REAL_KIND` restituisce il numero di kind del *più piccolo* valore reale che soddisfa alle caratteristiche richieste oppure `-1` se nel processore in uso *non* è possibile ottenere la precisione specificata con qualsiasi tipo di dato reale, `-2` se *non* è possibile ottenere il range specificato con qualsiasi tipo di dato reale, `-3` se *non* è possibile ottenere né la precisione né il range.

Un esempio di applicazione di questa funzione è dato dal seguente programma di test:

```
PROGRAM test_select_real
  IMPLICIT NONE
  INTEGER :: n1, n2, n3, n4, n5, n6, n7
  n1 = SELECTED_REAL_KIND(p=6,r=37)
  n2 = SELECTED_REAL_KIND(r=100)
  n3 = SELECTED_REAL_KIND(p=13,r=200)
  n4 = SELECTED_REAL_KIND(p=13)
  n5 = SELECTED_REAL_KIND(17)
  n6 = SELECTED_REAL_KIND(p=11,r=500)
  n7 = SELECTED_REAL_KIND(17,500)
  PRINT*, n1,n2,n3,n4,n5,n6,n7
END PROGRAM
```

Ancora una volta, eseguendo queste istruzioni sulla macchina di cui all'esempio precedente, il risultato sarebbe:

```
4   8   8   8   8  -1  -2  -3
```

dove il valore `4` rappresenta il numero di kind per la singola precisione mentre il valore `8` rappresenta il numero di kind per la doppia precisione (chiaramente altri processori potrebbero fornire valori differenti). Gli ultimi tre valori della funzione sono, rispettivamente, `-1`, `-2` e `-3` in quanto nessun PC con processore Intel possiede una precisione di 17 cifre decimali o un range che vada da -10^{500} a $+10^{500}$.

Esistono altre tre funzioni che possono tornare molto utili quando si lavora con tipi di dati parametrizzati. Esse sono:

`KIND(X)`
Restituisce il *numero di kind* di `X` essendo `X` una variabile o una costante di tipo qualsiasi (non necessariamente numerico).

`PRECISION(X)`
Restituisce la *precisione decimale* (ossia il numero di cifre decimali) della rappresentazione di `X`, essendo `X` un valore reale o complesso.

`RANGE(X)`
Restituisce il *range di potenze decimali* che può essere supportato dal tipo di `X`, essendo quest'ultimo un dato di tipo intero, reale o complesso.

L'uso di queste funzioni è illustrato nel seguente programma:

```
PROGRAM parametri_di_kind
  IMPLICIT NONE
! Dichiarazione dei parametri
  INTEGER, PARAMETER :: single=SELECTED\_REAL\_KIND(p=6,r=37)
  INTEGER, PARAMETER :: double=SELECTED\_REAL\_KIND(p=13,r=200)
! Dichiarazione del tipo delle variabili
  REAL(KIND=single) :: var1=0._single
  REAL(KIND=double) :: var2=0._double
! Stampa dei parametri di kind delle variabili dichiarate
  WRITE(*,100) 'var1:',KIND(var1),PRECISION(var1),RANGE(var1)
  WRITE(*,100) 'var2:',KIND(var2),PRECISION(var2),RANGE(var2)
100 FORMAT(1X,A,'KIND = ',I2,'Precisione = ',I2,'Range = ',I3)
END PROGRAM
```

Se questo programma viene compilato con il compilatore Digital Visual Fortran 6.0 ed eseguito su un PC Pentium III, si otterrà il seguente risultato:

```
var1: KIND = 4,  Precisione =   6   Range =   37
var1: KIND = 8,  Precisione =  15   Range =  307
```

Si noti che il programma richiede, per la variabile `var2`, una precisione di 13 cifre ed un range di 200 potenze di 10, ma la variabile *effettivamente* assegnata dal processore ha una precisione di 15 cifre ed un range di 307 potenze di 10. Questo tipo di variabile reale, infatti, è, nel processore in uso, quello di dimensione *più piccola* fra quelli che soddisfano o superano la specifica richiesta.

Si vuole concludere il paragrafo con un esempio che mostra quale effetto possa avere sui risultati di un codice di calcolo la scelta di un parametro di kind piuttosto che un altro. Si farà riferimento, in particolare, al compilatore Intel Fortran Compiler ver. 5.0 per il quale le clausole `KIND=4` e `KIND=8` sono rappresentative di oggetti di tipo `REAL` rispettivamente in singola e in doppia precisione. Il programma che segue ha lo scopo di valutare la derivata numerica di una funzione $f(x)$ secondo il *metodo esplicito di Eulero*:

$$\frac{df}{dx} = \frac{f(x+dx) - f(x)}{dx}$$

Si noti che il programma fa uso di particolari costrutti (come `DO` e `IF`) e di procedure interne che verranno trattate soltanto nei prossimi capitoli.

```
PROGRAM derivate
!-----------------------------------------------------------------
! Scopo: Illustrare l'effetto della scelta del parametro di KIND
! sull'accuratezza computazionale. Obiettivo del programma e'
! valutare la derivata della finzione di test
!  f(x) = sin(2x)
! usando passi dx decrescenti e confrontando i risultati via via
! ottenuti con il risultato analitico.
!-----------------------------------------------------------------
  IMPLICIT NONE
  REAL(KIND=4) :: f1_4, f2_4, dfdx_4, x_4 = 2.3_4, dx_4 = 1
  REAL(KIND=8) :: f1_8, f2_8, dfdx_8, x_8 = 2.3_8, dx_8 = 1
  REAL(KIND=4) :: dfdx_exact_4
  REAL(KIND=8) :: dfdx_exact_8
  REAL(KIND=4) :: rel_err_4, abs_err_4
  REAL(KIND=8) :: rel_err_8, abs_err_8
! Determina l'esatto valore della derivata
  dfdx_exact_4 = dfdx4(x_4)
  dfdx_exact_8 = dfdx8(x_8)
! Scrive l'header della tabella di confronto
  WRITE(*,*) "NB: I confronti sono stati effettuati per x=2.3"
  WRITE(*,'(T6,A,T14,A,T26,A,T37,A,T48,A,T59,A)') &
      "dx", "df/dx_ex", "df/dx_4", "rel_err", "df/dx_8", "rel_err"
  DO
! Calcola la derivata numerica per KIND=4
     f2_4 = f4(x_4 + dx_4)
     f1_4 = f4(x_4)
```

```
      dfdx_4 = (f2_4 - f1_4)/dx_4
      abs_err_4 = ABS(dfdx_exact_4 - dfdx_4)
      rel_err_4 = abs_err_4/ABS(dfdx_exact_4)
! Calcola la derivata numerica per KIND=8
      f2_8 = f8(x_8 + dx_8)
      f1_8 = f8(x_8)
      dfdx_8 = (f2_8 - f1_8)/dx_8
      abs_err_8 = ABS(dfdx_exact_8 - dfdx_8)
      rel_err_8 = abs_err_8/ABS(dfdx_exact_8)
! Stampa i risultati
      WRITE (*,'(ES10.2,5ES11.3)') dx_8, dfdx_exact_8, dfdx_4, &
         rel_err_4, dfdx_8, rel_err_8
! Riduce il passo dx
      dx_4 = dx_4*0.5_4
      dx_8 = dx_8*0.5_8
      IF (dx_4 < 1.0E-8) EXIT
  END DO
CONTAINS
   REAL(KIND=4) FUNCTION f4(x) RESULT(y)
      IMPLICIT NONE
      REAL(KIND=4), INTENT(IN) :: x
         y = SIN(2.0_4*x)
   END FUNCTION f4
!
   REAL(KIND=8) FUNCTION f8(x) RESULT(y)
      IMPLICIT NONE
      REAL(KIND=8), INTENT(IN) :: x
         y = SIN(2.0_8*x)
   END FUNCTION f8
!
   REAL(KIND=4) FUNCTION dfdx4(x) RESULT(dydx)
      IMPLICIT NONE
      REAL(KIND=4), INTENT(IN) :: x
         dydx = 2*COS(2*x)
   END FUNCTION dfdx4
!
   REAL(KIND=8) FUNCTION dfdx8(x) RESULT(dydx)
      IMPLICIT NONE
      REAL(KIND=8), INTENT(IN) :: x
         dydx = 2*COS(2*x)
   END FUNCTION dfdx8
END PROGRAM derivate
```

Eseguito su un PC con processore Intel Pentium III con sistema operativo Windows 2000 il

precedente programma fornisce il seguente output:

```
NB: I confronti sono stati effettuati per x=2.3
    dx      df/dx_ex    df/dx_4    rel_err    df/dx_8    rel_err
 1.00E+00 -2.243E-01  1.305E+00  6.819E+00  1.305E+00  6.819E+00
 5.00E-01 -2.243E-01  7.248E-01  4.232E+00  7.248E-01  4.232E+00
 2.50E-01 -2.243E-01  2.715E-01  2.210E+00  2.715E-01  2.210E+00
 1.25E-01 -2.243E-01  2.516E-02  1.112E+00  2.516E-02  1.112E+00
 6.25E-02 -2.243E-01 -9.967E-02  5.556E-01 -9.967E-02  5.556E-01
 3.12E-02 -2.243E-01 -1.621E-01  2.774E-01 -1.621E-01  2.774E-01
 1.56E-02 -2.243E-01 -1.932E-01  1.386E-01 -1.932E-01  1.386E-01
 7.81E-03 -2.243E-01 -2.088E-01  6.927E-02 -2.088E-01  6.926E-02
 3.91E-03 -2.243E-01 -2.165E-01  3.459E-02 -2.165E-01  3.462E-02
 1.95E-03 -2.243E-01 -2.204E-01  1.734E-02 -2.204E-01  1.731E-02
 9.77E-04 -2.243E-01 -2.224E-01  8.556E-03 -2.224E-01  8.653E-03
 4.88E-04 -2.243E-01 -2.233E-01  4.318E-03 -2.233E-01  4.326E-03
 2.44E-04 -2.243E-01 -2.238E-01  2.373E-03 -2.238E-01  2.163E-03
 1.22E-04 -2.243E-01 -2.239E-01  1.748E-03 -2.241E-01  1.082E-03
 6.10E-05 -2.243E-01 -2.242E-01  4.978E-04 -2.242E-01  5.408E-04
 3.05E-05 -2.243E-01 -2.238E-01  2.351E-03 -2.242E-01  2.704E-04
 1.53E-05 -2.243E-01 -2.249E-01  2.649E-03 -2.243E-01  1.352E-04
 7.63E-06 -2.243E-01 -2.232E-01  4.766E-03 -2.243E-01  6.760E-05
 3.81E-06 -2.243E-01 -2.277E-01  1.523E-02 -2.243E-01  3.380E-05
 1.91E-06 -2.243E-01 -2.367E-01  5.523E-02 -2.243E-01  1.690E-05
 9.54E-07 -2.243E-01 -2.234E-01  4.083E-03 -2.243E-01  8.450E-06
 4.77E-07 -2.243E-01 -1.968E-01  1.227E-01 -2.243E-01  4.224E-06
 2.38E-07 -2.243E-01 -1.436E-01  3.600E-01 -2.243E-01  2.112E-06
 1.19E-07 -2.243E-01 -2.871E-01  2.800E-01 -2.243E-01  1.058E-06
 5.96E-08 -2.243E-01  4.258E-01  2.898E+00 -2.243E-01  5.306E-07
 2.98E-08 -2.243E-01  8.515E-01  4.796E+00 -2.243E-01  2.571E-07
 1.49E-08 -2.243E-01  1.703E+00  8.593E+00 -2.243E-01  1.253E-07
```

Parametri di kind per il tipo `INTEGER`

Il meccanismo di selezione del valore di kind sulla base della precisione desiderata è rappresentato, per il tipo intero, dalla funzione intrinseca `SELECTED_INT_KIND`. Così l'espressione:

```
SELECTED_INT_KIND(r)
```

restituisce il valore del parametro di kind per il tipo di dati intero che permetta di rappresentare, *come minimo*, tutti gli interi compresi fra -10^r e $+10^r$.

In molti casi sono disponibili diversi parametri di kind che consentono di lavorare con la precisione richiesta. In questi casi il valore restituito dalla funzione sarà quello a cui compete il range di esponenti *più piccolo*.

Se non è disponibile, per la specifica implementazione, un modello avente la precisione (minima) specificata, la funzione `SELECTED_INT_KIND` restituirà valore `-1` e ogni dichiarazione che usi quel parametro di kind genererà un *errore* al tempo di compilazione.

Ad esempio, l'espressione:

```
SELECTED_INT_KIND(2)
```

definisce un tipo di intero corrispondente alla rappresentazione di kind capace di esprimere valori compresi nell'intervallo $-10^2 \div 10^2$. Un esempio appena più complesso è il seguente:

```
INTEGER :: short, medium, long, toolong
PARAMETER (short=SELECTED_INT_KIND(2), medium=SELECTED_INT_KIND(4), &
           long=SELECTED_INT_KIND(10), toolong=SELECTED_INT_KIND(100))
INTEGER(short) :: a, b, c
INTEGER(medium) :: d, e, f
INTEGER(long) :: g, h, i
INTEGER(toolong) :: l, m, n
```

Le precedenti dichiarazioni specificano che la precisione dovrebbe essere almeno:

- `short`: $(-10^2,\ 10^2)$
- `medium`: $(-10^4,\ 10^4)$
- `long`: $(-10^{10},\ 10^{10})$
- `toolong`: $(-10^{100},\ 10^{100})$

Le costanti di un tipo di kind selezionato sono denotate aggiungendo un *underscore* ("_") seguito dal numero di kind o, meglio, dal nome che (eventualmente) lo identifica:

```
100_2
1238_4
1050_short
54321_long
```

Chiaramente, ad esempio, l'espressione `1050_short` potrebbe non essere valida ove mai lo specificatore `KIND=short` non fosse capace di rappresentare numeri maggiori di `1000`.

Dal momento che spesso tutti gli interi di un programma (o buona parte di essi) si caratterizzano per il medesimo valore del parametro di kind, risulta più efficiente confinare la definizione di tale parametro all'interno di un modulo che sia reso disponibile a tutte le procedure che dichiarino tali interi. In tal modo, una eventuale variazione del valore di questo parametro nel solo file contenente il modulo avrà effetto "automaticamente" sull'intero programma, come nel seguente esempio:

```
MODULE abc
   IMPLICIT NONE
   INTEGER, PARAMETER :: range=SELECTED_INT_KIND(20)
END MODULE

PROGRAM gradi
  USE abc
  IMPLICIT NONE
  INTEGER(KIND=range) :: x, y, z
  ...
  x = 360_range
  y = 180_range
  z = 90_range
  ...
END PROGRAM gradi
```

Parametri di kind per il tipo `COMPLEX`

Anche una variabile complessa può avere un suo parametro di kind; ad esempio, la seguente istruzione dichiara due variabili complesse `v` e `w` aventi parametro di kind pari a tre:

```
COMPLEX(KIND=3) :: v, w
```

Si noti, tuttavia, che dal momento che un numero complesso rappresenta un'entità "composta" consistente in una coppia di numeri *reali*, il valore del selettore di kind, sia esso specificato in maniera implicita o esplicita, è applicato a *entrambi* i componenti. Pertanto, le due variabili complesse `v` e `w` precedentemente definite saranno caratterizzate entrambe da una coppia di componenti (la parte reale e la parte immaginaria) reali e aventi numero di kind pari a tre. Per la stessa ragione è possibile utilizzare la funzione intrinseca `SELECTED_REAL_KIND`, già vista per il tipo reale, per specificare la precisione ed il range degli esponenti in maniera portabile.

Il prossimo programma mostra l'uso dei numeri complessi aventi parametro di kind *non* di default:

```
MODULE costanti
   IMPLICIT NONE
   INTEGER, PARAMETER :: miokind=SELECTED_REAL_KIND(12,70)
END MODULE costanti

PROGRAM prova_complex
  USE costanti
  IMPLICIT NONE
  COMPLEX(KIND=miokind) :: z
  ...
  z = (3.72471778E-45_miokind,723.115798E-56_miokind)
```

```
  ...
END PROGRAM prova_complex
```

Specificare, invece, i parametri di kind per costanti di tipo complesso è un pò più complicato in quanto una costante complessa può essere specificata come una coppia ordinata di valori entrambi reali, entrambi interi, oppure di un valore intero e l'altro reale. Ovviamente, se sono usati degli interi essi vengono convertiti nei reali corrispondenti. E' conveniente trattare i diversi casi in maniera distinta:

- Se *entrambe* le componenti sono reali con parametro di kind di default, la costante complessa che ne deriva ha anch'essa parametro di kind di default. Così, ad esempio, la costante:

  ```
  (1.23,4.56)
  ```

  ha parametro di kind di default al pari delle sue componenti.

- Se *entrambe* le componenti sono valori interi, quale che sia il loro parametro di kind, queste vengono convertite nei valori reali corrispondenti ma con parametri di kind di default, sicché la costante complessa che ne deriva avrà anch'essa parametro di kind di default. Così, ad esempio, le costanti complesse:

  ```
  (1_2,3)
  (1,3)
  (1_2,3_4)
  ```

  hanno tutte parametro di kind di default.

- Se *una sola* delle componenti della costante complessa è una costante intera, il valore di quest'ultima viene convertito in reale con lo stesso parametro di kind della componente reale, e la costante complessa avrà anch'essa *quel* parametro di kind. Pertanto, le costanti complesse:

  ```
  (1,2.0)
  (1,2.0_3)
  ```

  hanno, la prima, parametro di kind di default, la seconda parametro di kind pari a `3`.

- Se *entrambe* le componenti sono reali, il parametro di kind della costante complessa è pari al parametro di kind della componente avente la *precisione decimale maggiore*. Se, tuttavia, le precisioni delle due componenti coincidono, il processore è libero di scegliere il parametro di kind di una qualsiasi delle due componenti. Pertanto, la costante complessa:

  ```
  (1.2_3,3.4_3)
  ```

  ha un parametro di kind pari a `3`, mentre la costante:

  ```
  (1.2_3,3.4_4)
  ```

avrà parametro di kind pari a 3 se il processore fornisce una precisione decimale maggiore per le costanti reali aventi kind uguale a 3 rispetto alle costanti reali con kind uguale a 4; avrà parametro di kind pari a 4 se il processore fornisce una precisione decimale maggiore per le costanti reali aventi kind uguale a 3 rispetto alle costanti reali con kind uguale a 4; avrà parametro di kind pari, indifferentemente, a 3 o a 4 se il processore fornisce la stessa precisione decimale per le costanti reali con kind uguale a 3 e per quelle con kind uguale a 4.

In ogni caso, per evitare queste complicazioni è sempre preferibile utilizzare, per formare una costante complessa, due componenti *reali* aventi il *medesimo* parametro di kind.

Parametri di kind per il tipo LOGICAL

Anche per il tipo LOGICAL un compilatore può prevedere più di un valore per il parametro di kind. Ad esempio, alcuni compilatori prevedono, oltre al tipo di default (che, tipicamente, occupa 4 byte) almeno un altro tipo logico che permette di lavorare con valori "compressi" ad un byte soltanto.

Così come per i tipi numerici, anche per il tipo LOGICAL il valore del parametro di kind segue la costante letterale ed è separata da essa da un underscore, come nel seguente esempio:

```
.TRUE.      ! parametro di kind di default
.FALSE._5   ! parametro di kind pari a 5
```

Parametri di kind per il tipo CHARACTER

Su molte installazioni, oltre al tipo di caratteri di default (che, tipicamente, è l'insieme ASCII) sono previsti altri *kind* che consentono di supportare, ad esempio, i caratteri degli alfabeti stranieri. Un esempio di dichiarazione di variabile di tipo carattere parametrizzato è il seguente:

```
CHARACTER(LEN=5) :: str1
CHARACTER(LEN=5, KIND=1) :: str2
```

in cui la stringa str1 ha tipo di default mentre str2 ha parametro di kind pari a 1, il cui significato dipende dal particolare compilatore.

Contrariamente a quanto avviene per tutti gli altri tipi di dati, per le costanti di tipo carattere, il valore del parametro di kind *precede* il valore della costante, separato da esso da un carattere underscore:

```
1_'aeiou'
```

La spiegazione di ciò risiede nel fatto che il processore necessita di sapere quale "tipo" di caratteri deve utilizzare *prima* di "scrivere" la stringa. Ancora, la seguente espressione:

greek_$\alpha\beta\gamma$

risulta corretta se la costante con nome `greek` è posta pari al valore del parametro di kind che identifica un set di caratteri che supporti i simboli dell'alfabeto greco. Allo stesso modo, se il valore del parametro di kind che identifica tale set di caratteri fosse 12, la costante stringa precedente potrebbe essere scritta equivalentemente come:

`12_`$\alpha\beta\gamma$

Dal momento che compilatori diversi possono assegnare valori differenti (e, comunque, arbitrari) ai parametri di kind, può essere utile "interrogare" il proprio compilatore con un programmino del tipo seguente allo scopo proprio di conoscere quantomeno i parametri di kind degli oggetti di tipo intrinseco di default:

```
PROGRAM parametri_kind
  IMPLICIT NONE
  INTEGER :: int=1
  REAL    :: real=1.0
  CHARACTER(LEN=1) :: char="y"
  LOGICAL :: logic=.TRUE.
  COMPLEX :: compl=(1.0,2.0)
!
  PRINT*, "Parametri di kind di default per i tipi: "
  PRINT*, " INTEGER:   ", KIND(int)
  PRINT*, " REAL :     ", KIND(real)
  PRINT*, " CHARACTER: ", KIND(char)
  PRINT*, " LOGICAL:   ", KIND(logic)
  PRINT*, " COMPLEX:   ", KIND(compl)
END PROGRAM parametri_kind
```

A titolo di esempio, compilando su un PC questo programma con l'Intel Fortran Compiler 4.5 si ottiene il seguente output:

```
 Intel Fortran Compiler 4.5
 Parametri di kind di default per i tipi:
  INTEGER:              4
  REAL :                4
  CHARACTER:            1
  LOGICAL:              4
  COMPLEX:              4
```

mentre l'Imagine1/NAGWare F Compiler (Release 20001002) fornisce il risultato che segue:

```
  INTEGER:    3
  REAL :      1
  CHARACTER:  1
  LOGICAL:    3
  COMPLEX:    1
```

Come si può notare, i valori attribuiti ai parametri di kind sono completamente diversi.

1.8 Tipi di dati derivati

Una caratteristica molto utile del Fortran 90 è la possibilità di definire *tipi di dati derivati* simili alle *strutture* del linguaggio C ed ai *record* del Pascal. Un tipo di dati derivato è un oggetto contenente una combinazione di dati di natura diversa presenti in numero qualsiasi. Ogni componente può essere di un tipo di dati intrinseco o può essere, a sua volta, un elemento di un altro tipo di dati derivato.

Un tipo di dati derivato è definito da una sequenza di istruzioni di dichiarazione preceduta da una istruzione `TYPE` e terminata da una istruzione `END TYPE`. Tra queste istruzioni si collocano le definizioni dei componenti del tipo di dati derivato. L'espressione generale di questa operazione di "assemblaggio" è la seguente:

```
TYPE nome_tipo
   componente_1
   ...
   componente_n
END TYPE [nome_tipo]
```

mentre la definizione di una variabile di questo tipo di dati avviene con una istruzione del tipo:

```
TYPE(nome_tipo) var_1 [, var_2, ...]
```

Ciascun componente di un tipo di dati derivato è detto *struttura* e può essere "assegnato" ed usato indipendente dagli altri come se fosse una comune variabile. Ogni componente è reso accessibile attraverso il *selettore di componente* che consiste nel nome della variabile seguito dal simbolo di percentuale (%) seguito, a sua volta, dal nome del componente:

```
var%componente
```

sicché l'assegnazione di un valore ad un oggetto definito dall'utente può avvenire in maniera selettiva componente×componente:

```
var%componente_1 = valore_1
var%componente_2 = valore_2
...
```

In questo modo, naturalmente, l'ordine in cui vengono specificati i componenti può essere del tutto casuale ed inoltre non è necessario specificare tutti i componenti dell'oggetto, con l'ovvio risultato che il valore di un componente non assegnato rimane inalterato. L'assegnazione, però, può avvenire anche in forma "globale" secondo il meccanismo noto come *costruttore di struttura* che consiste nello specificare tra parentesi tonde la lista dei valori di *tutti* i componenti (rigorosamente nell'ordine in cui essi sono elencati nella definizione del tipo) subito dopo il nome del tipo:

```
nome_tipo(valore_1,valore_2,...)
```

L'esempio che segue mostra come sia semplice definire un tipo di dati e dichiarare un oggetto di un tale tipo:

```
! Definizione di un tipo di dati derivato
TYPE :: obiettivo
   CHARACTER(15) :: name    ! nome del corpo celeste
   REAL          :: ra, dec ! coordinate celesti, in gradi
   INTEGER       :: time    ! tempo di esposizione, in secondi
END TYPE obiettivo

! Dichiarazione di un oggetto di questo nuovo tipo di dati
TYPE(obiettivo) :: mio_obiettivo
```

Si noti la possibilità di "mischiare" liberamente oggetti di tipo numerico e di tipo carattere contrariamente a quanto accade per altre strutture dati come gli array. Naturalmente sarà compito del compilatore quello di provvedere ad una idonea sistemazione fisica in memoria dei diversi componenti per un più efficiente accesso ai registri.

Una caratteristica peculiare dei tipi di dati definiti dall'utente è la possibilità di creare strutture "complesse", vale a dire un componente della struttura può, a sua volta, essere un oggetto di un tipo derivato, come illustrato nel seguente esempio:

```
TYPE punto
   REAL :: x, y
END TYPE punto

TYPE cerchio
   TYPE (punto) :: centro
   REAL :: raggio
END TYPE cerchio

TYPE (cerchio) :: c
```

Nelle righe precedenti si è definita una variabile `c` di tipo `cerchio`. Detta variabile ha formalmente due componenti: il primo componente, `centro`, è esso stesso un tipo di dati derivato che consiste in due componenti reali (`x` ed `y`), il secondo componente di `c` è una variabile "atomica" di tipo reale (`raggio`). A questa variabile `c` può essere assegnato un valore in uno dei due possibili modi:

```
c = cerchio(punto(0.,0.),1.)
```

oppure:

```
c%centro%x = 0.
c%centro%y = 0.
c%raggio = 1.
```

In entrambi i casi, all'oggetto "cerchio" `c` verranno assegnati un "centro" (il punto `(0.,0.)`) ed un "raggio" (di valore `1.`) In particolare, la prima delle due modalità di assegnazione è quella precedentemente definita come *costruttore di struttura*. Come si vede, essa consiste

semplicemente nel nome del tipo derivato seguito dai valori delle componenti elencate nell'ordine in cui sono state dichiarate nella definizione del tipo. Come ulteriore esempio, definito il tipo `persona` al modo seguente:

```
TYPE persona
   CHARACTER(LEN=15) :: nome
   CHARACTER(LEN=15) :: cognome
   INTEGER :: eta
   CHARACTER :: sesso
   CHARACTER(LEN=15) :: telefono
END TYPE persona
```

e dichiarati due oggetti di tipo `persona`:

```
TYPE(persona) :: pers1, pers2
```

queste due entità possono essere inizializzate come segue:

```
pers1 = persona('Luke','Skywalker',31,'M','323-6439')
pers2 = persona('Princess','Leila',24,'F','332-3060')
```

L'altra modalità illustrata per l'assegnazione di un valore ad un oggetto di tipo derivato è quella specificata a mezzo di un *selettore di componente*, che consiste, come visto, nel nome della variabile seguito dal simbolo di percentuale seguito a sua volta dal nome del componente. Ad esempio, volendo modificare l'età di `pers1` si può usare l'istruzione:

```
pers1%eta = 28
```

Il prossimo esempio ha lo scopo di illustrare ulteriormente entrambe le modalità si assegnazione:

```
PROGRAM assegn_strut
! Demo assegnazione di struttura
   IMPLICIT NONE
   INTEGER, PARAMETER :: lunghezza_nome = 18
   TYPE :: isotopo  ! Definizione di tipo derivato
      INTEGER :: numero_di_massa
      REAL :: abbondanza
   END TYPE isotopo
   TYPE :: elemento  ! Definizione di tipo derivato
      CHARACTER (LEN=lunghezza_nome) :: nome_elemento
      CHARACTER (LEN=3) :: simbolo
      INTEGER :: numero_atomico
      REAL :: massa_atomica
      TYPE (isotopo) :: isotopo_principale  ! Tipo definito in precedenza
   END TYPE elemento
   TYPE(elemento) :: Carbonio, Temp_E  ! Dichiarazione di tipo
```

```
   TYPE(isotopo) :: Temp_I = isotopo(999,999.9)
! start program Assign
   READ(*,*) Temp_E
   Carbonio = Temp_E
   Carbonio%isotopo_principale = Temp_I
   WRITE(*,*) Temp_E, Carbonio
   STOP
END PROGRAM assegn_strut
```

Si noti che se una variabile di un tipo derivato è inclusa in una istruzione `WRITE`, i componenti di questa variabile saranno stampati nello stesso ordine in cui sono stati dichiarati nella definizione di tipo. Allo stesso modo, se una tale variabile è inserita in una istruzione `READ`, i suoi componenti devono essere inseriti nel medesimo ordine previsto dalla definizione di tipo. Ciò è particolarmente vero in presenza di istruzioni di input/output formattate. A titolo di esempio, se `pers1` è la variabile precedentemente dichiarata, le seguenti istruzioni:

```
WRITE(*,100) pers1
100 FORMAT(2(1X,A,/),1X,I2,/,1X,A,/,1X,A)
```

forniscono come output la seguente lista:

```
Luke
Skywalker
28
M
323-6439
```

Le variabili di un tipo di dati derivato si comportano esattamente come ogni altra variabile definita in una unità di programma per cui possono essere inizializzate, modificate ed usate in istruzioni di chiamata di procedura come parametri di scambio. Per quanto concerne la loro inizializzazione, mentre il Fortran 90 consente soltanto l'inizializzazione dell'*intera* variabile all'atto della sua *dichiarazione*, ad esempio:

```
TYPE :: mio_tipo
   INTEGER :: m
   INTEGER :: n
   REAL :: x
END TYPE mio_tipo
TYPE(mio_tipo) :: mia_var=mio_tipo(5,10,3.14)
```

il Fortran 95 ha introdotto anche la possibilità di inizializzarne le *componenti* all'atto della *definizione* di tipo. Così l'esempio precedente potrebbe essere riscritto in maniera perfettamente equivalente come:

```
TYPE :: mio_tipo
   INTEGER :: m=5   ! NB: solo Fortran 95
```

```
   INTEGER :: n=10  ! NB: solo Fortran 95
   REAL :: x=3.14   ! NB: solo Fortran 95
END TYPE mio_tipo
TYPE(mio_tipo) :: mia_var
```

Chiaramente è possibile inizializzare anche soltanto alcune componenti, in maniera selettiva, rimandando l'assegnazione delle altre componenti ad una fase successiva:

```
TYPE :: mio_tipo
   INTEGER :: m=5   ! NB: solo Fortran 95
   INTEGER :: n
   REAL :: x=3.14   ! NB: solo Fortran 95
END TYPE mio_tipo
TYPE(mio_tipo) :: mia_var
...
mia_var%n = 10
```

Si osservi che per un oggetto di tipo derivato "innestato" l'inizializzazione di un componente è riconosciuta ad ogni livello. Ossia, considerate le seguenti definizioni di tipo:

```
TYPE :: entry
   REAL    :: value=2.5
   INTEGER :: index
   TYPE(entry), POINTER :: next=>NULL()
END TYPE entry

TYPE node
   INTEGER :: counter
   TYPE(entry) :: element
END TYPE node
```

e la dichiarazione:

```
TYPE(node) :: n
```

il componente `n%element%value` sarà inizializzato al valore `2.5`

Si noti, in conclusione, che quando un compilatore alloca memoria per un oggetto di un tipo di dati derivato, esso è, conformemente allo standard, *non* obbligato a riservare alle varie componenti locazioni di memoria consecutive. Questa "libertà" fu decisa per fare in modo che lo standard del linguaggio consentisse ai compilatori concepiti per architetture parallele di ottimizzare l'allocazione di memoria al fine di garantire le performance migliori in termini di velocità di esecuzione. Tuttavia, esistono dei casi in cui un ordine "stretto" delle componenti è fondamentale. Un esempio è rappresentato dalla eventuale necessità di passare una variabile di un tipo di dati derivato ad una procedura scritta in un altro linguaggio (ad esempio in C). In questi casi, allora, è necessario "forzare" il compilatore affinché esso riservi locazioni di memoria contigue per le componenti del tipo. Ciò può essere fatto inserendo la speciale

istruzione `SEQUENCE` all'interno del blocco riservato alla definizione del tipo, come esemplificato di seguito:

```
TYPE :: vettore
   SEQUENCE
   REAL :: a
   REAL :: b
   REAL :: c
END TYPE
```

1.9 Procedure Intrinseche

Le funzioni intrinseche non sono altro che particolari *sottoprogrammi* che ogni sistema mette a disposizione dell'utente.

Per utilizzare correttamente una qualunque funzione intrinseca è necessario rispettare le seguenti regole:

- Il *numero* degli argomenti attuali deve coincidere con il numero degli argomenti specificato per la funzione, a meno che non siano specificati alcuni parametri *opzionali*.
- Il *tipo* degli argomenti attuali deve essere uno dei tipi previsti per gli argomenti della funzione.

E', inoltre, importante notare che quando una funzione intrinseca può essere usata con argomenti di tipo diverso, il tipo del risultato è determinato da quello degli argomenti attuali. Così, ad esempio, la funzione `ABS` può operare su un argomento di tipo intero, reale in singola o doppia precisione, oppure complesso. Nei primi tre casi la funzione `ABS` fornisce il valore assoluto della quantità specificata quale argomento attuale ed il suo risultato è dello stesso tipo dell'argomento attuale; nell'ultimo caso, ossia se l'argomento attuale è di tipo complesso, la funzione `ABS` fornisce come risultato un valore reale che rappresenta il *modulo* dell'argomento. Con un unico nome, `ABS`, è pertanto possibile indicare operazioni diverse in base al tipo dell'argomento usato. La caratteristica di indicare con lo stesso nome di funzione (*nome generico*) operazioni diverse è comune a molte funzioni intrinseche. L'utilizzazione del nome generico permette, quindi, di selezionare, mediante il tipo degli argomenti, le operazioni che devono essere compiute per ottenere un risultato coerente con i dati forniti alla funzione. Questa flessibilità delle funzioni intrinseche rispetto al tipo dei loro argomenti deriva dal fatto che ciascun nome generico indica, in realtà, una *famiglia* di funzioni, ciascuna delle quali opera su un determinato tipo di argomenti per fornire uno specifico risultato. Il tipo degli argomenti attuali che, in un riferimento alla funzione, accompagnano il nome generico determina quale sottoprogramma della famiglia deve essere effettivamente utilizzato e determina, quindi, anche il tipo della funzione. Nella maggior parte dei casi, ogni sottoprogramma della famiglia identificata da un nome generico può essere utilizzato anche mediante il suo *nome specifico*. In questo caso gli argomenti attuali devono essere del tipo previsto per quel particolare sottoprogramma. Riprendendo l'esempio della funzione `ABS`, essa indica una famiglia di quattro sottoprogrammi le cui caratteristiche possono essere così riassunte:

Nome specifico	*Tipo dell'argomento*	*Tipo del risultato*
IABS	intero	intero
ABS	reale	reale
DABS	doppia precisione	doppia precisione
CABS	complesso	reale

1.9.1 Funzioni per la conversione di tipo

Le funzioni che consentono di passare da una rappresentazione interna di un dato numerico ad un'altra sono dette *funzioni per la conversione di tipo.*

In Fortran 90/95 la conversione di un tipo può avvenire attraverso le seguenti funzioni intrinseche:

`AINT(A[,KIND])`

Restituisce il valore dell'argomento `A` troncato alla parte intera, nella rappresentazione relativa al parametro `KIND` opzionalmente specificato.

L'argomento `A` è di tipo `REAL` mentre l'argomento opzionale `KIND` è di tipo `INTEGER`.

Ad esempio `AINT(3.7)` vale `3.0` mentre `AINT(-3.7)` vale `-3.0`.

`ANINT(A[,KIND])`

Restituisce il valore intero più prossimo all'argomento `A`, nella rappresentazione relativa al parametro `KIND` opzionalmente specificato.

L'argomento `A` è di tipo `REAL` mentre l'argomento opzionale `KIND` è di tipo `INTEGER`.

Ad esempio `AINT(3.7)` vale `4.0` mentre `AINT(-3.7)` vale `-4.0`

`CEILING(A[,KIND])`

Restituisce l'intero più prossimo che sia *maggiore o uguale* ad `A`. L'argomento `A` è di tipo `REAL`, l'argomento opzionale `KIND`, che è disponibile soltanto nel Fortran 95, è di tipo `INTEGER`.

Ad esempio `CEILING(3.7)` è `4` mentre `CEILING(-3.7)` è `-3`.

`CMPLX(X[,Y][,KIND])`

Restituisce il valore complesso con parametro di kind specificato da `KIND` così costruito:

1. Se `X` è di tipo `COMPLEX`, allora `Y` *non* deve essere presente e viene restituito il valore di `X`.
2. Se `X` *non* è di tipo `COMPLEX` e l'argomento `Y` è omesso, viene restituito il valore complesso `(X,0)`.
3. Se `X` *non* è di tipo `COMPLEX` e l'argomento `Y` è presente, viene restituito il valore complesso `(X,Y)`.

L'argomento `X` è di tipo numerico; `Y` può essere di tipo `REAL` o `INTEGER`; `KIND` deve essere di tipo `INTEGER`.

`DBLE(A)`

Converte il valore di `A` in un reale in doppia precisione. L'argomento `A` è di tipo numerico Se, in particolare, esso è di tipo `COMPLEX` soltanto la sua parte reale viene convertita.

`FLOOR(x)`

Restituisce l'intero più prossimo che sia *minore o uguale* ad `A`. L'argomento `A` è di tipo `REAL`, l'argomento opzionale `KIND`, che è disponibile soltanto nel Fortran 95, è di tipo `INTEGER`.

Ad esempio `FLOOR(3.7)` è `3` mentre `FLOOR(-3.7)` è `-4`.

`INT(A[,KIND])`

Tronca l'argomento `A` e lo converte nell'intero equivalente. L'argomento `A` è di tipo numerico, l'argomento opzionale `KIND` è un intero. Se `A` è di tipo `COMPLEX` soltanto la sua parte reale viene convertita. Se `A` è di tipo `INTEGER` la funzione ne cambia unicamente il parametro di kind.

`NINT(A[,KIND])`

Restituisce l'intero più prossimo al valore reale `A`, nella rappresentazione relativa al parametro opzionale `KIND`. L'argomento `A` è di tipo `REAL` mentre l'argomento `KIND` è di tipo `INTEGER`.

Ad esempio `NINT(3.7)` è `4` mentre `NINT(-3.7)` è `-4`.

`REAL(A[,KIND])`

Converte l'argomento `A` nella approssimazione reale corrispondente al parametro di kind eventualmente specificato. L'argomento `A` è di tipo numerico mentre l'argomento opzionale `KIND` è di tipo intero. In particolare, se `A` è di tipo `INTEGER`, all'argomento viene aggiunto il punto decimale; se è di tipo `COMPLEX` la funzione ne restituisce la parte reale; se, infine, è di tipo `REAL` la funzione si limita a cambiarne il parametro di kind conformemente al valore di `KIND`.

1.9.2 Funzioni matematiche

Il Fortran 90/95 prevede un nutrito set di funzioni intrinseche concepite per la realizzazione di calcoli algebrici e trigonometrici.

Per molte funzioni di questa classe sono previste alcune restrizioni sui valori degli argomenti in quanto il dominio delle corrispondenti funzioni matematiche è un sottoinsieme dell'insieme dei numeri reali o dei numeri complessi. Tali restrizioni sono, di solito, diverse per argomenti in singola o doppia precisione o per argomenti complessi; in ogni caso esse restano valide sia quando ci si riferisce ad una funzione intrinseca con un nome specifico sia quando se ne usi il nome generico.

`ABS(A)`

Restituisce il valore assoluto di `A`. L'argomento `A` è di tipo numerico ed il risultato ha sempre lo stesso tipo e parametro di kind dell'argomento, a meno che `A` non sia di tipo `COMPLEX` nel qual caso il risultato è di tipo reale e ha per valore il *modulo* di `A`.

`ACOS(X)`

Restituisce l'arcocoseno di `X`. Il risultato della funzione ha stesso tipo e parametro di kind dell'argomento. L'argomento deve essere di tipo `REAL` (con parametro di kind qualsiasi) ed il suo valore deve essere compreso tra `-1` ed `1`; il valore restituito è, invece, compreso tra `0` e π.

`AIMAG(Z)`

Fornisce la parte immaginaria dell'argomento complesso `Z`. In altre parole, se l'argomento è il valore complesso `(X,Y)`, la funzione restituisce il valore della componente `Y`. Il valore restituito ha tipo `REAL` e stesso parametro di kind dell'argomento.

`ASIN(X)`

Restituisce l'arcoseno di `X`. Il risultato della funzione ha stesso tipo e parametro di kind dell'argomento. L'argomento deve essere di tipo `REAL` (con parametro di kind qualsiasi) ed il suo valore deve essere compreso tra `-1` ed `1`; il valore restituito è compreso tra $-\pi/2$ e $\pi/2$.

`ATAN(X)`

Restituisce l'arcotangente di `X`. Il risultato della funzione ha stesso tipo e parametro di kind dell'argomento. L'argomento deve essere di tipo `REAL` (con parametro di kind qualsiasi); il valore restituito dalla funzione è, invece, compreso tra `0` e π.

`ATAN2(Y,X)`

Restituisce l'arcotangente del numero `Y/X`. Il risultato della funzione ha stesso tipo e parametro di kind dell'argomento. Gli argomenti devono essere entrambi di tipo `REAL` con medesimo parametro di kind e *non* possono avere simultaneamente valore nullo. Il valore restituito dalla funzione è sempre compreso tra $-\pi$ e π.

`CONJC(Z)`

Fornisce il valore complesso coniugato di `Z`, ossia, se l'argomento è il valore complesso `(X,Y)` la funzione restituisce il valore complesso `(X,-Y)`. Il valore restituito ha stesso tipo e parametro di kind dell'argomento.

`COS(X)`

Restituisce il coseno di `X`. Il risultato della funzione ha stesso tipo e parametro di kind dell'argomento. L'argomento può essere di tipo `REAL` o `COMPLEX` e rappresenta la misura di un arco espresso in radianti. Il valore restituito dalla funzione è sempre compreso tra `-1` e `1`.

`COSH(X)`

Restituisce il coseno iperbolico di `X`. Il risultato della funzione ha stesso tipo e parametro di kind dell'argomento. L'argomento deve essere di tipo `REAL` e rappresenta la misura di un arco espresso in radianti.

`DPROD(X,Y)`

Restituisce il valore in doppia precisione del prodotto degli argomenti. Questi ultimi devono essere di tipo `REAL` e avere parametro di kind di default.

`EXP(X)`

Restituisce l'esponenziale di `X`, ossia il valore della funzione e^x. Il valore fornito ha stesso tipo e parametro di kind dell'argomento. `X` può essere di tipo reale o complesso.

`LOG(X)`

Restituisce il logaritmo naturale di `X`, ossia il valore della funzione $\ln x$. Il valore fornito ha stesso tipo e parametro di kind dell'argomento. `X` può essere di tipo `REAL` o `COMPLEX`. Se, in particolare, è di tipo reale, dovrà risultare *maggiore* di zero; se, invece, di tipo complesso dovrà essere *diverso* da zero.

`LOG10(X)`

Restituisce il logaritmo in base 10 di `X`, ossia il valore della funzione $\log x$. Il valore fornito ha stesso tipo e parametro di kind dell'argomento. `X` deve essere di tipo `REAL` ed il suo valore deve essere positivo.

`SIN(X)`

Restituisce il seno di `X`. Il risultato della funzione ha stesso tipo e parametro di kind dell'argomento. L' argomento può essere di tipo `REAL` o `COMPLEX` e rappresenta la misura di un arco espresso in radianti. Il valore restituito dalla funzione è sempre compreso tra `-1` e `1`.

`SINH(X)`

Restituisce il seno iperbolico di `X`. Il risultato della funzione ha stesso tipo e parametro di kind dell'argomento. L'argomento deve essere di tipo `REAL` e rappresenta la misura di un arco espresso in radianti.

`SQRT(X)`

Restituisce la radice quadrata di `X`. Il valore fornito ha stesso tipo e parametro di kind dell'argomento. `X` può essere di tipo `REAL` o `COMPLEX`. Se, in particolare, è di tipo reale, dovrà risultare maggiore o uguale a zero; se, invece, è di tipo complesso la sua *parte reale* dovrà risultare maggiore o uguale a zero, a meno che `X` non sia un immaginario "puro", nel qual caso è la sua *parte immaginaria* che dovrà essere maggiore o uguale a zero.

`TAN(X)`

Restituisce la tangente di `X`. Il risultato della funzione ha stesso tipo e parametro di kind dell'argomento. L'argomento deve essere di tipo `REAL` e rappresenta la misura di un arco espresso in radianti.

`TANH(X)`

Restituisce la tangente iperbolica di `X`. Il risultato della funzione ha stesso tipo e parametro di kind dell'argomento. L'argomento deve essere di tipo `REAL` e rappresenta la misura di un arco espresso in radianti.

1.9.3 Funzioni numeriche

Sono raggruppate con questo nome altre funzioni intrinseche non contemplate fra quelle di conversione di tipo o fra quelle matematiche. Esse, in generale, prevedono argomenti di tipo intero, reale in singola o doppia precisione e forniscono risultati dello stesso tipo degli argomenti.

`DIM(X,Y)`

Restituisce la differenza positiva fra i suoi argomenti, ossia il massimo fra il valore di `X-Y` e `0`. Gli argomenti `X` e `Y` devono essere entrambi di tipo `INTEGER` o di tipo `REAL` e devono caratterizzarsi per il medesimo valore di kind. Il risultato della funzione ha stesso tipo e parametro di kind degli argomenti.

Esempio:

```
INTEGER :: i
REAL(KIND=single) :: r
REAL(KIND=double) :: d
...
i = DIM(10,5)          ! ris:  5
r = DIM(-5.1,3.7)      ! ris:  0.0
d = DIM(10.0D0,-5.0D0) ! ris: 15.0
```

`MAX(A1,A2[,A3,...])`

Restituisce l'argomento di valore massimo. Gli argomenti devono essere *tutti* di tipo `INTEGER` o *tutti* di tipo `REAL` e devono caratterizzarsi per il medesimo parametro di kind. Il risultato fornito dalla funzione ha stesso tipo e parametro di kind dei suoi argomenti.

Esempio:

```
INTEGER :: m1, m2
REAL :: r1, r2
m1 = MAX(5,6,7)           ! ris:  7
m2 = MAX(-5.7,-1.23,-3.8) ! ris: -1
r1 = MAX(-5,-6,-7)        ! ris: -5.0
r2 = MAX(-5.7,1.23,-3.8)  ! ris:  1.23
```

`MIN(A1,A2[,A3,...])`

Restituisce l'argomento di valore minimo. Gli argomenti devono essere *tutti* di tipo `INTEGER` o *tutti* di tipo `REAL` e devono caratterizzarsi per il medesimo parametro di kind. Il risultato fornito dalla funzione ha stesso tipo e parametro di kind dei suoi argomenti.

Esempio:

```
INTEGER :: m1, m2
REAL :: r1, r2
...
m1 = MIN(5,6,7)          ! ris:  5
m2 = MIN(-5.7,1.23,-3.8) ! ris: -5
r1 = MIN(-5,-6,-7)       ! ris: -7.0
r2 = MIN(-5.7,1.23,-3.8) ! ris: -5.7
```

`MOD(A,P)`

Restituisce il *resto intero* della divisione `A/P`, ossia il valore `A-P*INT(A/P)`. Il risultato dipende dal processore nel caso in cui il valore di `P` risulti nullo. Gli argomenti possono essere di tipo `INTEGER` o `REAL` ma devono avere in comune il tipo e il parametro di kind. Il risultato fornito dalla funzione ha stesso tipo e parametro di kind degli argomenti.

Esempio:

```
INTEGER :: i, j
REAL :: r
...
r = MOD(9.0,2.0)  ! ris:  1.0
i = MOD(18,5)     ! ris:  3
j = MOD(-18,5)    ! ris: -3
```

`MODULO(A,P)`

Restituisce il modulo di `A` rispetto a `P`, se `P` è diverso da zero, altrimenti fornisce un risultato dipendente dal processore. Gli argomenti devono essere entrambi di tipo `INTEGER` o entrambi di tipo `REAL`.

Il risultato fornito dalla funzione coincide con quello restituito dalla funzione `MOD` nel caso in cui gli argomenti si presentino con lo stesso segno. In ogni caso detto `R` il risultato, questo può essere calcolato secondo la seguente regola:

1. Se gli argomenti sono interi, il risultato `R` è pari a `A-Q*P` essendo `Q` un intero il cui valore sia tale da garantire la coincidenza dei segni di `P` ed `R` ed il rispetto della condizione $0 \leq$`ABS(R)`$<$`ABS(P)`.

2. Se gli argomenti sono reali, il risultato `R` è pari a `A-FLOOR(A/P)*P`.

Esempio:

```
INTEGER :: i. j
REAL :: r, s
...
i = MODULO(8,5)          ! ris:  3          N.B.: Q =  1
j = MODULO(-8,5)         ! ris:  2          N.B.: Q = -2
i = MODULO(8,-5)         ! ris: -2          N.B.: Q = -2
r = MODULO(7.285,2.35)   ! ris:  0.2350001  N.B.: Q =  3
s = MODULO(7.285,-2.35) ! ris: -2.115      N.B.: Q = -4
```

`SIGN(A,B)`

Restituisce il valore assoluto di `A` con il segno di `B`, ossia il numero: `ABS(A)*(B/ABS(B))`. Gli argomenti possono essere entrambi di tipo `INTEGER` o entrambi di tipo `REAL`. Il risultato ha stesso tipo e parametro di kind degli argomenti.

Si noti che, nel caso particolare in cui il valore di `B` sia `0` ed il processore sia in grado di distinguere fra uno *zero positivo* ed uno *zero negativo*, il valore restituito è proprio il segno del valore nullo di `B`. Quest'ultima evenienza è, però, prevista soltanto dal Fortran 95.

Esempio:

```
REAL :: x1, x2, x3
...
x1 = SIGN(5.2,-3.1)   ! ris: -5.2
x2 = SIGN(-5.2,-3.1)  ! ris: -5.2
x3 = SIGN(-5.2,3.1)   ! ris:  5.2
```

1.9.4 Funzioni di interrogazione

Molte delle funzioni descritte in questa sezione si basano sui modelli implementati dal Fortran per i dati di tipo intero e reale. Il linguaggio, infatti, utilizza dei *modelli numerici* per evitare al programmatore di dover conoscere i dettagli "fisici" della rappresentazione in macchina dei dati numerici.

Il modello Fortran per un intero i è:

$$i = s \times \sum_{k=0}^{q-1} w_k \times r^k$$

dove:

- r è un intero maggiore di 1;
- q è un intero positivo;
- ogni w_k è un intero non negativo minore di r;
- s è il *segno* di i e può valere ± 1.

I valori di r e di q determinano l'insieme dei dati interi supportati dal processore.

Il parametro r è chiamato *radice* (o *base*) del sistema di numerazione utilizzato. Normalmente il suo valore è 2, nel qual caso il valore di q è pari al numero di bit usati dal processore per rappresentare un intero, diminuito di un'unità (un bit, infatti, è usato per rappresentare il segno del numero).

A titolo di esempio, per una tipica rappresentazione di interi a 32 bit su un processore con base 2, il modello intero diventa:

$$i = \pm \sum_{k=0}^{30} w_k \times 2^k$$

dove ciascun w_k può valere 0 o 1.

Il modello Fortran per un reale x, invece, è:

$$x = \begin{cases} 0 \\ s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k} \end{cases}$$

dove:

- b e p sono interi maggiori di 1;
- ciascun f_k è un intero non negativo minore di b (e con, in particolare, f_1 diverso da zero);
- e è un intero giacente fra due estremi, e_{min} ed e_{max};
- s è il *segno* di x e può valere ± 1.

I valori di b e di p , e_{min} ed e_{max} determinano l'insieme dei dati reali supportati dal processore.

Il parametro b è chiamato *radice* (o *base*) del sistema di numerazione utilizzato per rappresentare i numeri reali su un dato processore. Praticamente tutti i moderni processori usano un sistema a 2 bit, sicché b vale 2 e ciascun termine f_k può valere 0 oppure 1 (fatta eccezione per f_1 che può valere solo 1).

I bit utilizzati per rappresentare un numero reale sono divisi in due gruppi: uno per la *mantissa* (la parte intera del valore) e l'altro per l'*esponente*. Per un sistema in base 2, p rappresenta il numero di bit per la mantissa mentre e è compreso in un intervallo rappresentabile con un numero di bit pari a quello dell'esponente diminuito di un'unità.

A titolo di esempio, per una tipica rappresentazione di reali in singola precisione a 32 bit su un processore con base 2, il modello reale diventa:

$$x = \begin{cases} 0 \\ \pm 2^e \times \left(\frac{1}{2} + \sum_{k=2}^{24} f_k \times 2^{-k}\right), \ -126 \leq e \leq 127 \end{cases}$$

dove ciascun f_k può valere 0 o 1.

`BIT_SIZE(I)`

Restituisce il numero di bit occupati dall'argomento `I` nella specifica implementazione, essendo `I` un oggetto di tipo `INTEGER`.

`DIGITS(X)`

Restituisce il numero di cifre significative di `X`. A seconda che l'argomento `X` sia di tipo `REAL` o di tipo `INTEGER`, il numero restituito sarà rappresentativo del valore di q o di p dei modelli di rappresentazione precedentemente citati.

Si noti che questa funzione restituisce il numero di cifre significative nella base di numerazione usata dal processore; se, come spesso accade, questa base è 2, la funzione fornisce il *numero di bit significativi*. Se, invece, si è interessati a conoscere il numero di *cifre decimali significative*, si dovrà fare uso della funzione `PRECISION(X)`.

`EPSILON(X)`

Restituisce quel numero che sommato a `1.0` produce il valore immediatamente successivo fra quelli rappresentabili sulla macchina in uso e per quel particolare parametro di kind. L'argomento deve essere di tipo `REAL` e può avere kind qualsiasi.

`EXPONENT(X)`

Fornisce l'esponente di `X` nella base di numerazione adottata dal processore. L'argomento deve essere di tipo `REAL`. Il valore fornito rappresenta il termine e del modello di rappresentazione dei reali.

`FRACTION(X)`

Fornisce la mantissa della rappresentazione di `X` nel modello di rappresentazione per i reali, ossia il valore del termine $\sum_{k=1}^{p} f_k \times b^{-k}$. L'argomento deve essere di tipo `REAL`. Il risultato ha stesso tipo e kind dell'argomento.

`HUGE(X)`

Restituisce il numero più grande fra quelli rappresentabili in macchina per il tipo ed il kind di `X`. L'argomento può essere di tipo `REAL` o `INTEGER` ed il valore fornito dalla funzione ha lo stesso tipo e kind dell'argomento.

`KIND(X)`

Restituisce il valore di kind dell'argomento. L'argomento `X` può essere di un qualsiasi tipo intrinseco.

`MAXEXPONENT(X)`

Restituisce il massimo valore dell'esponente relativo al tipo ed al kind dell'argomento. Il valore fornito coincide con il termine e_{max} del modello di rappresentazione dei reali. L'argomento deve essere di tipo `REAL`.

Si noti che questa funzione restituisce il massimo esponente nella base di numerazione usata dal processore; se, come spesso accade, questa base è 2, la funzione fornisce il massimo esponente in base 2. Se, invece, si è interessati a conoscere il massimo esponente in base 10, si dovrà fare uso della funzione `RANGE(X)`.

`MINEXPONENT(X)`

Restituisce il minimo valore dell'esponente relativo al tipo ed al kind dell'argomento. Il valore fornito coincide con il termine e_{min} del modello di rappresentazione dei reali. L'argomento deve essere di tipo `REAL`.

`NEAREST(X,S)`

Restituisce il numero più prossimo ad `X` nella direzione `S` fra quelli rappresentabili in macchina. Gli argomenti devono essere entrambi di tipo `REAL` ed inoltre `S` deve risultare diverso da zero; il risultato ha lo stesso tipo e parametro di kind degli argomenti.

`PRECISION(X)`

Fornisce il numero di cifre decimali significative con cui viene rappresentato in macchina un valore dello stesso tipo e kind di `X`. L'argomento `X` può essere di tipo `REAL` o `COMPLEX`.

`RADIX(X)`

Restituisce la base del modello di rappresentazione dei valori aventi tipo e kind di `X`. Poiché la maggior parte dei processori lavora su una base binaria, il valore della funzione sarà quasi sempre 2. Questo è il valore del termine r nel modello di rappresentazione degli interi, o b nel modello di rappresentazione dei reali. L'argomento `X` può essere di tipo `REAL` o `INTEGER`.

`RANGE(X)`

Fornisce il range di esponenti nella base decimale per i valori aventi tipo e kind di `X`. L'argomento può essere di tipo `INTEGER`, `REAL` o `COMPLEX`.

`RRSPACING(X)`

Restituisce il reciproco della spaziatura relativa dei numeri prossimi a `X`. Tale valore è pari a $|\mathtt{X} * b^{-e}| * b^p$, secondo il modello di rappresentazione dei numeri reali precedentemente definito. Il tipo della funzione e del suo argomento è `REAL`.

`SCALE(X,I)`

Restituisce il valore di $\mathtt{X} \times b^{\mathtt{I}}$, dove b è la base del modello utilizzato per la rappresentazione di `X`. La base b può essere determinata a mezzo della funzione `RADIX(X)` ma il suo valore è quasi sempre pari a 2.

L'argomento `X` deve essere di tipo `REAL` mentre l'argomento `I` deve essere di tipo `INTEGER`. Il risultato della funzione ha stesso tipo e parametro di kind di `X`.

`SELECTED_INT_KIND(R)`

Restituisce il numero di kind del *più piccolo* tipo di intero che può rappresentare tutti gli interi compresi tra $-10^{\mathtt{R}}$ e $10^{\mathtt{R}}$. Se nessun kind soddisfa alla condizione richiesta verrà restituito il valore `-1`.

L'argomento `P` deve essere di tipo `INTEGER`. Tale è anche il tipo del risultato.

`SELECTED_REAL_KIND(P,R)`

Restituisce il numero di kind del *più piccolo* tipo di reale che abbia una precisione decimale di almeno `P` cifre ed un range di esponenti di almeno `R` potenze di 10.

Se *non* è possibile ottenere la precisione specificata con qualsiasi tipo di dato reale nel processore in uso verrà restituito il valore `-1`.

Se *non* è possibile ottenere il range specificato con qualsiasi tipo di dato reale nel processore verrà restituito il valore `-2`.

Se non è possibile ottenere né la precisione né il range verrà restituito il valore `-3`.

Gli argomenti `P` ed `R` devono essere di tipo `INTEGER`. Tale è anche il tipo del risultato.

`SET_EXPONENT(X,I)`

Restituisce quel numero la cui parte decimale coincide con quella di `X` ed il cui esponente è pari ad `I`. Se `X` è pari a zero, anche la funzione ha valore nullo.

L'argomento `X` deve essere di tipo `REAL` mentre l'argomento `I` deve essere di tipo `INTEGER`. Il risultato della funzione ha lo stesso tipo di `X`.

`SPACING(X)`

Restituisce la spaziatura assoluta dei numeri prossimi a `X` nel modello utilizzato per la rappresentazione di `X`. Tale valore è pari a b^{e-p}, secondo il modello di rappresentazione dei numeri reali precedentemente definito. Se questo valore risulta *fuori range*, viene fornito il valore della funzione `TINY(X)`.

La funzione e il suo argomento hanno tipo `REAL` e il medesimo parametro di kind.

Il risultato di questa funzione è utile per stabilire un criterio di convergenza indipendente dal processore. Ad esempio, lo scarto massimo ammissibile per una procedura iterativa potrebbe essere imposto pari a dieci volte la spaziatura minima rappresentabile.

`TINY(X)`

Restituisce il più piccolo numero positivo dello stesso tipo e kind di `X`. Il valore fornito è pari a $b^{e_{min}-1}$ in cui b ed e_{min} sono stati precedentemente definiti. L'argomento della funzione deve essere di tipo `REAL` ed il valore fornito ha stesso tipo e kind dell'argomento.

Il seguente programma ha lo scopo di interrogare il processore riguardo i parametri relativa alla rappresentazione di macchina di oggetto di tipo intero o reale. Il programma è, al tempo stesso, un utile esempio per imparare a conoscere le procedure intrinseche analizzate in questa sezione.

```
PROGRAM parametri
  IMPLICIT NONE
  INTEGER, PARAMETER :: i=1
  REAL(KIND(1.E0)), PARAMETER :: r1=1.0
  REAL(KIND(1.D0)), PARAMETER :: r2=1.0
!
  WRITE(*,*) " Tipo INTEGER di default "
  WRITE(*,*) " Kind:               ", KIND(i)
  WRITE(*,*) " No di Cifre:        ", DIGITS(i)
  WRITE(*,*) " Base:               ", RADIX(i)
  WRITE(*,*) " Range di esponenti: ", RANGE(i)
  WRITE(*,*) " Massimo valore:     ", HUGE(i)
  WRITE(*,*) " N. di bit occupati: ", BIT_SIZE(i)
  WRITE(*,*)
  WRITE(*,*) " Tipo REAL in singola precisione "
```

```
  WRITE(*,*) " Kind:                ", KIND(r1)
  WRITE(*,*) " Bit significativi:   ", DIGITS(r1)
  WRITE(*,*) " Max esponente:       ", MAXEXPONENT(r1)
  WRITE(*,*) " Min esponente:       ", MINEXPONENT(r1)
  WRITE(*,*) " Cifre significative:", PRECISION(r1)
  WRITE(*,*) " Base:                ", RADIX(r1)
  WRITE(*,*) " Range di esponenti:  ", RANGE(r1)
  WRITE(*,*) " Epsilon:             ", EPSILON(r1)
  WRITE(*,*) " Spaziatura:          ", SPACING(r1)
  WRITE(*,*) " Minimo valore:       ", TINY(r1)
  WRITE(*,*) " Massimo valore:      ", HUGE(r1)
  WRITE(*,*)
  WRITE(*,*) " Tipo REAL in doppia precisione "
  WRITE(*,*) " Kind:                ", KIND(r2)
  WRITE(*,*) " Bit significativi:   ", DIGITS(r2)
  WRITE(*,*) " Max esponente:       ", MAXEXPONENT(r2)
  WRITE(*,*) " Min esponente:       ", MINEXPONENT(r2)
  WRITE(*,*) " Cifre significative:", PRECISION(r2)
  WRITE(*,*) " Base:                ", RADIX(r2)
  WRITE(*,*) " Range di esponenti:  ", RANGE(r2)
  WRITE(*,*) " Epsilon:             ", EPSILON(r2)
  WRITE(*,*) " Spaziatura:          ", SPACING(r2)
  WRITE(*,*) " Minimo valore:       ", TINY(r2)
  WRITE(*,*) " Massimo valore:      ", HUGE(r2)
  STOP
END PROGRAM parametri
```

Compilato con l'Essential Lahey Fortran 90 Compiler (ELF90) 4.00b ed eseguito su un PC con processore Intel Pentium III, questo programma fornisce il seguente output:

```
 Tipo INTEGER di default
 Kind:                          4
 No di Cifre:                  31
 Base:                          2
 Range di esponenti:            9
 Massimo valore:       2147483647
 N. di bit occupati:           32

 Tipo REAL in singola precisione
 Kind:                          4
 Bit significativi:            24
 Max esponente:                128
 Min esponente:               -125
 Cifre significative:           6
```

```
 Base:                          2
 Range di esponenti:           37
 Epsilon:               1.192093E-07
 Spaziatura:            1.192093E-07
 Minimo valore:         1.175494E-38
 Massimo valore:        3.402823E+38

 Tipo REAL in doppia precisione
 Kind:                          8
 Bit significativi:            53
 Max esponente:              1024
 Min esponente:             -1021
 Cifre significative:          15
 Base:                          2
 Range di esponenti:          307
 Epsilon:               2.22044604925031D-016
 Spaziatura:            2.22044604925031D-016
 Minimo valore:         2.22507385850720D-308
 Massimo valore:        1.79769313486232D+308
```

Naturalmente, su macchine differenti si possono ottenere risultati differenti. Lo stesso dicasi nel caso in cui questo programma venisse compilato con un diverso compilatore, in particolare per quanto riguarda il valore dei numeri di kind che, come è noto, non sono fissati dallo standard ma lasciati alla scelta dei produttori di compilatori.

Un ulteriore esempio di impiego delle funzioni di questo gruppo potrebbe essere la definizione dei parametri di kind di interi con differenti dimensioni e di reali di differenti livelli di precisione. Queste definizioni possono essere efficacemente inserite all'interno di un modulo a fornire, così, una collezione di parametri "globali" utilizzabili da diverse unità di programma. Sebbene lo studio delle unità modulo sarà affrontato solo più avanti, si ritiene comunque utile premettere questo semplice esempio:

```
MODULE numeric_kinds
! costanti con nome per interi a 4, 2 e 1 byte
   INTEGER, PARAMETER ::                 &
            i4b = SELECTED_INT_KIND(9),  &
            i2b = SELECTED_INT_KIND(4),  &
            i1b = SELECTED_INT_KIND(2)
! costanti con nome per reali in singola, doppia e
!  quadrupla precisione
   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
```

Naturalmente su calcolatori che non supportano tutti i parametri di kind elencati alcuni valori potranno essere non rappresentativi. Ad esempio su un PC con processore Intel molti compilatori attribuiscono alla costante `qp`, rappresentativa della *quadrupla precisione*, il valore `-1`.

1.9.5 Funzioni per il trattamento dei caratteri

`ACHAR(I)`

Restituisce il carattere corrispondente alla posizione `I` nella tabella ASCII. Se `I` è compreso tra `0` e `127`, il risultato fornito è effettivamente l'`I_`*mo* carattere della tabella; se, invece, `I` è maggiore di `127`, il risultato sarà dipendente dal processore.

Ad esempio, se `a` è una variabile di tipo `CHARACTER`, l'istruzione:

```
a = ACHAR(87)
```

restituisce il valore `'W'`.

`ADJUSTL(STRING)`

Restituisce la stringa `STRING` *giustificata a sinistra*. In altre parole rimuove i bianchi di testa e li sposta in coda.

Esempio:

```
CHARACTER(LEN=16) :: str
str = ADJUSTL('  Fortran 90  ')   ! restituisce: 'Fortran 90    '
```

`ADJUSTR(STRING)`

Restituisce la stringa `STRING` *giustificata a destra*. In altre parole rimuove i bianchi di coda e li sposta in testa.

Esempio:

```
CHARACTER(LEN=16) :: str
str = ADJUSTR('  Fortran 90  ')   ! restituisce: '    Fortran 90'
```

`CHAR(I[,KIND])`

Restituisce il carattere che occupa la posizione `I` nella sequenza di collating del processore, associato al parametro di kind `KIND` opzionalmente specificato. Gli argomenti `I` e `KIND` devono entrambi di tipo intero. In particolare il valore di `I` deve essere compreso tra `0` ed `n-1`, essendo n il numero di caratteri del sistema di codifica adottato dal processore. La funzione `CHAR` è l'inversa di `ICHAR`.

`IACHAR(C)`

Restituisce la posizione occupata dalla variabile `C`, di tipo `CHARACTER`, nel sistema di codifica ASCII. Il risultato, che è di tipo `INTEGER`, è un valore compreso tra `0` ed `n-1`, essendo `n` il numero di caratteri del sistema di codifica adottata dal processore.

Ad esempio, se `i` è una variabile di tipo `INTEGER`, l'istruzione:

```
i = IACHAR('W')
```

restituisce il valore `87`.

`ICHAR(C)`

Restituisce la posizione occupata dalla variabile `C`, di tipo `CHARACTER`, nel sistema di codifica utilizzato dall'elaboratore, associato al parametro di kind dell'argomento. Il risultato, che è di tipo `INTEGER`, è un valore compreso tra `0` ed *n-1*, essendo `n` il numero di caratteri del sistema di codifica adottata dal processore.

`INDEX(STRING,SET[,BACK])`

Se la variabile logica `BACK` è assente o vale `.FALSE.` restituisce la posizione di partenza di `SET` nella stringa `STRING`. Se la variabile logica `BACK` vale `.TRUE.` allora la funzione restituisce la posizione di partenza dell'*ultima* occorrenza. Il valore ritornato è zero nel caso in cui `SET` *non* sia presente in `STRING`.

Esempio:

```
INTEGER :: i
...
i = INDEX('banana','an',BACK=.TRUE.)  ! ris: i = 4
i = INDEX('banana','an')              ! ris: i = 2
i = INDEX('banana','on')              ! ris: i = 0
```

`LEN(STRING)`

Restituisce la lunghezza della stringa `STRING` comprensiva degli eventuali spazi bianchi di coda.

Esempio:

```
CHARACTER(LEN=11) str
INTEGER :: i
...
i = LEN (str)     ! restituisce 11
i = LEN('Una frase con 6 spazi bianchi ')   ! restituisce 30
```

`LEN_TRIM(STRING)`

Restituisce la lunghezza della stringa `STRING` depurata degli eventuali spazi bianchi di coda. Se `STRING` è completamente "bianca", il valore restituito è zero.

Esempio:

```
CHARACTER(LEN=11) str
INTEGER :: i
...
i = LEN ('   ')      ! restituisce 0
i = LEN('Una frase con 6 spazi bianchi ')   ! restituisce 24
```

`LGE(STRING_A,STRING_B)`

Restituisce il valore `.TRUE.` se `STRING_A >= STRING_B` secondo la sequenza di collating ASCII. Gli argomenti `STRING_A` e `STRING_B` devono essere stringhe di caratteri con kind di default.

Esempio:

```
LOGICAL :: l
...
l = LGE('abc','abd')       ! ris: .FALSE.
l = LGE ('ab','aaaaaaab')  ! ris: .TRUE.
```

`LGT(STRING_A,STRING_B)`

Restituisce il valore `.TRUE.` se `STRING_A > STRING_B` secondo la sequenza di collating ASCII. Gli argomenti `STRING_A` e `STRING_B` devono essere stringhe di caratteri con kind di default.

Esempio:

```
LOGICAL :: l
...
l = LGT('abc','abc')    ! ris: .FALSE.
l = LGT('abc','aabc')   ! ris: .TRUE.
```

`LLE(STRING_A,STRING_B)`

Restituisce il valore `.TRUE.` se `STRING_A <= STRING_B` secondo la sequenza di collating ASCII. Gli argomenti `STRING_A` e `STRING_B` devono essere stringhe di caratteri con kind di default.

Esempio:

```
LOGICAL :: l
...
```

```
l = LLE('ABC','ABC')    ! ris .TRUE.
l = LLE('ABC','AABCD')  ! ris .FALSE.
```

`LLT(STRING_A,STRING_B)`

Restituisce il valore `.TRUE.` se `STRING_A < STRING_B` secondo la sequenza di collating ASCII. Gli argomenti `STRING_A` e `STRING_B` devono essere stringhe di caratteri con kind di default.

Esempio:

```
LOGICAL :: l
...
l = LLT('abc','abd')      ! ris: .FALSE.
l = LLT('aaxyz','abcde')  ! ris: .TRUE.
```

`REPEAT(STRING,NCOPIES)`

Restituisce una stringa formata concatenando `NCOPIES` copie di `STRING` una dietro l'altra. Se `STRING` è completamente "bianca" e `NCOPIES` vale zero, la stringa restituita avrà lunghezza nulla.

Esempio:

```
CHARACTER(LEN=6) ::  str
str = REPEAT('ho',3) !  restituisce la stringa 'hohoho'
```

`SCAN(STRING,SET[,BACK])`

Restituisce la *prima* ricorrenza di un carattere di `SCAN` in `STRING` operando una "scansione" da sinistra. Se il parametro opzionale `BACK` è presente e vale `.TRUE.` la scansione avviene da destra ma il "conteggio" avviene ancora da sinistra. Se `BACK` vale `.TRUE.` la scansione viene effettuata da destra e la funzione restituisce la posizione dell'*ultima* ricorrenza di un carattere di `SCAN` in `STRING`. Se *nessun* carattere di `SET` si trova anche in `STRING`, oppure se la lunghezza di `SET` o di `STRING` è nulla, la funzione restituisce il valore zero.

Gli argomenti `SET` e `STRING` sono stringhe di caratteri mentre l'argomento opzionale `BACK` deve essere di tipo `LOGICAL`.

Esempio:

```
INTEGER :: i
...
i = SCAN ('Fortran','tr')               ! ris: i = 3
i = SCAN ('Fortran','tr', BACK=.TRUE.)  ! ris: i = 5
i = SCAN ('Fortran','for')              ! ris: i = 0
```

`TRIM(STRING)`

Restituisce la stringa `STRING` depurata dagli eventuali spazi bianchi di coda. Se `STRING` è completamente "bianca", il valore restituito è una stringa nulla.

Esempio:

```
!  Questa istruzione stampa il valore 31
WRITE(*,*) LEN( Stringa con caratteri bianchi )
!  Questa istruzione stampa il valore 26
WRITE(*,*) LEN(TRIM( Stringa con caratteri bianchi ))
```

`VERIFY(STRING,SET[,BACK])`

Restituisce la posizione della *prima* ricorrenza di un carattere di `STRING` che *non* è presente in `SET` operando la "scansione" da sinistra. Se la variabile opzionale `BACK` è presente con valore `.TRUE.` allora la funzione restituisce la posizione dell'*ultima* ricorrenza di un carattere di `STRING` che *non* è presente in `SET`. Il risultato della funzione è zero se *tutti* i caratteri di `STRING` sono presenti in `SET`.

Se *tutti* i caratteri di `STRING` sono presenti anche in `SET` oppure se entrambe le stringhe hanno lunghezza nulla, la funzione restituisce il valore zero.

Esempio:

```
INTEGER(4) position
...
position = VERIFY ('banana','nbc')                ! ris: position = 2
position = VERIFY ('banana','nbc', BACK=.TRUE.)   ! ris: position = 6
position = VERIFY ('banana','nbca')               ! ris: position = 0
```

Capitolo 2

Istruzioni di controllo

2.1 Introduzione

La più semplice forma di programma è quella *sequenziale*, in cui una serie di istruzioni devono essere eseguite nell'ordine in cui appaiono nel listato sorgente. Tipicamente un programma sequenziale legge dei dati di input, li elabora, fornisce i risultati e si interrompe. Un esempio molto semplice di programma sequenziale è il seguente che valuta la caduta di tensione ai capi di un elemento di circuito elettrico nota la resistenza e la corrente che lo percorre:

```
PROGRAM Legge_di_Ohm
   IMPLICIT NONE
   REAL :: Voltaggio, Corrente, Resistenza
! start
   WRITE(*,*) " Inserisci la corrente [ampere] e la resistenza [ohm]: "
   READ(*, *) Corrente, Resistenza
   Voltaggio = Corrente * Resistenza
   WRITE(*,*) " Il voltaggio e': ", Voltaggio, " volt"
   WRITE(*,*) " OK. "
   STOP
END PROGRAM Legge_di_Ohm
```

Tuttavia, spesso un programma deve essere in grado di scegliere un'azione appropriata a seconda delle "circostanze", ossia in funzione dello *stato* di alcuni *parametri di controllo*. Ciò può avvenire a mezzo di particolari istruzioni che consentono al programmatore di controllare l'ordine in cui le operazioni devono essere eseguite all'interno di un programma. Queste *istruzioni di controllo* possono essere di due tipi:

- *istruzioni di diramazione*
- *istruzioni di ripetizione*

Al primo gruppo appartengono le *istruzioni di selezione* `IF` e `SELECT CASE`, al secondo gruppo appartengono le *istruzioni cicliche a conteggio* o *condizionali* `DO`.

2.2 Istruzioni di diramazione

Il Fortran 90/95 fornisce due possibili meccanismi che consentono al programmatore di scegliere azioni alternative, e quindi di eseguire delle specifiche sezioni di codice (*blocchi*) saltando altre parti di codice, a seconda del valore assunto da alcune variabili logiche. Questi due meccanismi sono le istruzioni `IF` e `SELECT CASE`.

2.2.1 Istruzione e costrutto `IF`

La più semplice forma di istruzione `IF` è rappresentata dal cosiddetto `IF` *logico* in cui una *singola azione* è basata su una *singola condizione*:

```
IF(espressione_logica) istruzione
```

Solo se *espressione_logica* ha valore `.TRUE.` *istruzione* viene eseguita. Un possibile esempio di `IF` *logico* è il seguente:

```
IF(x<0.0) x = 0.0
```

In questo caso, se la variabile `x` ha valore negativo allora questo viene posto uguale a zero, altrimenti resta inalterato.

Una forma più comune dell'istruzione `IF` è, però, il *costrutto* `IF` il quale specifica che un *intero blocco* di istruzioni sarà eseguito se e solo se una certa condizione logica risulti vera. Esso si presenta nella forma:

```
[nome:]  IF (espressione_logica) THEN
  istruzione_1
  istruzione_2
  ...
END IF [nome]
```

Se *espressione_logica* risulta vera, il programma esegue le istruzioni del blocco compreso tra le istruzioni `IF` e `END IF`.

Si noti che la "coppia" `IF ...THEN` è un'*unica* istruzione Fortran per cui va scritta nella stessa riga, ogni istruzione del blocco `IF` deve occupare una riga distinta ed inoltre l'istruzione `END IF` non può essere etichettata (ossia non può avere un *numero* di istruzione). Si osservi, infine, che l'istruzione `END IF` è *obbligatoria*, ed è richiesta al fine di indicare la fine del processo di selezione.

Un esempio di costrutto `IF` è fornito dal seguente frammento di programma che calcola le radici "reali e distinte" di una equazione algebrica di secondo grado:

```
REAL :: a, b, c    ! coefficienti dell'equazione
REAL :: d, x1, x2  ! discriminante e radici dell'equazione
... ! lettura dei coefficienti
d = b**2-4.*a*c
IF (d > 0.) THEN
```

```
   x1 = (-b-SQRT(d))/(2.*a)
   x2 = (-b+SQRT(d))/(2.*a)
END IF
```

La forma più completa di costrutto `IF` va sotto il nome di *costrutto* `IF THEN ELSE`. Questo costrutto permette di eseguire un blocco di istruzioni se una certa condizione è verificata, oppure un altro blocco se risultano verificate altre condizioni. La struttura di un costrutto `IF THEN ELSE` dipende dal numero di condizioni da considerare, e si presenta nella seguente forma generale:

```
[nome:]  IF (espressione_logica_1) THEN
  blocco_1
ELSE IF (espressione_logica_2) THEN [nome]
  blocco_2
  ...
[ELSE [nome]
  blocco]
END IF [nome]
```

in cui *espressione_logica_i* può essere una qualunque variabile o espressione di tipo `LOGICAL`. Chiaramente ciascun blocco può contenere più istruzioni e, quindi, prevedere più azioni. Il *nome* opzionalmente associato al costrutto può avere una lunghezza massima di 31 caratteri, deve essere unico all'interno di ogni unità di programma e non può coincidere con il nome di una costante o di una variabile all'interno della stessa unità di programma. Se ad un costrutto `IF` è associato un nome, lo stesso nome deve comparire anche nell'istruzione `END IF`, mentre esso è facoltativo nelle clausole `ELSE` ed `ELSE IF`.

Questa forma di costrutto viene usata quando un certo numero di istruzioni dipende dalla medesima condizione. Se *espressione_logica_1* è vera il programma esegue le istruzioni del *blocco_1* per poi saltare alla prima istruzione eseguibile che segue `END IF`; altrimenti il programma verifica lo stato di *espressione_logica_2* e, se questa è vera, esegue le istruzioni del *blocco_2* per saltare alla prima istruzione eseguibile che segue `END IF`. Questo "percorso" si può estendere a un qualsivoglia numero di clausole `ELSE IF`. Se tutte le espressioni logiche sono false, il programma esegue le istruzioni del gruppo `ELSE` per cui la parte `ELSE` agisce come una possibilità di riserva (o di *default*) allo scopo di coprire tutte le altre eventualità.

Le istruzioni `ELSE` ed `ELSE IF`, così come `END IF`, devono occupare righe separate e non possono avere un'etichetta.

Un esempio di uso del costrutto `IF THEN ELSE` è fornito dal seguente programma che valuta la radice quadrata di un numero reale a patto che il radicando sia non minore di zero:

```
PROGRAM radq
! Calcola la radice quadrata di un numero con test sul segno
   IMPLICIT NONE
   REAL :: radicando
!
   WRITE(*,*) " Inserisci un valore: "
```

```
   READ(*,*)  radicando
   IF (radicando>=0.0) THEN
      WRITE(*,*) " La sua radice quadrata vale: "
      WRITE(*,*) SQRT(radicando)
   ELSE
      WRITE(*,*) " Errore: inserito valore negativo! "
   END IF
   STOP
END PROGRAM radq
```

Il programma seguente, invece, valuta il giorno dell'anno in cui cade la Pasqua sfruttando l'algoritmo proposto da Oudin e Tondering. Al fine di comprendere le istruzioni contenute nel programma giova ricordare che la Pasqua cade sempre la domenica seguente il plenilunio successivo all'equinozio di primavera (21 marzo). Si ricorda, inoltre, che si chiama *età della luna* a un dato momento, il numero di giorni trascorsi dall'ultimo novilunio fino a quel momento (per tutti i giorni dell'anno si può dunque esprimere l'età della luna, in generale mediante un numero intero di giorni). Si chiama, invece, *epatta* relativa a un determinato anno l'età della luna al primo gennaio di quell'anno. Convenedo di chiamare 0 il giorno in cui la Luna è nuova, l'epatta può variare tra 0 e 29. Infine, sovrapponendo al ciclo solare annuale il *ciclo metonico* di 19 anni ne consegue che ogni 19 anni, prescindendo da errori di computo, i fenomeni celesti solari e lunari devono ripetersi identicamente.

```
PROGRAM Pasqua
! Scopo: calcolare la data in cui cade la Pasqua in un dato anno
  IMPLICIT NONE
  INTEGER :: year, metcyc, century, error1, error2, day
  INTEGER :: epact, luna, temp
!
  WRITE(*,*) " Inserire l'anno (successivo al 1582): "
  READ(*,*) year
! Calcolo del ciclo metonico
  metcyc = MOD(year,19)+1
  century = (year/100)+1
  error1 = (3*century/4)-12
  error2 = ((8*century+5)/25)-5
  day = (5*Year/4)-error1-10
  temp = 11*metcyc+20+error2-error1
! Calcolo dell'epatta
  epact = MOD(temp,30)
  IF(epact<=0) THEN
     epact = 30+epact
  ENDIF
  IF((epact==25.AND.metcyc>11).OR.epact==24) THEN
     epact = epact+1
```

```
  ENDIF
  luna= 44-epact
  IF (luna<21) THEN
     luna = luna+30
  ENDIF
  luna = luna+7-(MOD(day+luna,7))
  IF (luna>31) THEN
     luna = luna - 31
     WRITE(*,'(A11,I4)') " Nell'anno ", year
     WRITE(*,'(A27,/,1X,I2,A7)') &
           " la Pasqua cade il giorno: ",luna, " Aprile"
  ELSE
     WRITE(*,'(A11,I4)') " Nell'anno ", year
     WRITE(*,'(A27,/,1X,I2,A7)') &
           " la Pasqua cade il giorno: ",luna, " Marzo"
  ENDIF
  STOP
END PROGRAM Pasqua
```

Un esempio di impiego di questo programma può servire a testare la bontà del codice:

```
  Inserire l'anno (successivo al 1582):
2002
 Nell'anno 2002
 la Pasqua cade il giorno:
 31  Marzo
```

Con il prossimo programma, infine, si vuole fornire un ultimo esempio di costrutto `IF THEN ELSE`, questa volta, però, caratterizzato da diverse clausole `ELSE`. Esso rappresenta una generalizzazione dell'esempio già proposto in precedenza e concepito per il calcolo delle radici di una equazione di secondo grado. In questo caso, in aggiunta, viene gestita anche la possibilità di trovare radici complesse e coniugate:

```
PROGRAM eq_2
  IMPLICIT NONE
  REAL :: a, b, c, d  ! coefficienti dell'equazione e discriminante
  COMPLEX :: x1, x2   ! radici complesse dell'equazione
! lettura dei coefficienti
  PRINT*, "Coefficiente a: "
  READ(*,*) a
  PRINT*, "Coefficiente b: "
  READ(*,*) b
  PRINT*, "Coefficiente c: "
  READ(*,*) c
! calcolo del discriminante
```

```
  d = b**2-4.*a*c
  IF (d > 0.) THEN
     PRINT*, "Radici reali e distinte"
     x1 = (-b-SQRT(d))/(2.*a)
     x2 = (-b+SQRT(d))/(2.*a)
  ELSE IF (d == 0.) THEN
     PRINT*, "Radici reali e coincidenti"
     x1 = -b/(2.*a)
     x2 = x1
  ELSE
     PRINT*, "Radici complesse coniugate"
     x1 = CMPLX(-b,-SQRT(-d))/(2.*a)
     x2 = CONJG(x1)
  END IF
  WRITE(*,*) x1, x2
  STOP
END PROGRAM eq_2
```

E' importante osservare che, per motivi di efficienza, sarebbe opportuno che venissero esaminate (e quindi scritte) per prime le espressioni logiche a più elevata probabilità di risultare vere, in modo da rendere probabilisticamente minimo il numero di espressioni da calcolare fino a trovare quella vera. Si noti ancora che non è necessario che le espressioni logiche siano mutuamente esclusive, ma è solo necessario che per $j > i$ *espressione_logica_j possa* risultare vera se *espressione_logica_i* è falsa (ma potrebbe esserlo anche se *espressione_logica_i* risultasse vera). Infine, si consideri che se la valutazione di un'espressione logica è onerosa, essa può essere conveniente sostituirla con un'altra espressione "algebricamente" equivalente che sia, però, di più semplice valutazione. Ad esempio, piuttosto che verificare che risulti $\sqrt{x} > 2$, si può in maniera più efficiente controllare che sia $x > 4$.

Come visto precedentemente, la forma generale dei costrutti `IF` e `IF THEN ELSE` prevede anche la presenza di un *nome* opzionale. Assegnare un nome a questi costrutti può tornare particolarmente utile in presenza di blocchi `IF` *innestati*. Si fa presente che due costrutti `IF` si dicono innestati o "annidati" quando uno di essi giace completamente *all'interno* dell'altro. In questi casi la presenza di un "riferimento" ai singoli blocchi può contribuire a rendere un programma più semplice da gestire e da interpretare, come mostrato dal seguente esempio:

```
outer: IF (x > 0.0) THEN
   ...
ELSE outer
   inner: IF (y < 0.0) THEN
      ...
   ENDIF inner
ENDIF outer
```

E' sempre consigliabile assegnare dei nomi ai costrutti `IF` quando questi sono annidati in quanto il nome indica esplicitamente a quale costrutto `IF` è associata una determinata clausola `ELSE IF`

o una istruzione END IF. Se i costrutti non hanno nome, il compilatore associa sempre una istruzione END IF all'*ultima* istruzione IF; naturalmente questo metodo funziona bene con un programma scritto correttamente ma può provocare messaggi di errore abbastanza ambigui in presenza di qualche errore (basti pensare al caso in cui, durante una sessione di editing, una istruzione END IF venga cancellata accidentalmente).

2.2.2 Il costrutto SELECT CASE

Il *costrutto* SELECT CASE è un'altra forma di costrutto di diramazione e fornisce un'alternativa ad una serie di istruzioni IF THEN ELSE. Esso permette al programmatore di selezionare un particolare blocco di istruzioni da eseguire in funzione del valore assunto da una variabile di controllo di tipo intero o carattere, o da un'espressione logica. La sua forma generale è:

```
[nome:]  SELECT CASE(espressione)
CASE(valore) [selettore_1]
  blocco_1
CASE(valore) [selettore_2]
  blocco_2
  ...
[CASE DEFAULT
  blocco]
END SELECT [nome]
```

Il risultato di *espressione* può essere di tipo INTEGER, CHARACTER o LOGICAL; la variabile *selettore_i* *deve* essere dello stesso tipo di *espressione* e può essere una combinazione di:

- un singolo valore di tipo INTEGER, CHARACTER o LOGICAL;
- *min* : col significato di un qualsiasi valore dello stesso tipo di *min* e che sia di questo "non minore";
- :*max* col significato di un qualsiasi valore dello stesso tipo di *min* e che sia di questo "non maggiore";
- *min*:*max* col significato di un qualsiasi valore compreso tra i due limiti *min* e *max*;
- una *lista* formata da una combinazione qualsiasi delle forme precedenti, separate da virgole.

Se il valore di *espressione* è compreso nell'intervallo dei valori di *selettore_1* verranno eseguite le istruzioni del *blocco_1*; analogamente, se il valore di *espressione* è compreso nell'intervallo dei valori di *selettore_2* verranno eseguite le istruzioni del *blocco_2*, e così via. Tale criterio si applica a tutti gli altri "casi" del costrutto. La clausola CASE DEFAULT è opzionale e copre tutti gli altri possibili valori di *espressione* non previsti dalle altre istruzioni CASE. Se il blocco DEFAULT viene omesso e il valore di *espressione* non è compreso in un intervallo dei selettori, non sarà eseguita alcuna istruzione.

Una cosa importante da tenere a mente è che tutti i selettori devono *escludersi a vicenda*, nel senso che uno stesso valore *non* può apparire in più di un selettore.

Un altro esempio interessante è rappresentato dal seguente programma il quale rappresenta semplicemente una variazione dell'esempio già visto in precedenza per il calcolo delle radici reali di una equazione algebrica di secondo grado; l'unica differenza è rappresentata dal fatto che in questo caso le condizioni sul segno del discriminante vengono gestite da un costrutto `SELECT CASE` piuttosto che da un costrutto `IF THEN ELSE`:

```
PROGRAM eq2bis
! Scopo: calcolare le (eventuali) radici reali di una
!        equazione algebrica di secondo grado
  IMPLICIT NONE
  REAL, PARAMETER :: epsilon=1.0e-8
! Coefficienti dell'equazione e discriminante
  REAL :: a, b, c, delta
! Radici reali
  REAL :: root1, root2
! Variabili di appoggio
  REAL :: rt_d, two_a
! Selettore del costrutto SELECT CASE
  INTEGER :: selettore
!
  PRINT*, "Introdurre i coefficienti:"
  READ(*,*) a,b,c
  delta = b**2 - 4.0*a*c
! Valuta il selettore
  selettore = INT(delta/epsilon)
  SELECT CASE (selettore)
  CASE(1:)  ! Radici reali e distinte
     rt_d  = SQRT(delta)
     two_a = 2.0*a
     root1 = (-b+rt_d)/two_a
     root2 = (-b-rt_d)/two_a
     PRINT*, "Le radici sono: ",root1,root2
  CASE(0)   ! Radici reali e coincidenti
     root1 = -b / 2.0 * a
     PRINT*, "La radice (di molteplicita' 2) e': ",root1
  CASE(:-1) ! Radici complesse e coniugate
     PRINT*, "L'equazione non ammette radici reali"
  END SELECT
END PROGRAM eq2bis
```

Un esempio di impiego del precedente programma potrebbe essere il seguente:

```
C:\MYPROG>eq2case
```

```
 Introdurre i coefficienti:
1 5 3
 Le radici sono:  -0.6972244      -4.302776
```

Così come per il costrutto IF, è possibile assegnare un *nome* anche ad un costrutto SELECT CASE. Tale nome deve essere unico all'interno di ogni unità di programma e non può coincidere con il nome di una variabile o di una costante usata all'interno della stessa unità di programma. Se a un costrutto SELECT CASE è associato un nome, lo stesso nome deve comparire anche nell'istruzione END SELECT, mentre la sua presenza è facoltativa nelle clausole CASE.

A titolo di esempio, si consideri il seguente frammento di programma:

```
INTEGER :: mese
stagioni: SELECT CASE(mese)
CASE(3:5)
   WRITE(*,*) 'Primavera'
CASE(6:8)
   WRITE(*,*) 'Estate'
CASE(9:11)
   WRITE(*,*) 'Autunno'
CASE(12,1,2)
   WRITE(*,*) 'Inverno'
CASE DEFAULT
   WRITE(*,*) "Non e' un mese"
END SELECT stagioni
```

L'esempio precedente stampa il nome della stagione associata al mese indicato. Se il valore del mese non appartiene all'intervallo intero $1 \div 12$ viene applicata l'istruzione di default e viene, pertanto, stampato un messaggio di errore, altrimenti viene applicata una delle istruzioni CASE. Si noti che *non* esiste alcun ordine preferenziale nei valori di una istruzione CASE.

Così come i costrutti IF, anche i costrutti SELECT CASE possono essere innestati allo scopo di gestire condizioni più complesse, come esemplificato dal seguente insieme di istruzioni:

```
zero: SELECT CASE(n)
   CASE DEFAULT zero
      altri: SELECT CASE(n)
      CASE(:-1)
         segno = -1
      CASE(1:) altri
         segno = 1
      END SELECT altri
   CASE(0)
      segno = 0
END SELECT zero
```

che hanno lo scopo di assegnare alla variabile `segno` il valore `+1`, `-1` oppure `0` a seconda che il valore `n` sia, rispettivamente, positivo, negativo o nullo. Un esempio lievemente più "corposo" è il seguente:

```
PROGRAM demo_CASE
! *** Sezione dichiarativa ***
  IMPLICIT NONE
  INTEGER :: n, m, type_of_op, op
! *** Sezione Esecutiva ***
  PRINT*, "Inserisci i valori di m ed n"
  READ(*,*)m, n
  PRINT*, "Scegli il tipo di operazione"
  PRINT*, "1 per aritmetica"
  PRINT*, "2 per relazionale"
  READ(*,*), type_of_op
  SELECT CASE(type_of_op)
  CASE(1)  ! Operazione aritmetica
     PRINT*, "Scegli l'operazione da effettuare"
     PRINT*, "1 per l'addizione"
     PRINT*, "2 per la moltiplicazione"
     READ(*,*), op
     SELECT CASE(op)
     CASE(1)
        PRINT*, n+m
     CASE(2)
        PRINT*, n*m
     CASE DEFAULT
        PRINT*, "Operazione non valida"
     END SELECT
  CASE(2)
     PRINT*, "Scegli l'operazione da effettuare"
     PRINT*, "1 per il check n == m"
     PRINT*, "2 per il check n <= m"
     READ(*,*), op
     SELECT CASE(op)
     CASE(1)
        PRINT*, n == m
     CASE(2)
        PRINT*, n <= m
     CASE DEFAULT
       PRINT*, "Operazione non valida"
     END SELECT
  CASE DEFAULT
       PRINT*, "Tipo di operazione non valida"
```

```
  END SELECT
END PROGRAM demo_CASE
```

Analogamente è possibile inserire uno o più costrutti `IF` (eventualmente innestati) all'interno di un costrutto `SELECT CASE`, come dimostra il seguente esempio:

```
PROGRAM giorni
! Scopo: Fornisce il numero di giorni presenti in un mese
IMPLICIT NONE
   INTEGER :: month, year, ndays
! start
   WRITE(*,*) " Inserisci l'anno: "
   READ(*,*) year
   WRITE(*,*) " Inserisci il mese: "
   READ(*,*) month
   SELECT CASE(month)
      CASE(11,4,6,9)    ! 30 gg a Nov con Apr, Giu e Set...
         ndays = 30
      CASE(2)           ! ...di 28 ce n'e' uno...
         IF(MODULO(year,4)==0) THEN
            ndays = 29  ! (anno bisestile)
         ELSE
            ndays = 28  ! (anno non bisestile)
         END IF
      CASE DEFAULT      ! ...tutti gli altri ne han 31
         ndays = 31
   END SELECT
   WRITE(*,*) " Il numero di giorni del mese no. ", month, " e': ", ndays
   STOP
END PROGRAM giorni
```

2.2.3 L'istruzione `GOTO`

L'*istruzione di salto incondizionato* `GOTO` è un'istruzione di diramazione che può essere utilizzata per trasferire il controllo ad un punto ben preciso del programma. Essa si presenta nella forma:

```
GOTO etichetta
```

In altre parole l'istruzione `GOTO` trasferisce il controllo del programma all'istruzione caratterizzata da *etichetta*. Ad esempio:

```
IF(x<5)  GOTO 100
...
100  y = x**2
```

L'istruzione GOTO dovrebbe essere sempre evitata in quanto i programmi che ne fanno uso sono notoriamente difficili da interpretare e da mantenere. D'altra parte in queste note essa è stata inserita soltanto per motivi di completezza. In ogni caso si tenga presente che è vietato "saltare" all'interno di un ciclo o di un costrutto di diramazione a partire da un punto esterno; al contrario, da un ciclo o da un blocco IF è lecito saltarne fuori mediante un'istruzione GOTO.

2.3 Istruzioni di ripetizione

Un'importante caratteristica di ogni linguaggio di programmazione è l'abilità di ripetere, in maniera "controllata", l'esecuzione di un intero blocco di istruzioni.

In Fortran 90/95 è il costrutto (o *ciclo*) DO che consente al programmatore di ripetere un insieme di istruzioni. Il costrutto DO ha la seguente forma generale:

```
[nome:]  DO [clausola di controllo]
   blocco
END DO [nome]
```

A seconda del tipo di *clausola di controllo* il costrutto DO può avere una delle seguenti forme:

- *ciclo a conteggio*
- *ciclo a condizione*

La differenza fra questi due tipi di cicli consiste essenzialmente nel modo in cui avviene il controllo del numero di ripetizioni. Il ciclo a conteggio, infatti, viene eseguito un numero determinato di volte e questo numero è noto *prima* che il ciclo venga eseguito. Il corpo di un ciclo a condizione, invece, viene ripetuto finché non venga soddisfatta una particolare condizione per cui il numero di volte che verrà eseguito *non* è noto a priori.

Di entrambi questi meccanismi si parlerà diffusamente nei prossimi paragrafi.

2.3.1 Il ciclo a conteggio

Il *ciclo a conteggio* (o *ciclo for*) usa la *clausola_di_controllo* in forma di *contatore* per ripetere le istruzioni del blocco un numero *predeterminato* di volte:

```
[nome:]  DO indice = inizio, fine [, passo]
   blocco
END DO [nome]
```

In questo caso, dunque, la clausola di controllo assume la forma:

```
indice = inizio, fine [, passo]
```

in cui:

- *indice* è una variabile intera usata come *contatore*;
- *inizio* è una variabile (o una espressione) intera indicante il valore iniziale di *indice*;
- *fine* è una variabile (o una espressione) intera indicante il valore finale di *indice*.
- *passo* è una variabile (o una espressione) intera indicante lo step di cui la variabile *indice* viene incrementata. Questo step è opzionale ed ha valore di default unitario.

All'avvio del ciclo, la variabile contatore *indice* assume il valore *inizio*, alla seconda iterazione (dopo, cioè, una prima esecuzione delle istruzioni del *blocco*) *indice* ha valore pari a *inizio*+*passo* (ovvero *inizio*+1 se *passo* è omesso), e così via fino all'ultima iterazione quando assumerà il valore *fine* (oppure il massimo valore intero minore di *fine* ma tale che se incrementato di *passo* risulterebbe maggiore di *fine*).

Il numero di volte che le istruzioni vengono eseguite può essere facilmente calcolato come:

$$\texttt{MAX}((\mathit{fine} + \mathit{passo} - \mathit{inizio})/\mathit{passo}, 0)$$

Pertanto, se il valore di *fine* è minore del valore di *inizio* e *passo* risulta positivo, allora *indice* assumerà valore zero ed il ciclo non avrà luogo.

Esempi di applicazione del ciclo a conteggio sono offerte dai seguenti frammenti di codice:

```
tutti: DO i = 1, 10
   WRITE(*,*) i      ! stampa i numeri da 1 a 10
END DO tutti

pari: DO j = 10,2,-2
   WRITE(*,*) j      ! stampa i numeri pari 10,8,6,4,2
END DO pari
```

Il programmino che segue, invece, stampa la tabellina del 5:

```
PROGRAM tabellina
  IMPLICIT NONE
  INTEGER ::  i, j=5
  PRINT 100, "Tabellina del", j
  DO i=1,10
    PRINT 200, j,"per",i,"=",i*j
  END DO
100 FORMAT(1X,A13,1X,I2)
200 FORMAT(1X,I2,1X,A3,1X,I2,1X,A1,1X,I2)
END PROGRAM tabellina
```

ed infatti il suo output è:

```
 Tabellina del  5
  5 per  1 =  5
  5 per  2 = 10
  5 per  3 = 15
  5 per  4 = 20
  5 per  5 = 25
  5 per  6 = 30
  5 per  7 = 35
  5 per  8 = 40
  5 per  9 = 45
  5 per 10 = 50
```

Un altro esempio di applicazione del ciclo a conteggio è rappresentato dal seguente programma che stima il valore di π con il *Metodo Monte Carlo*. Si rammenta brevemente il metodo in esame. Generate t_n coppie (x, y) di numeri random compresi tra 0 e 1 e detto i_n il numero di punti che cadono nel quarto di cerchio inscritto nel quadrato di lato unitario, si approssima il valore d π mediante il rapporto $4i_n/t_n$.

```
PROGRAM pi_greco
! Stima il valore di pi greco con il metodo Monte Carlo
  IMPLICIT NONE
  REAL :: x, y           ! coordinate del colpo
  INTEGER :: in=0, tn=0  ! colpi 'a segno' e colpi totali
  INTEGER :: loop        ! variabile di controllo del ciclo
! start
  WRITE(*,*) " Quanti 'colpi'? "
  READ(*,*) tn
  DO loop = 1,tn
     CALL RANDOM_NUMBER(HARVEST=x)
     CALL RANDOM_NUMBER(HARVEST=y)
     IF(x**2 + y**2 < 1.0) in = in+1.0
  END DO
  WRITE(*,*) "Il valore stimato di pi_grego e': ", 4.0*in/tn
  STOP
END PROGRAM pi_greco
```

La *procedura intrinseca* `RANDOM_NUMBER` utilizzata nel programma serve a generare numeri pseudo-random nell'intervallo `0÷1` e sarà descritta nel capitolo 5. Vale la pena controllare la bontà del risultato prodotto:

```
C:\MYPROG>pi_greco
  Quanti 'colpi'?
7000
 Il valore stimato di pi_grego e':    3.145714
```

Una cosa su cui fare particolarmente attenzione è la necessità di *non* modificare mai la variabile indice all'interno di un ciclo. Modificare il valore di questa variabile può portare a risultati inaspettati come, ad esempio, il generarsi di un *ciclo infinito*. Il seguente programma mostra un esempio di tale tipo di errore:

```
PROGRAM errore
  IMPLICIT NONE
  INTEGER :: i
  DO i=1,4
     ...
     i = 2    ! Errore: e' vietato modificare il contatore
     ...
  END DO
END PROGRAM errore
```

Come si nota, ad ogni passaggio del ciclo il valore della variabile `i` viene impostato a `2` per cui l'indice del ciclo non potrà mai essere maggiore di `4`. Si noti, però, che quasi tutti i compilatori sono in grado di riconoscere un errore di questo tipo e generano, di conseguenza, un errore in fase di compilazione.

Due cicli si dicono *annidati* quando un ciclo giace interamente all'interno dell'altro. Un "nido" di `DO` è, ad esempio, il seguente:

```
DO i=1,3
   DO j=1,3
      prod = i*j
      PRINT*, prod
   END DO
END DO
```

In questo esempio, il ciclo *esterno* assegna il valore `1` all'indice `i`, poi viene eseguito il ciclo *interno*, per tre volte, con l'indice `j` che assume in successione i valori `1`, `2` e `3`. Quando il ciclo interno è stato completato, il ciclo esterno assegna il valore `2` alla variabile `i` e viene eseguito nuovamente il ciclo interno. Il processo viene iterato fino ad esaurimento del ciclo esterno. In ogni caso il ciclo interno viene sempre eseguito completamente prima che l'indice del ciclo esterno venga incrementato.

Dal momento che non è possibile modificare la variabile indice all'interno del corpo di un ciclo `DO` non è possibile utilizzare lo stesso indice per due cicli annidati in quanto il ciclo interno tenterebbe di cambiare il valore dell'indice all'interno del corpo del ciclo esterno.

In presenza di cicli annidati può tornare particolarmente utile assegnare un nome a ciascun ciclo così da avere un "riferimento" a ciascuno di essi. Ciò può facilitare la manutenzione di programmi complessi e l'individuazione di eventuali errori commessi in fase di editing. Le regole da seguire per assegnare un nome ad un ciclo sono le stesse viste con i costrutti `IF`: il nome può essere lungo al massimo 31 caratteri, deve essere unico all'interno della unità di programma che ospita il ciclo, non può coincidere con il nome di una variabile o di una costante usata all'interno della stessa unità di programma. Inoltre, se il nome è presente nell'istruzione `DO`

esso deve essere presente anche nell'istruzione `END DO` mentre è facoltativo nelle istruzioni `EXIT` e `CYCLE` (di queste ultime si parlerà fra poco). Ad esempio, il prossimo frammento fornisce un esempio di cicli annidati in maniera non corretta; avendo assegnato loro un nome il compilatore è in grado di segnalare l'errore, errore che in presenza di cicli senza nome si sarebbe presentato in maniera più ambigua soltanto in fase di esecuzione:

```
esterno: DO i=1,m
   ...
   interno: DO j=1,n
      ...
END DO esterno
   END DO interno
```

Le operazioni eseguite da un ciclo iterativo possono essere gestite anche attraverso due ulteriori istruzioni di controllo: l'istruzione `EXIT` e l'istruzione `CYCLE`.

L'istruzione `EXIT`

L'istruzione `EXIT` rappresenta una utile *facility* per trasferire il controllo all'esterno del ciclo `DO` prima che venga raggiunta l'istruzione `END DO` o che venga completata l'esecuzione della iterazione corrente. La sua sintassi è:

```
EXIT [nome]
```

L'istruzione `EXIT` può essere inserita in un qualsiasi punto all'interno di un ciclo `DO`. Dopo che una istruzione `EXIT` sia stata eseguita l'esecuzione del ciclo viene interrotta e il controllo viene trasferito alla prima istruzione immediatamente successiva al ciclo.

Si noti che, in presenza di cicli `DO` innestati, un'istruzione `EXIT` *senza nome* ha effetto sempre sul ciclo *più interno* mentre un'istruzione `EXIT` *con nome* ha effetto sul ciclo avente *quel* nome, anche se non è quello più interno.

Il costrutto che segue rappresenta un esempio di impiego dell'istruzione `EXIT`. Il suo scopo è di valutare la `somma` di `5` elementi inseriti dall'utente con l'obbligo, però, di arrestarsi nel caso in cui il valore della somma raggiunga o superi una `soglia` massima:

```
somma = 0.0
DO i = 1,5
   READ(*,*) add
   somma = somma+add
   IF(somma>=soglia) EXIT
END DO
PRINT*, "Fatto!"
```

L'istruzione `CYCLE`

L'istruzione `CYCLE` permette di trasferire il controllo all'inizio del ciclo e di avviare una nuova iterazione senza che venga eseguita (o solo completata) l'iterazione corrente. La sua sintassi è:

```
CYCLE [nome]
```

L'istruzione `CYCLE` può essere inserita in un qualsiasi punto all'interno di un ciclo `DO`. Quando una istruzione `CYCLE` viene eseguita, tutte le istruzioni che la seguono all'interno del ciclo vengono ignorate ed il controllo ritorna all'inizio del ciclo. L'indice del ciclo viene quindi incrementato e l'esecuzione riprende dall'inizio purché l'indice non abbia raggiunto il suo valore limite.

Si noti che, in presenza di cicli `DO` innestati, un'istruzione `CYCLE` *senza nome* ha effetto sempre sul ciclo *più interno* mentre un'istruzione `CYCLE` *con nome* ha effetto sul ciclo avente *quel* nome, anche se non è quello più interno.

Il costrutto che segue mostra un esempio di funzionamento dell'istruzione `CYCLE`:

```
DO i = 1,5
   PRINT*, i
   IF(i>3) CYCLE
   PRINT*, i
END DO
PRINT*, "Fatto!"
```

Il suo output è, chiaramente, il seguente:

```
     1
     1
     2
     2
     3
     3
     4
     5
Fatto!
```

Un esempio di utilizzo comparato delle istruzioni `EXIT` e `CYCLE` è, infine, fornito dal seguente frammento di codice:

```
loop1: DO i = 1,10
   loop2:  DO j = 1,10
      n = n+1
      IF (n > nmax) EXIT loop1   ! esce dal ciclo esterno
      CYCLE loop2                ! salta la prossima istruzione
      n = n*2
   END DO loop2
END DO loop1
```

Si noti come la presenza di cicli con nome aumenti la leggibilità di un costrutto in presenza di istruzioni `EXIT` o `CYCLE`.

La variabile di controllo all'uscita del ciclo

Un problema che insorge dopo aver lasciato un ciclo a conteggio riguarda il valore della variabile contatore. A tal riguardo è bene notare che:

- La variabile indice del ciclo `DO` viene fissata al suo valore iniziale *prima* che venga presa qualsiasi decisione se eseguire o meno il ciclo.
- La variabile indice viene aggiornata alla fine di ogni passaggio attraverso il ciclo, ma sempre *prima* che sia deciso se eseguire o meno un altro passaggio.
- La variabile indice *non* viene modificata in nessun altro punto del programma.

Se ne deduce che, se si esce dal ciclo prima che esso sia stato completato, il contatore conserva il valore che aveva durante l'ultimo passaggio (ossia il suo valore corrente). Se il ciclo viene completato, il contatore viene incrementato prima che la condizione logica di chiusura del ciclo venga verificata per cui esso assumerà il valore che avrebbe (teoricamente) avuto al passaggio successivo.

Può tornare utile chiarire il concetto con qualche esempio.

```
! Esempio 1: ciclo interrotto
DO i=1,5
   PRINT*, i
   IF(i==3) EXIT
END DO
PRINT*, i
```

Il suo output è:

```
1
2
3
3
```

```
! Esempio 2: ciclo completato
DO i=1,5
   PRINT*, i
   IF(i==3) EXIT
END DO
PRINT*, i
```

Il suo output è:

```
1
2
3
4
5
6
```

Allo stesso modo è facile comprendere che, dato il seguente costrutto:

```
! Esempio 3: cicli innestati completati
DO i=1,10
   DO j=i,i**2
      DO k=j+1,i*10
         sum = i+j+k
      END DO
   END DO
END DO
PRINT*, i, j, k, sum
```

il risultato da esso prodotto sarà il record:

```
 11 101 101 209
```

2.3.2 Il ciclo a condizione

Un *ciclo a condizione* rappresenta un particolare blocco di istruzioni che viene ripetuto finché non risulti soddisfatta una particolare condizione logica. La forma generale di un ciclo a condizione è:

```
[nome:]  DO
   ...
   IF (espressione_logica) EXIT
   ...
END DO [nome]
```

Il blocco di istruzioni compreso fra `DO` e `END DO` viene ripetuto finché l'*espressione logica* non diventi vera; a questo punto viene eseguita l'istruzione `EXIT` e il controllo delle operazioni passa alla prima istruzione che segue `END DO`. Come si nota dallo schema precedente, normalmente il blocco di istruzioni del corpo del ciclo è diviso in più "frammenti" separati da una o più istruzioni "di uscita".

L'istruzione `IF` può trovarsi in qualsiasi punto all'interno del corpo del ciclo e viene eseguita ad ogni passaggio. Chiaramente, se *espressione_logica* risulta vera già la prima volta che viene raggiunto il ciclo le istruzioni che, nel blocco, seguono l'istruzione `IF` non vengono mai eseguite.

In generale, il test sulla *condizione_logica* può essere inserito in un punto qualsiasi fra le istruzioni `DO` e `END DO`, ma le varianti più comuni sono quelle note come *ciclo a condizione iniziale* e *ciclo a condizione finale*, utilizzate spesso nella programmazione di problemi di carattere iterativo. Inoltre, è possibile inserire anche più di una istruzione `IF` all'interno dello stesso loop, ciascuna con la relativa clausola di uscita, sebbene un "eccesso" di clausole `CYCLE` ed `EXIT` può rendere il codice più difficile da leggere e da gestire. Infine, per l'attribuzione di un *nome* al costrutto `DO` condizionale, valgono le stesse regole e le raccomandazioni già discusse a proposito del ciclo `DO` a conteggio.

Il seguente frammento di programma illustra in maniera estremamente semplice sia la possibilità di interrompere il ciclo con un'istruzione `EXIT`, sia la possibilità di "saltare" anche solo una parte del blocco `DO` mediante una o più istruzioni `CYCLE`:

```
i = 0; j = 0
DO
   i = i+1
   j = j+1
   PRINT*, i
   IF (i > 4) EXIT
   IF (j > 3) CYCLE
   i = i+2
END DO
```

L'output prodotto dal precedente costrutto è, chiaramente:

```
1
4
7
```

Ciclo a condizione iniziale

Il *ciclo a condizione iniziale* (anche detto ciclo *while*) si caratterizza per la presenza del test sulla *condizione_logica* in testa al blocco di istruzioni, subito dopo l'istruzione `DO`, allo scopo di determinare se le istruzioni del corpo del ciclo debbano essere o meno eseguite *almeno* una volta. La forma generale di tale ciclo è la seguente:

```
[nome:]  DO
  IF (espressione_logica) EXIT
  blocco_di_istruzioni
END DO [nome]
```

Un esempio di ciclo a condizione iniziale è il seguente:

```
REAL :: diff, old_val, new_val
...
diff = ... ! assegna a diff un valore di partenza
DO
   IF (diff < 0.001) EXIT
   ...
   new_val = ... !calcola il nuovo valore
   diff = old_val-new_val
   old_val = new_val
END DO
```

L'effetto di questo ciclo è quello di eseguire le istruzioni del blocco fintanto che il valore di `diff` risulti minore di `0.001`. Chiaramente se già il valore di partenza di `diff` risulta maggiore o uguale di `0.001` le istruzioni del ciclo non verranno mai eseguite.

Il programma che segue, concepito per il calcolo del massimo comun divisore di due interi con il *metodo di Euclide*, rappresenta un ulteriore esempio di ciclo a condizione iniziale:

```
! ------------------------------------------------------------
! Questo programma calcola il MCD di due numeri interi positivi
! usando il metodo di Euclide. Dati i1 e i2 con i1 >= i2, il
! metodo di Euclide lavora come segue:
! - si divide i1 per i2 valutando il resto intero tmp;
! - se tmp vale zero:
!      il MCM e' pari proprio a i2;
! - se tmp e' diverso da zero:
!      i2 diventa i1 e tmp diventa i2;
!      si ripete il procedimento fino a che temp non e' zero.
! ------------------------------------------------------------

PROGRAM  MCD
  IMPLICIT  NONE
  INTEGER   :: i1, i2, tmp

  WRITE(*,*) "Inserisci due interi positivi: "
  READ(*,*)  i1,i2
  IF (i1 < i2) THEN       ! poiche' deve risultare i1 >= i2, i dati
      tmp = i1            ! vengono sottoposti a swap se i1 < i2
      i1 = i2
      i2 = tmp
  END IF
  tmp = MOD(i1,i2)       ! tmp adesso e' il resto intero
  DO                     ! ora certamente risulta i1 <= i2
     IF (tmp == 0) EXIT  ! se tmp vale zero il MCM e' i2
     i1 = i2             ! altrimenti i2 diventa i1...
     i2 = tmp            ! tmp diventa i2...
     tmp = MOD(i1,i2)    ! viene  rivalutato il resto intero...
  END DO                 ! il procedimento e' ripetuto
  WRITE(*,*) "Il MCD e' ", i2
END PROGRAM  MCD
```

Una variante di questo programma, concepita per la scomposizione di un valore intero in fattori primi, è riportata di seguito. Il programma che segue mostra anche un interessante impiego di costrutti `DO` innestati:

```
PROGRAM Factorize
```

```
   IMPLICIT NONE
   INTEGER  :: Input
   INTEGER  :: Divisore
   INTEGER  :: Count
!
   WRITE(*,*) "Inserire l'intero da scomporre"
   READ(*,*) Input
   Count = 0
   DO
      IF ((MOD(Input,2)/=0).OR.(Input==1)) EXIT
      Count  = Count+1
      WRITE(*,*) "Fattore n. ", Count , ": ", 2
      Input = Input / 2
   END DO
   Divisore = 3
   DO
      IF (Divisore>Input) EXIT
      DO
         IF ((MOD(Input,Divisore)/=0).OR.(Input==1)) EXIT
         Count = Count + 1
         WRITE(*,*) "Fattore n. ", Count,": ", Divisore
         Input = Input/Divisore
      END DO
      Divisore = Divisore+2
   END DO
END PROGRAM Factorize
```

Un esempio di utilizzo del programma è il seguente:

```
C:\MYPROG>factorize
 Inserire l'intero da scomporre
60
 Fattore n.           1 :           2
 Fattore n.           2 :           2
 Fattore n.           3 :           3
 Fattore n.           4 :           5
```

Il ciclo a condizione iniziale può essere costruito anche a mezzo del *costrutto* `DO WHILE` la cui sintassi è:

```
[nome:]  DO WHILE(espressione_logica)
  blocco_di_istruzioni
END DO [nome]
```

e la cui funzione è quella di eseguire ripetutamente le istruzioni del blocco fintanto che il valore di *espressione_logica* rimane `.TRUE.`

Il blocco di istruzioni interno al ciclo è eseguito secondo quest'ordine:

1. Viene valutato il valore di *espressione_logica*:
   - se questo valore è `.FALSE.`, nessuna istruzione del blocco viene eseguita e il controllo passa alla prima istruzione che segue l'istruzione `END DO`;
   - se questo valore è `.TRUE.`, viene eseguito il blocco di istruzioni a partire dalla prima istruzione che segue l'istruzione `DO WHILE`.
2. Quando viene raggiunta l'istruzione finale del ciclo, il controllo ritorna all'istruzione `DO WHILE`, *espressione_logica* viene valutata di nuovo ed il ciclo riprende.

Naturalmente anche il loop `DO WHILE` può essere "generalizzato" prevedendo una o più istruzioni `CYCLE` e `EXIT`.

Un esempio di utilizzo del costrutto `DO WHILE` è il seguente:

```
CHARACTER(1) input
input = ' '
DO WHILE((input /= 'n') .AND. (input /= 'y'))
   WRITE(*,'(A)') 'Enter y or n: '
   READ(*,'(A)') input
END DO
```

ed ha lo scopo di sospendere l'esecuzione del programma in cui esso è inserito fino a che l'utente non introduca da tastiera il carattere `y` oppure il carattere `n`.

Ciclo a condizione finale

Il *ciclo a condizione iniziale* (anche detto ciclo *repeat-until*) si caratterizza per la presenza del test sulla *condizione_logica* in coda al blocco di istruzioni, subito prima dell'istruzione `END DO`. In tal caso, ovviamente, le istruzioni del corpo del ciclo verranno sempre eseguite *almeno* una volta. La forma generale di tale ciclo è la seguente:

```
[nome:]  DO
  blocco_di_istruzioni
  IF (espressione_logica) EXIT
END DO [nome]
```

Un esempio di ciclo a condizione finale è rappresentato dal seguente frammento di programma:

```
INTEGER :: n
...
READ(*,*) n
DO
   n = n**3
```

```
   PRINT*, n
   IF (n > 1000) EXIT
END DO
PRINT*, n
```

la cui funzione è quella di elevare al cubo ciascun numero introdotto fino a che il cubo del valore inserito non superi il valore 1000, nel qual caso il ciclo si arresta e viene stampato il risultato.

Un ulteriore esempio di ciclo a condizione finale è rappresentato dal programma che segue il cui scopo è quello di costruire una tabella di conversione della temperatura dalla scala Celsius alle scale Kelvin, Fahrenheit e Réaumur, rispettivamente. L'intervallo di temperatura prescelto varia da $0°C$ e $100°C$ essendo queste le temperature standard del ghiaccio fondente e dell'acqua bollente; la ripartizione dT dell'intervallo è, invece, scelto dall'utente in fase di esecuzione. Al fine di favorire la comprensione del codice che segue, si ricordano le formule di conversione dalla scala Celsius alle altre tre:

$$T[K] = T[°C] + 273.15$$
$$T[°F] = \frac{9}{5}T[°C] + 32$$
$$T[°R] = 0.8 \times T[°C]$$

```
PROGRAM temperatura
! Scopo: convertire temperature da gradi Celsius a gradi Fahrenheit,
!        Reaumur e nella scala Kelvin
  IMPLICIT NONE
  REAL :: T_c   ! temperatura in gradi Celsius
  REAL :: T_k   ! temperatura in Kelvin
  REAL :: T_f   ! temperatura in gradi Fahrenheit
  REAL :: T_r   ! temperatura in gradi Reaumur
  REAL :: dT    ! incremento nella temperatura
  REAL, PARAMETER :: Tmin_c=0.0  ! temperatura di congelamento dell'acqua
  REAL, PARAMETER :: Tmax_c=100. ! temperatura di ebollizione dell'acqua
  REAL, PARAMETER :: cost=1.8    ! fattore 9/5 della convers. Cels->Fahr

  WRITE(*,*) "Incremento nella T: "; READ(*,*) dT
  WRITE(*,*) "   Celsius        Kelvin       Fahrenheit       Reaumur  "
  WRITE(*,*) "  ==========    ==========    ===========     ========== "
  T_c = Tmin_c
  DO
     T_f = cost*T_c + 32.
     T_k = 273.15 + T_c
     T_r = 0.8*T_c
     WRITE(*,*) T_c, T_k, T_f, T_r
     T_c = T_c+dT
     IF (T_c > Tmax_c) EXIT
  END DO
END PROGRAM temperatura
```

Si riporta, per motivi di completezza, anche un esempio di impiego del programma:

```
 Intervallo di temperatura:
10
    Celsius        Kelvin       Fahrenheit       Reaumur
   ==========    ==========    ============     ==========
   0.000000E+00   273.1500       32.00000       0.000000E+00
   10.00000       283.1500       50.00000       8.000000
   20.00000       293.1500       68.00000       16.00000
   30.00000       303.1500       86.00000       24.00000
   40.00000       313.1500       104.0000       32.00000
   50.00000       323.1500       122.0000       40.00000
   60.00000       333.1500       140.0000       48.00000
   70.00000       343.1500       158.0000       56.00000
   80.00000       353.1500       176.0000       64.00000
   90.00000       363.1500       194.0000       72.00000
   100.0000       373.1500       212.0000       80.00000
```

Un altro ineressante esempio di ciclo a condizione finale è rappresentato dal programma che segue il quale valuta la *media aritmetica* e la *deviazione standard* di un insieme di valori reali introdotti da tastiera. Il numero di questi valori non è noto a priori per cui al termine di ogni passaggio del ciclo viene chiesto all'utente se il computo deve continuare oppure se si è introdotto l'ultimo campione. Si ricordano, brevemente, le definizioni di media e di deviazione standard:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (x_i - \bar{x})^2}$$

```
PROGRAM MeansAndStandardDeviation
! Variabili usate:
!    mean  - media aritmetica
!    sd    - deviazione standard
!    ssq   - somma quadratica
!    x     - generico valore di input
!    w,r,s - variabili di lavoro
   IMPLICIT NONE
   REAL :: mean=0.0, ssq=0.0, x, w, sd, r, s
   INTEGER :: n=0
   CHARACTER(LEN=1) :: risp
   ext: DO
      n = n+1
      WRITE(*,*) "Introduci il valore: "
```

```
         READ(*,*) x
         w = x-mean
         r = REAL(n-1)
         s = REAL(n)
         mean = (r*mean+x)/s
         ssq = ssq+w*w*r/s
         int: DO
            WRITE(*,*) "Un altro valore?"
            READ(*,*) risp
            SELECT CASE(risp)
               CASE("s","S")
                  EXIT int
               CASE("n","N")
                  EXIT ext
               CASE DEFAULT
                  CYCLE int
               END SELECT
            END DO int
      END DO ext
      sd = (ssq/r)**0.5
      PRINT*, " Media Aritmetica:    ", mean
      PRINT*, " Deviazione Standard: ", sd
   END PROGRAM MeansAndStandardDeviation
```

2.3.3 Cicli e istruzioni di diramazione innestati

I cicli `DO` con indice o condizionali e i costrutti di diramazione possono essere innestati tra loro anche nella forme più disparate a formare strutture particolarmente complesse. In ogni caso, però va ricordato che due costrutti risultano innestati correttamente se e solo se quello "esterno" giace *completamente* in quello "interno". Con gli esempi che seguono si vuole fornire solo un piccolo campionario delle svariate possibilità che si possono realizzare:

```
INTEGER :: i
...
DO i = 1, 5
   IF (i == 3) THEN
      CYCLE
   ELSE
      WRITE(*,*)  i
   END IF
END DO
```

L'esempio che segue mostra come sia possibile "controllare" l'input di un programma in maniera interattiva. Infatti questo frammento di codice continua a chiedere all'utente di introdurre un valore di input e l'esecuzione della parte rimanente di programma riprende solo quando il dato introdotto soddisfa alla condizione logica (ossia quando l'utente introduce un intero positivo).

```
INTEGER :: range
...
DO
   WRITE(*,*) "Inserisci un intero positivo: "
   READ(*,*) range
   IF (range <= 0) THEN
      WRITE(*,*) "L'input non e' corretto"
      CYCLE
   END IF
   ... ! istruzioni che coinvolgono range
END DO
```

Il programma che segue valuta tutti i numeri di Armstrong nell'intervallo [0-999]. Si ricorda che un numero è detto di Armstrong se la somma dei cubi delle cifre di cui si compone è pari al valore del numero stesso. Così, ad esempio, 371 è un numero di Armstrong in quanto $3^3+7^3+1^3 = 371$.

```
PROGRAM  ArmstrongNumber
  IMPLICIT  NONE
  INTEGER :: a, b, c          ! le tre cifre
  INTEGER :: abc, a3b3c3      ! il numero e la sua 'somma cubica'
  INTEGER :: Count            ! un contatore
!
  Count = 0
  DO a = 0, 9                          ! ciclo sulla cifra di sinistra
     DO b = 0, 9                       ! ciclo sulla cifra centrale
        DO c = 0, 9                    ! ciclo sulla cifra di destra
           abc = a*100 + b*10 + c      ! il numero
           a3b3c3 = a**3 + b**3 + c**3 ! la somma dei cubi delle cifre
           IF (abc == a3b3c3) THEN     ! se risultano uguali...
              Count = Count + 1        ! ...il numero e' di Armstrong
              WRITE(*,100)  'Armstrong number ', Count, ': ', abc
           END IF
        END DO
     END DO
  END DO
100 FORMAT(1X,A17,I2,A2,I3)
END PROGRAM  ArmstrongNumber
```

Per completezza si riporta anche l'output di questo programma:

```
 Armstrong number  1:   0
 Armstrong number  2:   1
 Armstrong number  3: 153
 Armstrong number  4: 370
 Armstrong number  5: 371
 Armstrong number  6: 407
```

Può insorgere confusione quando le istruzioni `EXIT` e `CYCLE` vengono utilizzate all'interno di cicli innestati. In tali casi è altamente raccomandato attribuire un nome a ciascun ciclo. A titolo di esempio si possono considerare i seguenti programmi:

```
! Esempio n. 1
PROGRAM media_voti
  IMPLICIT NONE
  REAL :: voto, media
  INTEGER :: stid, loop
  INTEGER, PARAMETER :: n_esami=5
  mainloop: DO
     WRITE(*,*) 'Inserisci la matricola:'
     READ(*,*) stid
     IF (stid==0) EXIT mainloop
     media=0
     innerloop: DO loop=1,n_esami
        WRITE(*,*) 'Inserisci il voto:'
        READ(*,*) voto
        IF (voto==0) CYCLE innerloop
        flag: IF (voto<0) THEN
           WRITE(*,*) 'Voto non corretto! Ripeti'
           CYCLE mainloop
        END IF flag
        media = media+voto
     END DO innerloop
     media = media/n_esami
     WRITE(*,*) 'Il voto medio dello studente ', stid, ' è: ', media
  END DO mainloop
END PROGRAM media_voti

! Esempio n. 2
PROGRAM  sommatoria
  IMPLICIT NONE
  INTEGER :: sum=0
  outer: DO
     inner: DO
        READ(*,*) j
```

```
        IF (j<=i) THEN
           PRINT*, 'Il valore deve essere maggiore di: ', i
           CYCLE inner
        END IF
        sum = sum+j
        IF (sum>500) EXIT outer
        IF (sum>100) EXIT inner
      END DO inner
      sum = sum+i
      i = i+10
  END DO outer
  PRINT*, 'sum = ',sum
END PROGRAM  sommatoria
```

2.3.4 Osservazioni sull'uso dei cicli

E' errore assai comune (e spesso molto meno ovvio di quanto si crede) quello di eseguire uno o più calcoli non "indiciati" all'interno di un ciclo. Il seguente loop, ad esempio, è molto poco efficiente in quanto esegue il medesimo calcolo (la divisione per `4` della variabile `pi` e la radice quadrata del risultato) `10` volte:

```
DO i = 1,10
   x(i) = x(i)*SQRT(pi/4.0)  ! NB: x è un array: gli array saranno
END DO                       !     trattati nel prossimo capitolo
```

mentre sarebbe molto più efficiente calcolare il termine `SQRT(pi/4.0)` una sola volta all'esterno del ciclo dal momento che esso non dipende dall'indice `i` del loop (si parla in tal caso di *loop invariant code motion*). Così, ad esempio, è senz'altro preferibile sostituire il ciclo precedente con questo:

```
factor = SQRT(pi/4.0)
DO i=1,10
   x(i) = x(i)*factor
END DO
```

o, meglio ancora, con un'assegnazione in forma globale (quest'ultima soluzione, come si vedrà nel capitolo 3, è senz'altro da preferirsi ogni qualvolta le circostanze lo permettano):

```
factor = SQRT(pi/4.0)
x = x*factor
```

C'è da dire che molti compilatori hanno la possibilità di eseguire questo tipo di ottimizzazione sul codice sorgente ogni qualvolta la sostituzione appaia "ovvia". Esistono, però, dei casi in cui l'interpretazione dell'*invarianza* di un fattore non è immediata, soprattutto in presenza di funzioni definite dall'utente. Il prossimo ciclo ne è un esempio:

```
DO i=1,n
   x(i) = x(i)*myfunc(n)
END DO
```

In questo caso il compilatore potrebbe non essere in grado di stabilire se il risultato della funzione `myfunc` dipenda o meno da `i` o da `x`. In casi come questo deve essere responsabilità del programmatore sapere se il risultato del fattore è invariante o meno e, in caso affermativo, spostarne il calcolo all'esterno del ciclo.

Un'altra caratteristica che rende spesso poco efficiente un ciclo è il fatto che, al suo interno, un grande sforzo di calcolo è dedicato a compiti banali, compiti che spesso possono essere rimossi semplicemente cambiando l'utilizzo di alcune variabili. Ad esempio, per rimuovere l'assegnazione `i = j`, basta trattare, nella successiva parte di codice, la variabile `j` come se fosse niente di più che un *alias* di `i`. Un esempio, concepito per la valutazione dei numeri di Fibonacci, aiuterà a meglio comprendere questo concetto. I numeri di Fibonacci, si ricorda, sono definiti, in forma ricorsiva, al modo seguente:

$$\text{fib}(n) = \begin{cases} 1, \text{ se } n = 1 \text{ o se } n = 2 \\ \text{fib}(n-1) + \text{fib}(n-2), \text{ se } n > 2 \end{cases}$$

Un semplice programma per il loro calcolo potrebbe essere:

```
PROGRAM fibonacci
  IMPLICIT NONE
  INTEGER :: n     ! valore di input
  INTEGER :: fib   ! numero di Fibonacci
  INTEGER :: a, b  ! variabili di appoggio
  PRINT*, " Inserisci n > 2: "
  READ(*,*) n
  a = 1
  b = 1
  DO i = 3,n
     c = a+b
     a = b
     b = c
  END DO
  fib = c
  PRINT*, fib
END PROGRAM fibonacci
```

Come si può notare, per ogni "spazzata" del ciclo vengono effettuate tre assegnazioni, di cui due "banali"; ebbene, queste ultime possono essere facilmente rimosse e rimpiazzate da un'unica istruzione di assegnazione, al modo seguente:

```
PROGRAM fibonacci
  IMPLICIT NONE
```

```
  INTEGER :: n     ! valore di input
  INTEGER :: fib   ! numero di Fibonacci
  INTEGER :: a, b  ! variabili di appoggio
  PRINT*, " Inserisci n > 2: "
  READ(*,*) n
  a = 1
  b = 1
  DO i = 1,n/2-1
     a = a+b
     b = b + a
  END DO
  IF (MOD(n,2)==1) b = b+a
  fib = b
  PRINT*, fib
END PROGRAM fibonacci
```

In tal modo si è facilmente migliorata l'efficienza del codice. Si noti, a proposito del secondo procedimento, che il valore finale dell'indice `i` del ciclo è stato dimezzato in quanto, come si può facilmente verificare, a monte della prima istruzione di assegnazione, i valori di `a` e `b` sono, rispettivamente, `fib(2×i-1)` e `fib(2×i)`.

Un'ultima doverosa osservazione da fare riguarda i cicli a condizione. Si guardi il seguente programmino:

```
PROGRAM prova_sum
  IMPLICIT NONE
  INTEGER :: n
  REAL :: somma=0.0
  REAL, PARAMETER :: base=2.0/3.0
  n = 1
  DO
     somma = somma+(base)**n
     PRINT*, i, somma
     IF(somma>=5.0) EXIT
     n = n+1
  END DO
  PRINT*, "Convergenza raggiunta dopo ",n," iterazioni"
  STOP
END PROGRAM prova_sum
```

il cui scopo, come appare evidente, è quello di determinare la somma parziale della serie numerica

$$\sum_{n=1}^{\infty} \left(\frac{2}{3}\right)^n$$

arrestando la somma quando il valore di quest'ultima raggiunge o supera il valore `5.0`. Ebbene, il problema grosso è che la serie in esame converge al valore `2.0` per cui il precedente ciclo

realizzerà un *loop infinito* senza possibilità di poter essere arrestato se non interrompendo "dall'esterno" l'esecuzione del programma. Al fine di evitare simili problemi può essere utile (quando non addirittura necessario) accoppiare alla condizione logica di chiusura del ciclo (sia esso di tipo a condizione iniziale o finale) una ulteriore clausola che limiti ad un "tetto" massimo il numero di iterazioni che è possibile compiere. In tal modo si perviene a codici più sicuri sebbene a prezzo di un incremento del numero di condizioni logiche da analizzare ad ogni iterazione. A chiarimento di quanto detto, il codice precedente potrebbe essere riscritto al modo che segue:

```
PROGRAM prova_sum2
  IMPLICIT NONE
  INTEGER :: n
  INTEGER, PARAMETER :: nmax=100
  REAL :: somma=0.0
  REAL, PARAMETER :: base=2.0/3.0
  LOGICAL :: converge=.FALSE.
  n = 1
  DO
     somma = somma+(base)**i
     PRINT*, n, somma
     n = n+1
     IF(somma>=5.0 .OR. n<=nmax) THEN
        converge = .TRUE.
        EXIT
     END IF
  END DO
  IF(converge) THEN
     PRINT*, "Convergenza raggiunta dopo ",n," iterazioni"
  ELSE
     PRINT*, "Superato il numero massimo di iterazioni"
  END IF
  STOP
END PROGRAM prova_sum2
```

Una ulteriore possibilità potrebbe essere quella di sostituire direttamente il ciclo a condizione con un ciclo a conteggio, in questo modo:

```
PROGRAM prova_sum3
  IMPLICIT NONE
  INTEGER :: n
  INTEGER, PARAMETER :: nmax=100
  REAL :: somma=0.0
  REAL, PARAMETER :: base=2.0/3.0
  LOGICAL :: converge=.FALSE.
```

```
    DO n=1,nmax
       somma = somma+(base)**i
       PRINT*, n, somma
       IF(somma>=5.0) THEN
          converge = .TRUE.
          EXIT
       END IF
    END DO
    IF(converge) THEN
       PRINT*, "Convergenza raggiunta dopo ",n," iterazioni"
    ELSE
       PRINT*, "Superato il numero massimo di iterazioni"
    END IF
    STOP
  END PROGRAM prova_sum3
```

Quest'ultima possibilità rappresenta nulla di più che una variante alla soluzione precedente, non esistendo alcuna differenza sostanziale fra i due tipi di approcci.

Il problema di determinare una condizione di arresto per un ciclo a condizione è, in molti casi, assai critico per il motivo opposto, cioè si può commettere l'errore di arrestare anzitempo le iterazioni. In questa problematica va riconosciuto che, purtroppo, assai spesso nella pratica computazionale si è costretti a procedere in modo empirico seguendo criteri pratici il che può comportare pericoli piuttosto subdoli. Ad esempio è facilmente prevedibile che qualora da un processo iterativo scaturisse una successione di valori del tipo:

$$
\begin{aligned}
&S_1 = 1 \\
&S_2 = 1 + \frac{1}{2} \\
&S_3 = 1 + \frac{1}{2} + \frac{1}{3} \\
&\ldots \\
&S_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n}
\end{aligned}
$$

la tecnica di controllo adoperata per decidere l'arresto potrebbe essere (incautamente) quella di verificare che l'*n_mo* valore della successione sia minore di un dato *epsilon*. In questo modo non solo la successione in esame fornirebbe un valore fortemente dipendente dalla accuratezza richiesta (il valore di ϵ) ma completamente errato visto che la successione in realtà diverge. E' d'altronde vero che in situazioni in cui addirittura *non vi è convergenza* per il problema iterativo ma solo avvicinamento alla soluzione le tecniche di arresto consentono la "cattura" di una utile approssimazione alla soluzione. Questo mostra l'importanza di garantire, se possibile, *a priori* la convergenza con considerazioni teoriche e tuttavia la questione della scelta di buoni criteri di arresto rimane, in molti casi, quantomai aperta.

Capitolo 3

Array

3.1 Variabili dimensionate

Il linguaggio Fortran consente di associare un nome simbolico, oltre che ad una *singola* locazione di memoria, anche ad un *insieme ordinato* di locazioni consecutive atte a contenere dati *tutti* del *medesimo tipo*. Un tale insieme viene detto *variabile dimensionata* o, più comunemente, **array**, e le locazioni che lo costituiscono sono dette *elementi della variabile dimensionata*. Ognuno di tali elementi può essere individuato specificandone in modo opportuno la posizione tramite uno o più *indici*.

Gli array rappresentano uno strumento estremamente conciso ed efficace per svolgere operazioni ripetitive, essendo possibile svolgere calcoli su specifici elementi o su tutti gli elementi dell'array "contemporaneamente" o anche su un suo *sottoinsieme*.

In pratica gli array rendono possibile tradurre in Fortran la notazione vettoriale (o, più in generale, matriciale) comunemente impiegata in ambito scientifico, possibilità questa che si rivela indispensabile nel trattamento di insiemi di valori omogenei.

Affinché un nome simbolico possa essere usato per indicare un array è necessario che il compilatore abbia tutte le informazioni necessarie allo scopo. In particolare dovranno essere fornite le seguenti indicazioni:

- Il *nome della variabile dimensionata* (ossia il nome simbolico con il quale l'intero *set* di elementi è riferito).
- Il *tipo della variabile* (ossia il tipo comune a tutti i suoi elementi).
- Il *numero delle dimensioni*.
- L'*ampiezza di ogni dimensione*.

Le ragioni per fare uso di un array sono molteplici. Limitandosi a citare soltanto le più comuni, è necessario tenere conto che:

- Sono più semplici da dichiarare rispetto ad una lista di variabili indipendenti.
- Grazie ad operazioni di tipo *globale* è molto semplice operare con essi, specialmente per quanto riguarda operazioni di tipo matematico.

- Le modalità di accesso alle informazioni contenute in un array sono molteplici potendo, in particolare, lavorare con interi array, sottoinsiemi di essi o, più semplicemente, estraendone soltanto degli elementi.
- Rendono un programma più semplice da comprendere a da mantenere.
- Riducono le dimensioni di un programma e, con esso, la probabilità di commettere errori.
- Permettendo di operare su molti oggetti simultaneamente, agevolano il compito di "parallelizzare" un programma.
- Forniscono notevoli opportunità di "ottimizzazione" per i produttori di compilatori.

3.2 Dichiarazione di un array. Terminologia

Un *array* è un gruppo di variabili, tutte dello stesso tipo, che è identificato da un nome unico. Ogni valore all'interno di un array si chiama *elemento dell'array*; esso viene identificato dal *nome* dell'array e da un *indice* che punta ad una particolare posizione all'interno dell'array.

La sintassi generale di una istruzione di dichiarazione di array (che è semplicemente un caso speciale della comune istruzione di dichiarazione di tipo) è la seguente:

tipo, DIMENSION(*estensioni*) [, *attributo*, ...] :: *lista_di_array*

in cui *tipo* è il il tipo di appartenenza di tutti i componenti della *lista_di_array*, la *parola chiave* DIMENSION è un *attributo* che serve a specificare la *dimensione dell'array* che si sta dichiarando, mentre *estensioni* fornisce il *range* in cui spaziano gli *indici* dell'array.

Si noti che per *tipo* è da intendersi indifferentemente un tipo di dati *predefinito* o uno *definito dall'utente* (purché la *definizione* di quest'ultimo sia accessibile all'unità di programma sede della dichiarazione di array). L'attributo DIMENSION specifica la natura dell'array che si sta dichiarando ed introduce le dimensioni dello stesso nella apposita lista delle *estensioni*. Quest'ultima può fornire le dimensioni dell'array o in forma di *costanti intere* o mediante *espressioni intere* che usino *parametri formali* o *costanti* oppure ancora sottoforma di *due punti* (":") nel caso di *array allocabili* o di *array fittizi di forma presunta*

Gli *attributi* compatibili con la dichiarazione di array sono: PARAMETER, PUBLIC, PRIVATE, POINTER, TARGET, ALLOCATABLE, DIMENSION, INTENT, OPTIONAL, SAVE, EXTERNAL, INTRINSIC.

A titolo di esempio, si considerino le due seguenti istruzioni di dichiarazione:

```
REAL, DIMENSION(-2:2) :: a, somma
INTEGER, DIMENSION(0:100) :: dati_input
```

Gli elementi degli array a e somma sono di tipo REAL ed hanno una sola dimensione all'interno della quale un indice varia nell'intervallo -2÷2. Gli elementi dell'array dati_input sono, invece, di tipo INTEGER ed il rispettivo indice è definito nell'intervallo 0÷100.

Analogamente, un array di 50 variabili di tipo *stringa* (ciascuna composta da 20 caratteri) può essere dichiarata al modo seguente:

```
CHARACTER(LEN=20), DIMENSION(50) :: cognome
```

Un modo alternativo per dichiarare un array è di "accoppiarne" le dimensioni direttamente al nome, esemplificativamente:

```
REAL :: array1(10), array2(50), array3(50,50)
```

oppure, di sostituire l'*attributo* `DIMENSION` con la corrispondente *istruzione*:

```
REAL :: array1, array2, array3
DIMENSION array1(10), array2(50), array3(50,50)
```

Si tratta, comunque, di due forme di dichiarazione abbastanza "arcaiche" per cui se ne sconsiglia vivamente l'adozione, favorendo in tal modo l'impiego di istruzioni di dichiarazione separate per array di dimensioni differenti.

Gli interi usati come *limiti delle estensioni* lungo le dimensioni di un array possono essere definiti come *costanti con nome*, mediante l'attributo `PARAMETER`, come nel seguente esempio:

```
INTEGER, PARAMETER :: max_size=100
LOGICAL, DIMENSION(1:max_size) :: answer
...
INTEGER, PARAMETER :: lower_bound=-10
INTEGER, PARAMETER :: upper_bound=10
REAL, DIMENSION(lower_bound:upper_bound) :: score, mark
```

In questo caso, il range dell'array logico `answer` è compreso tra `1` e `100`, mentre il range degli array reali `score` e `mark` spazia da `-10` a `+10`.

Per default, gli indici di un array partono sempre da `1`, tuttavia, come si sarà già intuito dagli esempi precedenti, si può specificare un intervallo differente una volta forniti i limiti (*bound*) inferiore e superiore. Ad esempio, le seguenti istruzioni:

```
REAL, DIMENSION(50) :: w
REAL, DIMENSION(5:54) :: x
LOGICAL, DIMENSION(50) :: y
INTEGER, DIMENSION(11:60) :: z
```

dichiarano quattro array monodimensionali (`w`, `x`, `y` e `z`) ciascuno contenente `50` elementi.

In generale, il numero di elementi di un array in una data dimensione può essere determinato mediante la seguente espressione:

$$estensione = limite\_superiore - limite\_inferiore + 1$$

Il Fortran 90/95 consente ad un array di avere fino a *sette* indici, ognuno dei quali relativo ad una *dimensione* dell'array. Le dimensioni di un array devono essere specificate a mezzo dell'attributo `DIMENSION`. Gli array aventi più di una dimensione sono particolarmente utili per rappresentare dati organizzati in forma tabellare.

E' anche possibile definire una *costante di array*. Una costante di array è formata interamente da valori costanti ed è definita inserendo i valori delle costanti fra speciali delimitatori chiamati *costruttori di array*. Il delimitatore iniziale del costruttore di array è formato da una parentesi tonda aperta seguita dal simbolo di *slash*, "`(/`". Il delimitatore finale, invece, è formato dalla coppia *slash* e parentesi tonda chiusa, "`/)`". Ad esempio, la seguente istruzione definisce una costante di array contenente cinque elementi interi:

```
(/1,2,3,4,5/)
```

Se le dimensioni degli array vengono cambiate spesso per adattare il programma a diversi problemi o per eseguire il programma su processori differenti, allora è consigliabile dichiarare le dimensioni degli array utilizzando *costanti con nome*. Queste ultime, infatti, semplificano enormemente l'operazione di modifica delle dimensioni di un array, come dimostra il seguente frammento di codice nel quale una variazione della sola costante `isize` permette di modificare le dimensioni di tutti gli array dichiarati:

```
INTEGER, PARAMETER :: isize=1000
REAL, DIMENSION(isize) :: array1
LOGICAL, DIMENSION(isize,2*isize) :: array2
INTEGER, DIMENSION(isize,isize,-isize:isize) :: array3
```

Al fine di meglio comprendere tutti i variegati aspetti associati agli array, è bene chiarire il significato di alcuni termini di cui si farà largo uso nel prosieguo:

- Il *rango* di un array coincide, per definizione, con il *numero delle sue dimensioni*. Così, ad esempio, una variabile scalare ha rango `0`, un vettore ha rango `1` e una matrice ha rango `2`. Si noti che il rango di un array non ha nulla in comune con la omonima grandezza algebrica.

- L'*estensione* (*extent*) è riferita, invece, alla particolare dimensione, e rappresenta il *numero di elementi* in tale dimensione.

- La *forma* (*shape*) di un array è un vettore i cui elementi rappresentano le estensioni di ciascuna dimensione. La forma di un array è, in qualche modo, una combinazione del rango e dell'estensione dell'array in ogni dimensione. Così due array avranno la stessa forma se e solo se hanno lo stesso rango e la stessa estensione in ciascuna dimensione.

- L'*ampiezza* (*size*) di un array è il *numero totale di elementi* che costituiscono l'array. Questo numero *può* essere zero, nel qual caso si parla di *array di ampiezza nulla*.

Due array sono detti *compatibili* (*conformable*) se hanno la *stessa forma*. Si noti che tutti gli array sono compatibili con uno scalare, poiché lo scalare può sempre essere riguardato come un array avente la stessa forma dell'array dato e caratterizzato da tutti gli elementi uguali (e pari al valore scalare di partenza).

A chiarimento di quanto detto, si considerino i seguenti array:

```
REAL, DIMENSION :: a(-3:4,7)
REAL, DIMENSION :: b(8,2:8)
REAL, DIMENSION :: d(8,1:9)
INTEGER :: c
```

L'array `a` ha *rango* `2`, *estensioni* `8` e `7`, *forma* `(/8,7/)`, *ampiezza* `56`. Inoltre, `a` è compatibile con `b` e `c`, poiché anche l'array `b` ha forma `(/8,7/)` mentre `c` è uno scalare. Invece, `a` non è compatibile con `d`, avendo quest'ultimo forma `(/8,9/)`.

3.3 Uso degli array

3.3.1 Elementi di array

Un *elemento* di array può essere "estratto" al modo seguente:

nome_array`(`*espressione_intera*`)`

dove *nome_array* è il nome dell'array, mentre *espresione_intera* è una espressione il cui risultato finale, che deve essere un intero appartenente al range dell'estensione dell'array, fornisce l'indice (o la *n_pla* di indici) che puntano al desiderato elemento di array. Ad esempio, se si considerano i seguenti array:

```
REAL, DIMENSION(-1:2) :: a
INTEGER, DIMENSION(0:99) :: b
```

i quattro elementi di `a` sono: `a(-1)`, `a(0)`, `a(1)` e `a(2)`, mentre i cento elementi dell'array `b` sono: `b(0)`, `b(1)`, ..., `b(99)`. Chiaramente, se le variabili intere `i` e `j` hanno valore `3` e `8`, rispettivamente, allora l'espressione `b(j-i)` è del tutto equivalente a `b(5)`, così come `b(i*j)` e `b(j/i)` sono del tutto equivalenti, rispettivamente, a `b(24)` e `b(2)`. E', invece, illegale scrivere `b(3.0)`, poiché l'indice di array *deve* essere di *tipo intero*.

Ogni elemento di array è da riguardarsi come una normale variabile, per cui esso può essere incluso, qualora ciò sia desiderabile, in una qualsiasi espressione logica o aritmetica, oppure è possibile assegnargli il valore di una espressione che abbia, chiaramente, lo stesso tipo dell'array. Ad esempio, una volta dichiarati i seguenti array:

```
INTEGER, DIMENSION(10) :: arr1
LOGICAL, DIMENSION(3,3) :: arr2
REAL, DIMENSION(3,5,5) :: arr3
```

le seguenti istruzioni sono tutte perfettamente valide:

```
arr1(4)=1
arr2(2,2)=.TRUE.
arr3(2,3,3)=REAL(arr1(4))/5.
WRITE(*,*) "arr1(4) = ", arr1(4)
```

3.3.2 Costruttore di array

Un *costruttore di array* crea un array di *rango uno* (ossia un *vettore*) contenente dei valori specificati. Questi ultimi possono essere forniti *da lista* oppure attraverso un *ciclo* `DO` *implicito*, o, al limite, a mezzo di una combinazione dei due sistemi.

La forma generale del costruttore di array è la seguente:

```
(/ lista_dei_valori_del_costruttore di_array /)
```

A titolo di esempio, si possono considerare le seguenti istruzioni di assegnazione di array:

```
REAL, DIMENSION(5) :: a
a=(/(i,i=1,5)/)              != (/1,2,3,4,5/)
a=(/7,(i,i=1,3),9/)          != (/7,1,2,3,9/)
a=(/1./REAL(i),i=1,5)/)      != (/1./1.,1./2.,1./3.,1./4.,1./5./)
a=(/((i+j,i=1,2),j=1,2),5/) != (/((1+j,2+j,3+j),j=1,2)/) = (/2,3,3,4,5/)
a=(/a(i,2:4),a(1:4:2,i+3)/) != (/a(i,2),a(i,3),a(i,4),a(1,i+3),a(3,i+3)/)
```

Esiste una restrizione secondo cui sono possibili unicamente costruttori di array *monodimensionali*. E' possibile, tuttavia, "trasferire" i valori di un array di rango unitario in un array di rango qualsiasi a mezzo della *funzione intrinseca* `RESHAPE`. La funzione `RESHAPE` ha la seguente sintassi:

```
RESHAPE(SOURCE,SHAPE[,PAD][,ORDER])
```

in cui l'argomento `SOURCE` può essere, in generale, un array di forma qualsiasi (nel caso specifico un array di rango unitario). Il risultato dell'applicazione della funzione `RESHAPE` al vettore `SOURCE` fa sì che gli elementi di quest'ultimo vengano "riarrangiati" in modo da formare un array di forma `SHAPE`.

Si noti che se `SOURCE` contiene un numero di elementi maggiori del risultato di `RESHAPE`, allora gli elementi superflui saranno ignorati. Se, al contrario, il numero di elementi del risultato di `RESHAPE` è maggiore di quelli di `SOURCE`, allora l'argomento `PAD` *deve* essere presente. L'argomento `PAD` è un array dello *stesso tipo* di `SOURCE`, ed i suoi elementi vengono usati (se necessario, in maniera ripetuta) in sostituzione degli elementi "mancanti" di `RESHAPE`. Infine, l'argomento opzionale `ORDER` consente agli elementi di `RESHAPE` di essere disposti secondo un ordine alternativo rispetto al consueto *ordinamento degli elementi di array*. A tale scopo è necessario che l'argomento `ORDER` sia un array avente *stessa ampiezza* e *stessa forma* di `SHAPE`, e che contenga le dimensioni di `RESHAPE` nell'*ordine* desiderato.

Un semplice esempio servirà a chiarire l'utilizzo della funzione `RESHAPE`. Le seguenti istruzioni:

```
REAL, DIMENSION(3,2) :: arr
arr=RESHAPE((/((i+j,i=1,3),j=1,2)/),SHAPE=(/3,2/))
```

creano l'array: `arr` $= \begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix}$

Se venisse incluso anche l'argomento `ORDER`, come nel seguente esempio:

```
arr=RESHAPE((/((i+j,i=1,3),j=1,2)/),SHAPE=(/3,2/),ORDER=(/2,1/))
```

allora il risultato sarebbe la creazione della matrice: `arr` $= \begin{bmatrix} 2 & 3 \\ 4 & 3 \\ 4 & 5 \end{bmatrix}$

3.3.3 Inizializzazione degli elementi di un array

Esattamente come avviene con le comuni variabili, prima di poter usare un array è necessario inizializzare i suoi elementi. Se un array non viene inizializzato, il contenuto dei suoi elementi risulta *indefinito*. Lo standard del linguaggio *non* definisce i valori degli elementi di un array non inizializzato, sicché alcuni compilatori impostano automaticamente a zero gli elementi di un tale array, altri li impostano con valori del tutto casuali (ad esempio i valori precedentemente immagazzinati nei registri di memoria corrispondenti), altri ancora generano un errore al tempo di esecuzione. A causa di queste differenze, un programma che sembri funzionare bene su un processore potrebbe non funzionare su un altro sistema. Per ovviare ad una tale evenienza è *opportuno* inizializzare i valori di un array prima di utilizzarlo.

Gli elementi di un array possono essere inizializzati in uno dei seguenti modi:

- Inizializzazione mediante *istruzione di assegnazione*.
- Inizializzazione attraverso l'*istruzione di dichiarazione di tipo*.
- Inizializzazione mediante un'*istruzione* `READ`.

Rimandando l'ultima modalità di inizializzazione al paragrafo relativo alle operazioni di I/O con gli array, si descriveranno ora le prime due tecniche elencate.

Inizializzazione mediante istruzioni di assegnazione

I valori iniziali di un array possono essere assegnati durante la fase esecutiva del programma utilizzando delle opportune istruzioni di assegnazione, *un elemento alla volta* mediante un *ciclo* `DO`, oppure *globalmente* attraverso un *costruttore di array*. I due metodi sono esemplificati di seguito:

```
REAL, DIMENSION(5) :: array1
REAL, DIMENSION(5) :: array2
...
DO i=1,5
   array1(i)=i**2
END DO
...
array2=(/1,4,9,16,25/)
```

Inizializzazione attraverso l'istruzione di dichiarazione di tipo

I valori iniziali possono essere caricati nell'array già in fase di compilazione specificandoli nella istruzione di dichiarazione di tipo. A tale scopo è necessario utilizzare un costruttore di array che specifichi i valori iniziali degli elementi. Ad esempio, la seguente istruzione dichiara l'array `arr` di cinque elementi interi e, al tempo stesso, ne imposta i valori degli elementi a `1`, `2`, `3`, `4` e `5`:

```
INTEGER, DIMENSION(5) :: arr=(/1,2,3,4,5/)
```

Chiaramente, il numero degli elementi nella costante di array deve coincidere con il numero degli elementi dell'array da inizializzare. Se questi due numeri sono diversi si genererà un errore di compilazione.

Il metodo testé descritto è adatto ad inizializzare array di piccole dimensioni. Viceversa, per inizializzare array relativamente grandi è più comodo utilizzare un *ciclo* `DO` *implicito*, il quale ha la seguente forma:

```
(arg_1, arg_2, ..., indice = inizio, fine [, incremento])
```

in cui *arg_1*, *arg_2*, ..., sono parametri che vengono calcolati ogni volta che il ciclo viene eseguito; *indice* è una variabile *contatore* il cui valore parte da *inizio* e termina a *fine* con uno *step* pari a *incremento* (con quest'ultimo pari ad uno per default). Ad esempio, la precedente dichiarazione di array poteva essere scritta, più semplicemente, come:

```
INTEGER, DIMENSION(5) :: arr=(/(i, i=1,5)/)
```

Allo stesso modo, un array reale di mille elementi può essere inizializzato con i valori `1`, `2`, ..., `1000` utilizzando il seguente `DO` implicito:

```
REAL, DIMENSION(1000) :: vett1=(/(REAL(i), i=1,1000)/)
```

I cicli `DO` impliciti possono essere annidati o combinati con le costanti di array per creare schemi di inizializzazione più complessi. Ad esempio, la seguente istruzione imposta a zero gli elementi di `vett2` se il loro indice non è divisibile per cinque, altrimenti gli elementi assumono i valori dei loro indici:

```
INTEGER, DIMENSION(25) :: vett2=(/(0, i=1,4), 5*j, j=1,5)/)
```

In questo caso, il ciclo interno `(0, i=1,4)` viene eseguito completamente ad ogni passaggio del ciclo esterno, pertanto lo schema dei valori risultante è il seguente:

```
0, 0, 0, 0, 5, 0, 0, 0, 0, 10, 0, 0, 0, 0, 15, ...
```

Analogo discorso vale per array di rango maggiore di uno. Ad esempio, volendo costruire, in fase di dichiarazione di tipo, una matrice "identica" di tipo reale di `10`×`10` elementi, si può utilizzare la seguente istruzione:

```
REAL, DIMENSION(10,10) :: mat_identica= &
                     RESHAPE((/(1.0,(0.0,i=1,10),j=1,9),1.0),(/10,10/))
```

Infine, tutti gli elementi di un array possono essere inizializzati con un unico valore, specificando quest'ultimo nell'istruzione di dichiarazione di tipo. La seguente istruzione, ad esempio, imposta ad uno tutti gli elementi dell'array `vett3`:

```
REAL, DIMENSION(50) :: vett3=1.
```

3.3.4 Indici fuori range

Ogni elemento di un array è identificato da uno o più indici interi. Il *range* dei numeri interi che corrispondono agli elementi dell'array dipende dalla dimensione dell'array così come specificato nella relativa istruzione di dichiarazione. Ad esempio, per un array reale dichiarato come:

```
REAL, DIMENSION(5) ::  a
```

gli indici interi `1`, `2`, `3`, `4` e `5` corrispondono agli elementi dell'array. Qualsiasi altro numero intero, minore di `1` o maggiore di `5`, *non* può essere utilizzato come indice poiché *non* corrisponde ad una locazione della memoria riservata all'array. Questi interi sono detti *fuori range* (*out of bounds*). Cosa accade quando un programma faccia uso di un indice fuori range dipende dallo specifico compilatore. In alcuni casi il programma viene interrotto immediatamente, in altri casi viene raggiunta la locazione della memoria che sarebbe stata utilizzata se fosse stato utilizzato quell'indice. Ad esempio, se si utilizzasse l'elemento `a(6)`, essendo `a` l'array definito in precedenza, il computer accederebbe alla prima locazione ubicata dopo il confine dell'array `a`; dal momento che questa locazione potrebbe essere allocata per scopi totalmente diversi, il programma potrebbe fallire in maniera molto "subdola".

E' da precisare che alcuni compilatori prevedono un *controllo* sul programma per verificare che nessuno tra gli indici utilizzati sia fuori range. Normalmente questi controlli richiedono molto tempo e rallentano l'esecuzione dei programmi per cui, se il compilatore lo consente, è buona norma attivare tale controllo in fase di *debugging* ed escluderlo una volta che il programma sia stato testato in modo da avere un file eseguibile più "snello" e veloce.

3.4 Operazioni globali su array

Una importante caratteristica del Fortran 90/95 è la possibilità di eseguire operazioni con *interi array*, ossia trattando l'array nel suo complesso come un singolo oggetto, anziché operare singolarmente su ciascun elemento. Si rimuove così la necessità di lavorare con una serie di cicli `DO` innestati che, tipicamente, sono più *time-consuming* e meno leggibili di operazioni "globali".

Affinché possano avere luogo operazioni globali su array è necessario che (per operazioni binarie) i due array siano *compatibili* (giova ricordare, a questo proposito, che due array sono detti compatibili se hanno la *stessa forma*, anche se *non* hanno lo stesso range di indici in ogni dimensione, ed inoltre, che un qualunque array è sempre compatibile con uno scalare). Le operazioni tra due array compatibili sono sempre eseguite sugli elementi corrispondenti, ed inoltre tutti gli *operatori intrinseci* possono essere applicati ad interi array in forma globale con l'effetto che l'operatore è applicato *elemento*×*elemento*.

Se uno degli operandi è uno scalare, allora lo scalare viene "trasferito" in un array compatibile con l'altro operando. Questa operazione di trasferimento (*broadcasting*) degli scalari è particolarmente utile nelle istruzioni di inizializzazione degli array. Ad esempio, una volta eseguito il seguente frammento di programma:

```
REAL, DIMENSION(4) :: a=(/1.,2.,3.,4./)
REAL, DIMENSION(4) :: c
REAL, DIMENSION b=10
c = a*b
```

l'array `c` conterrà i valori: `10.`, `20.`, `30.`, `40.`

Molte funzioni intrinseche del linguaggio che sono utilizzate con i valori scalari accettano anche gli array come argomenti di input e restituiscono array in output. Queste funzioni si chiamano *funzioni intrinseche di elemento* in quanto operano sugli array elemento×elemento. Molte delle funzioni più comuni, come `ABS`, `SQRT`, `SIN`, `COS`, `EXP` e `LOG`, sono funzioni di elemento. Ad esempio, considerato l'array `a` definito come:

```
REAL, DIMENSION(5) :: a=(/-1.,2.,-3.,4.,-5./)
```

la funzione `ABS(a)` fornisce il risultato: `(/1.,2.,3.,4.,5./)`. Dunque, un array può essere usato come argomento di procedure intrinseche allo stesso modo di uno scalare, con l'effetto che la procedura intrinseca verrà applicata, separatamente ed indipendentemente, a ciascun elemento dell'array. Come ulteriore esempio, dati due array `a` e `b`, l'istruzione:

```
b=SQRT(a)
```

applica l'operatore di radice quadrata a ciascun elemento dell'array `a` ed assegna il risultato all'array `b`. E' chiaro che, nel caso in esame, gli array `a` e `b` dovranno essere compatibili. Allo stesso modo, dato una array di caratteri, `ch`, la seguente istruzione valuta la lunghezza della stringa escludendo i *bianchi di coda* da *tutti* gli elementi dell'array.

```
length=LEN_TRIM(ch)
```

Naturalmente procedure intrinseche possono essere applicate ad array di qualsiasi rango. Ad esempio, se `a` è un array tridimensionale di forma `(/l,m,n/)`, allora `SIN(a)` sarà anch'esso un array di rango tre e forma `(/l,m,n/)`, il cui `(i,j,k)`_*mo* elemento avrà valore `SIN(a(i,j,k))`, per ogni `i=1,...,l`, `j=1,...,m`, `k=1,...,n`. Analogo discorso vale per procedure *non unarie*; ad esempio, dati due array `a` e `b` avente rango unitario e stessa estensione, la funzione `MAX(0,a,b)` restituirà un array di rango uno e tale che il suo `i`_*mo* elemento avrà valore pari a `MAX(0,a(i),b(i))`.

Il vantaggio di lavorare con istruzioni globali può essere facilmente compreso se si osservano i prossimi esempi, in cui si confrontano le soluzioni di tipici problemi di *array processing* risolti sia con operazioni programmate elemento×elemento che con meccanismi globali.

Si considerino tre array monodimensionali, tutti di estensione pari a venti, `a`, `b` e `c`. Si assegni il valore zero a ciascun elemento, quindi, dopo una successiva fase esecutiva, si esegua l'assegnazione: `a(i) = a(i)/3.1 + b(i)*SQRT(c(i))` per ogni valore di `i`.

- Soluzione *elemento*×*elemento*:

```
REAL a(20), b(20), c(20)
...
DO i=1,20
   a(i) = 0.
END DO
...
DO i=1,20
   a(i) = a(i)/3.1 + b(i)*SQRT(c(i))
END DO
```

- Soluzione *globale*:

```
REAL, DIMENSION(20) :: a, b, c
...
a = 0.
...
a = a/3.1 + b*SQRT(c)
```

  Si noti che, nell'esempio in esame, la funzione intrinseca `SQRT` opera su *ciascun* elemento dell'array `c`.

L'utilità di lavorare con array in forma globale è resa tanto più evidente quanto maggiore è il *rango* e meno "concordi" sono i *limiti delle estensioni* degli array operandi. Ad esempio, dichiarati tre array di rango quattro al modo seguente:

```
REAL, DIMENSION(10,10,21,21) :: x
REAL, DIMENSION(0:9,0:9,-10:10,-10:10) :: y
REAL, DIMENSION(11:20,-9:0,0:20,-20:0) :: z
```

la semplice istruzione:

```
x = y+z
```

è perfettamente equivalente al seguente insieme di cicli `DO` innestati:

```
DO i=1,10
   DO j=1,10
      DO k=1,21
         DO l=1,21
            x(i,j,k,l) = y(i-1,j-1,k-11,l-11) + z(i+10,j-10,k-1,l-21)
         END DO
      END DO
   END DO
END DO
```

Come ulteriore esempio, si consideri, ora, un array tridimensionale e se ne trovi il valore massimo fra tutti quelli minori di `1000` e se ne valuti il valore medio fra tutti gli elementi maggiori di `3000`.

Un meccanismo basato su operazioni programmate elemento×elemento richiederebbe tre cicli `DO` innestati e due istruzioni `IF`, mentre lavorando con gli array in forma globale basta, allo scopo, il seguente frammento:

```
REAL, DIMENSION(10,10,10) :: a
...
amax = MAXVAL(a,MASK=(a<1000))
amed = SUM(a,MASK=(a>3000))/COUNT(MASK=(a>3000))
```

In questo frammento di programma si è fatto uso delle seguenti funzioni intrinseche per array:

`MAXVAL`	restituisce il valore dell'elemento massimo dell'array.
`SUM`	restituisce la somma dei valori degli elementi dell'array.
`COUNT`	restituisce il numero degli elementi `.TRUE.` di un array logico.

Queste e tutte le altre funzioni intrinseche relative agli array verranno trattate in dettaglio nel capitolo 6. Per il momento si noti l'uso dell'argomento opzionale `MASK`; si tratta semplicemente di una espressione logica di tipo array per la quale soltanto quegli elementi di `a` che *corrispondono* agli elementi di `MASK` aventi valore `.TRUE.` prendono parte alla chiamata di funzione `MAXVAL` o `SUM`. Così, in questo esempio, `amax` rappresenta il valore massimo di `a` fra tutti quelli minori di `1000` mentre `amed` rappresenta la media aritmetica dei valori di `a` che risultano maggiori di `3000`.

A parte l'aspetto stilistico, è sempre bene fare uso di operazioni con array in forma globale ogni volta se ne abbia la possibilità in quanto esse vengono sempre gestite dal compilatore in modo tale da ottimizzare l'accesso ai componenti. Quanto detto non si applica soltanto in presenza di operazioni algebriche ma in relazione a tutte quante le operazioni di gestione degli array, incluse le operazioni di I/O. Pertanto, anche nelle operazioni di ingresso/uscita specificare il nome dell'intero array piuttosto che riferirsi a ciascun componente a mezzo di uno o più cicli `DO` espliciti o impliciti è sempre la soluzione più rapida ed efficiente.

3.5 Array di ampiezza nulla

Se il *limite inferiore* di una *dimensione* di un array è maggiore del corrispondente *limite superiore*, allora l'array avrà *ampiezza nulla*. Gli array di ampiezza nulla seguono le comuni regole dei normali array per cui, ad esempio, possono essere usati come operandi in istruzioni che coinvolgono array solo a patto di essere compatibili con gli altri operandi.

Gli array di ampiezza nulla tornano molto utili nelle operazioni cosiddette "al contorno". Un esempio di questo particolare array è fornito dal seguente frammento di programma:

```
DO i=1, n
   x(i) = b(i)/a(i,i)
   b(i+1:n) = b(i+1:n)-a(i+1:n,i)*x(i)      ! ampiezza nulla per i=n
END DO
```

il cui scopo è risolvere un sistema di equazioni lineari del tipo *triangolare inferiore* del tipo:

$$\begin{aligned} a_{11}x_1 &= b_1 \\ a_{21}x_1 + a_{22}x_2 &= b_2 \\ &\cdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots a_{nn}x_n &= b_n \end{aligned}$$

Evidentemente si è in presenza di un array di ampiezza nulla quando l'indice `i` assume il suo valore finale `n`.

3.6 Sezioni di array

Oltre che singoli elementi di array o interi array, nelle istruzioni di assegnazione e nelle operazioni aritmetiche è possibile utilizzare anche un *sottoinsieme* o *sezione di array*. Una sezione di array viene specificata sostituendo un indice di array con una *tripletta di indici* o con un *indice vettoriale*. Il formato generale di una tripletta di indici è il seguente:

indice_1 : *indice_2* : *passo*

dove *indice_1* è il *primo* indice da includere nella sezione di array, *indice_2* è l'*ultimo* indice mentre *passo* è l'*incremento* dell'indice nell'insieme dei dati dell'array. Una tripletta di indici opera in modo molto simile a un *ciclo* `DO` *implicito*. La tripletta specifica così l'insieme *ordinato* di tutti gli indici che iniziano da *indice_1* e finiscono con *indice_2* con un incremento pari a *passo*.

Si consideri, a titolo di esempio, il seguente array:

```
INTEGER, DIMENSION(10) :: vett=(/1,2,3,4,5,6,7,8,9,10/)
```

La sezione di array:

```
vett(1:10:2)
```

sarà un array contenente soltanto gli elementi `vett(1)`, `vett(3)`, `vett(5)`, `vett(7)`, `vett(9)`.

Le triplette di indici possono essere scritte nelle seguenti forme alternative:

- Se manca *indice_1*, il primo indice della tripletta viene automaticamente posto uguale al primo indice dell'array.
- Se manca *indice_2*, il secondo indice della tripletta viene automaticamente posto uguale all'ultimo indice dell'array.
- Se manca *passo*, l'incremento della tripletta viene automaticamente posto uguale a uno.

Ad esempio, dato un array `a` definito come:

```
REAL, DIMENSION(10) :: a
```

e tre variabili intere `m`, `n` e `k` aventi valori `3`, `7` e `2` rispettivamente, le seguenti espressioni rappresentano tutte sezioni di array valide:

```
a(:)       ! l'intero array
a(3:9)     ! il sottoinsieme (/a(3),a(4),...,a(9)/)
a(m:n+2:)  ! identico al precedente
a(8:3:-1)  ! il sottoinsieme (/a(3),a(4),...,a(8)/)
a(m+2:)    ! il sottoinsieme (/a(5),a(6),...,a(10)/)
a(:n:k)    ! il sottoinsieme (/a(1),a(3),...,a(7)/)
a(::3)     ! il sottoinsieme (/a(1),a(4),a(7),a(10)/)
a(5::)     ! il sottoinsieme (/a(5),a(6),...,a(10)/)
a(m:m)     ! il solo elemento a(3)
```

Allo stesso modo degli array monodimensionali, è possibile definire sezioni anche per array di rango maggiore di uno. Ad esempio, data la matrice `array_2` dichiarata come:

```
INTEGER, DIMENSION(2:9,-2:1) :: array_2
```

la sua sezione:

```
array_2(4:5,-1:0)
```

rappresenta la seguente "estratta":

```
| array_2(4,-1)  array_2(4,0) |
| array_2(5,-1)  array_2(5,0) |
```

Anche per le sezioni di array di rango non unitario *non* è necessario specificare i valori di tutti gli indici. Considerata, ad esempio, la matrice `array_3` così definita:

```
INTEGER, DIMENSION(3,4) :: array_3
```

le seguenti espressioni sono tutte validi esempi di sezioni:

```
array_3(:,:)   ! l'intera matrice
array_3(2,:)   ! la seconda riga di array_3
array_3(:,3)   ! la terza colonna di array_3
array_3(2,3:4) ! il vettore (/array_3(2,3),array_3(2,4)/)
a(2:2,3:3)     ! il solo elemento array_3(2,3)
```

Come visto finora, le triplette selezionano sezioni *ordinate* degli elementi di array che devono essere inclusi nei calcoli aritmetici. Gli indici vettoriali, invece, consentono delle combinazioni arbitrarie degli elementi di array. Un *indice vettoriale* è un array monodimensionale (ossia un *vettore*) che specifica gli elementi dell'array che devono essere utilizzati nei calcoli aritmetici. Gli elementi dell'array possono essere specificati in qualsiasi ordine ed inoltre uno stesso elemento può essere specificato più volte (nel qual caso si parla di *sezione di array con elementi ripetuti*). L'array risultante contiene un elemento per ogni indice specificato nel vettore. A titolo di esempio, si considerino le seguenti istruzioni di dichiarazione di tipo:

```
REAL, DIMENSION(7) :: a=(/-1.,3.14,0.5,9.,-4.1,12.,2.72/)
INTEGER, DIMENSION(4) :: ind=(/1,3,1,5/)
```

con queste istruzioni, il vettore `a(ind)` sarà l'array `(/-1.,0.5,-1.,-4.1/)`. Si presti attenzione al fatto che una sezione di array con elementi ripetuti *non* può essere utilizzata sul lato sinistro di una istruzione di assegnazione poiché specificherebbe che valori diversi dovrebbero essere assegnati *contemporaneamente* allo *stesso* elemento di array. A chiarimento di quanto detto si consideri il seguente frammento di programma:

```
REAL, DIMENSION(3) :: a=(/10,20,30/)
INTEGER, DIMENSION(3) :: ind=(/1,2,1/)
REAL, DIMENSION(2) :: b
b(ind) = a        ! Operazione errata
```

In questo caso l'istruzione di assegnazione tenterebbe di assegnare all'elemento `b(1)` contemporaneamente i valori `10` e `30`, e ciò è impossibile.

Si vuole mostrare, in conclusione di paragrafo, quanto le operazioni su array in forma globale unitae ad un efficace utilizzo delle sezioni possa condurre a codici eccezionalmente compatti ed efficienti. Il programma che segue ne è la dimostrazione. Esso risolve l'equazione di Laplace alle differenze finite in un dominio quadrato di 10×10 nodi a spaziatura costante con il *metodo di Jacobi*. In altre parole l'equazione:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

che regge, ad esempio, il campo termico conduttivo all'interno di una piastra a conducibilità uniforme, viene discretizzata in ogni *nodo* (i, j) come:

$$\frac{T_{i-1,j} + T_{i+1,j} + T_{i,j-1} + T_{i,j+1} - 4T_{i,j}}{\Delta x^2} = 0$$

A partire, dunque, da un campo di tentativo è possibile, in maniera iterativa, aggiornare la soluzione in ogni nodo come:

$$T_{i,j} = \frac{T^{old}_{i-1,j} + T^{old}_{i+1,j} + T^{old}_{i,j-1} + T^{old}_{i,j+1}}{4}$$

fino a che non risulti verificata la condizione di convergenza qui espressa come *massima differenza fra i due passi*. Le condizioni al contorno assunte sono di campo costante sui lati superiore, inferiore e destro, variabile linearmente sul lato sinistro del dominio computazionale.

```
PROGRAM laplace
! *** Sezione dichiarativa ***
  IMPLICIT NONE
  REAL,DIMENSION(10,10) :: Told, T
  REAL    :: diff
  INTEGER :: i, j, niter
```

```
! *** Sezione esecutiva ***
! Valori iniziali di tentativo
  T = 0
! Condizioni al contorno
  T(1:10,1) = 1.0
  T(1,1:10) = (/(0.1*j,j=10,1,-1)/)
! Inizio iterazioni
  Told = T
  niter = 0
  DO
     T(2:9,2:9) = (Told(1:8,2:9)+Told(3:10,2:9)        &
                     +Told(2:9,1:8)+Told(2:9,3:10))/4.0
     diff = MAXVAL(ABS(T(2:9,2:9)-Told(2:9,2:9)))
     niter = niter+1
   ! Test di convergenza
     PRINT *, "Iter n. ",niter, diff
     IF (diff < 1.0E-4) THEN
        EXIT
     END IF
     Told(2:9,2:9) = T(2:9,2:9)
  END DO
! Stampa dei risultati
  PRINT *, "Campo finale: "
  DO i = 1,10
     PRINT "(10F7.3)", Told(1:10,i)
  END DO
END PROGRAM laplace
```

Si riporta, per completezza, uno stralcio dell'output:

```
 Iter n.            1  0.4750000
 Iter n.            2  0.1812500
 Iter n.            3  0.1125000
 ...
 Iter n.           94   1.116693E-04
 Iter n.           95   1.049638E-04
 Iter n.           96   9.861588E-05
 Campo finale:
  1.000  1.000  1.000  1.000  1.000  1.000  1.000  1.000  1.000  1.000
  0.900  0.890  0.879  0.866  0.848  0.820  0.774  0.686  0.494  0.000
  0.800  0.782  0.762  0.737  0.705  0.660  0.590  0.477  0.291  0.000
  0.700  0.675  0.648  0.617  0.577  0.524  0.449  0.342  0.192  0.000
  0.600  0.570  0.539  0.504  0.462  0.409  0.340  0.249  0.134  0.000
  0.500  0.467  0.434  0.399  0.359  0.312  0.254  0.181  0.095  0.000
```

```
0.400  0.364  0.331  0.299  0.265  0.226  0.181  0.127  0.066  0.000
0.300  0.259  0.227  0.201  0.175  0.148  0.117  0.081  0.042  0.000
0.200  0.144  0.118  0.101  0.087  0.073  0.057  0.040  0.020  0.000
0.100  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
```

Si noti che la funzione intrinseca `MAXVAL`, che sarà ampiamente descritta in seguito, restituisce il valore massimo fra gli elementi dell'array a cui è applicata.

3.7 Ordinamento degli elementi di array

Il Fortran 90/95 *non* specifica come gli array debbano essere organizzati in memoria; in particolare, lo standard non richiede nemmeno che gli elementi di un array occupino locazioni di memoria consecutive. Ci sono, tuttavia, delle situazioni in cui un certo ordinamento è richiesto, come ad esempio nelle operazioni di I/O. In questi casi lo standard richiede che gli elementi di array "appaiano" ordinati in maniera consecutiva: si parla, allora, di *ordinamento ideale* dando per assodato che l'ordine *attuale* così come implementato dal processore potrebbe essere differente. L'allocazione degli elementi di array è stata volutamente lasciata libera in modo da implementare il linguaggio su processori paralleli dove modelli di memoria differenti possono risultare più appropriati.

In ogni caso, l'*ordinamento degli elementi di array* (*array element order*) è tale che:

- Per array monodimensionali:

  `a(i)` precede `a(j)` se `i<j`

- Per array bidimensionali si adopera il precedente criterio sul *secondo* indice e, a parità di questo, sul *primo*; in altri termini l'ordinamento è *per colonne*:

  `a(1,1),a(2,1),...,a(n,1),a(1,2),a(2,2),...,a(n,2),...`

- Per array pluridimensionali, il criterio precedente viene esteso, ordinando in funzione dell'*ultimo* indice e via via per quelli precedenti, ossia, nello scorrere gli elementi dell'array il primo indice varia più rapidamente, il secondo meno rapidamente del primo e così via.

Tale ordinamento "ideale" è quello che va tenuto a mente nelle operazioni di I/O e con alcune funzioni intrinseche (`TRANSFER`, `RESHAPE`, `PACK`, `UNPACK` e `MERGE`), come si avrà modo di vedere in seguito.

3.8 Ottimizzazione delle operazioni con array

Scopo di questo paragrafo è quello di passare in rassegna alcuni "provvedimenti" che se applicati possono dar luogo a codici davvero molto efficienti. Le considerazioni che si faranno adesso, in realtà riguardano prettamente i cicli `DO` e, come tali, si sarebbero potute anticipare al capitolo 2 nel quale è stato affrontato proprio lo studio di questi tipi di costrutto. Si è tuttavia preferito introdurre soltanto ora questo argomento poiché probabilmente è alla luce delle considerazioni svolte in queste pagine che i seguenti argomenti trovano una più proficua dimensione.

3.8.1 *Loop interchange*

I moderni processori impiegano la memoria a disposizione secondo una struttura gerarchica. In questa scala gerarchica è la cosiddetta *cache memory* a caratterizzarsi per l'accesso più veloce ai registri per cui utilizzare i dati mentre risiedono nella *cache memory* consente al programma di ottimizzare i tempi di calcolo. L'accesso ai dati che risiedono temporaneamente nella cache avviene in maniera più efficiente se il programma, per spostarsi da un valore ad un altro, impiega ogni volta un *passo* unitario. Tale passo è definito come l'incremento usato per accedere al successivo elemento di array che si sta processando. Per conoscere il passo che il programma sta usando è necessario sapere in che ordine gli elementi dell'array sono immagazzinati in memoria. Normalmente i compilatori Fortran immagazzinano gli elementi di un array per colonne (contrariamente, ad esempio, a quanto fa il C che, invece, utilizza un ordinamento per righe) per cui, nella maggior pare dei casi, l'ordinamento effettivo rispecchia quello ideale. Così, ad esempio, se l'array `a` è rappresentativo della seguente matrice `4×3`:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{bmatrix}$$

i suoi elementi vengono disposti in memoria nel seguente ordine:

```
a(11) a(21) a(31) a(41) a(12) a(22) a(32) a(42) a(13) a(23) a(33) a(43)
```

Pertanto, il seguente costrutto:

```
DO j = 1,3
   DO i = 1,4
      c(i,j) = b(i,j) + a(i,j)
   END DO
END DO
```

consente di accedere agli elementi di `a` con un passo unitario (ciò garantendo operazioni di accesso più efficienti), al contrario, invece, di quanto accade con quest'altro costrutto ottenuto dal precedente invertendone i cicli interno ed esterno:

```
DO i = 1,4
   DO j = 1,3
      c(i,j) = b(i,j) + a(i,j)
   END DO
END DO
```

Naturalmente, in questo caso, dal momento che le dimensioni dell'array sono molto limitate, l'effetto che ha la scelta del procedimento adottato sulla velocità di esecuzione del programma è irrisoria. Ben altro discorso, invece, si ha quando si lavora con array di dimensioni considerevoli. Ad esempio, considerato il seguente programma:

```
PROGRAM prova_ordinamento
  IMPLICIT NONE
  INTEGER :: i,j
  REAL,DIMENSION(5000,5000) :: array1=1.0,array2=2.0,array3=0.0
  REAL :: inizio,fine
  CALL CPU_TIME(inizio)
  DO j=1,5000
     DO i=1,5000
        array3(i,j) = array1(i,j)+array2(i,j)
     END DO
  END DO
  CALL CPU_TIME(fine)
  WRITE(*,*) "Tempo trascorso: ", fine-inizio
  STOP
END PROGRAM prova_ordinamento
```

si può subito notare che esso accede agli elementi dei array per colonna, ossia secondo il meccanismo ottimale che rispetta l'ordinamento in memoria degli elementi stessi. Compilato con l'Essential Lahey Fortran 90 Compiler (Rel. 4.00b) e mandato in esecuzione su un PC con processore Intel Pentium III, l'output prodotto è:

```
Tempo trascorso:      3.20461
```

Se, poi, si sostituisce il nido di `DO` con una più compatta ed elegante assegnazione in forma globale:

```
array3 = array1+array2
```

il risultato è molto prossimo al precedente:

```
Tempo trascorso:      3.10446
```

Ben altro risultato, invece, si ottiene quando si ritorna al doppio ciclo e si invertono gli indici `i` e `j`:

```
DO i=1,5000
  DO j=1,5000
    array3(i,j) = array1(i,j)+array2(i,j)
  END DO
END DO
```

In questo caso, infatti, l'output è:

```
Tempo trascorso:     24.6555
```

che è un valore di un ordine di grandezza superiore rispetto ai casi precedenti ed è, pertanto, rappresentativo di una condizione di lavoro certamente peggiore.

La subroutine intrinseca `CPU_TIME`, utilizzata nel precedente esempio, sarà trattata in dettaglio al capitolo 5. Per il momento si anticipa solamente che il suo output è il *tempo del processore* (espresso in secondi) secondo una approssimazione dipendente dal processore stesso. A causa di questa dipendenza dal sistema di elaborazione si preferisce produrre il risultato della computazione sempre in termini di differenza fra due "istanti" (tipicamente l'inizio e la fine di un processo) piuttosto che come tempo "assoluto" che, di per sé, avrebbe scarso significato.

E' da notare che taluni compilatori ottimizzanti "riordinano" i cicli del codice sorgente proprio in funzione dell'efficienza dell'accesso ai registri di memoria (così, ad esempio, il loop precedente verrebbe automaticamente "riscritto" nella prima forma secondo un meccanismo noto come *loop interchange*), tuttavia è sempre buona norma sviluppare il codice sorgente in modo tale da minimizzare questo tipo di intervento da parte del compilatore.

Molta cura, inoltre, si dovrebbe avere nel dimensionare un array multidimensionale. Una regola da seguire sempre, infatti, dovrebbe essere quella di dichiarare array caratterizzati da dimensioni degli indici decrescenti andando dal primo all'ultimo indice (a tal proposito si usa spesso dire "*il maggiore per primo*"). Così, ad esempio, in un array bidimensionale, il numero di righe dovrebbe sempre superare il numero di colonne. La seguente dichiarazione di array:

```
REAL(KIND(1.D0)), DIMENSION(100:20) :: matrix
```

rispetta tale regola dal momento che l'estensione del primo indice, `100`, è maggiore di quella del secondo indice, `20`. Così `matrix` si compone di `20` colonne e `100` righe o ciò che è lo stesso, ciascuna colonna di `matrix` contiene `100` elementi mentre ciascuna riga ne contiene soltanto `20`. Il beneficio comportato dall'indicizzare i cicli in questo modo è intimamente associato alla minimizzazione dei salti che la testina di lettura deve compiere per accedere ai dati immagazzinati in celle non contigue della cache memory. In questo modo l'esecuzione di istruzioni di questo tipo:

```
DO j = 1,20
   DO i = 1,100
      matrix(i,j) = matrix(i,j)*SQRT(matrix(i,j))
   END DO
END DO
```

sarà efficacemente ottimizzata. Non solo. Se, infatti, le dimensioni dell'array sono tali che esso non possa essere contenuto per intero nella cache memory, lavorando nel modo anzidetto (ossia facendo sì che la dimensione maggiore dell'array sia quella delle colonne) si permette all'elaboratore di memorizzare temporaneamente nella cache soltanto una (presumibilmente intera) colonna per volta ossia, per ogni ciclo interno, soltanto quella colonna i cui elementi devono essere processati nello step corrente.

Nel caso in cui si fosse comunque costretti ad accedere agli elementi di un array multidimensionale nella maniera meno efficiente, ossia per righe, si badi perlomeno a non fare in modo che la prima dimensione dell'array sia una potenza di due. Infatti, dal momento che le dimensioni della cache memory, espresse in unità di memorizzazione, rappresentano una potenza di due,

nel caso in cui anche la prima dimensione di un array bidimensionale (o, in generale, multidimensionale) fosse una potenza intera di due, il codice farebbe un uso molto limitato della cache rendendo, di fatto, l'uso della memoria eccezionalmente inefficiente. Il prossimo esempio rappresenta un caso del genere:

```
REAL, DIMENSION(512,100) :: a  ! warning: la prima dimensione
...                            !          e' una potenza di 2!!!
DO i = 2,511    ! warning: elementi di array letti secondo
   DO j = 2,99  !          le righe => modo non efficiente!!!
      a(i,j)=(a(i+1,j-1)+a(i-1,j+1))*0.5
   END DO
END DO
```

Un modo per ovviare a questo inconveniente consiste nell'incrementare ad hoc la prima dimensione dell'array, ad esempio definendo la matrice `a` come:

```
REAL, DIMENSION(520,100) :: a
```

In questo caso si migliorano le operazioni di accesso agli elementi dell'array e, con esse, le performance del ciclo, a discapito però della "mole" dell'array che in questo modo presenta elementi aggiuntivi inutilizzati. Si noti che questa operazione di *padding* è, talvolta, effettuata automaticamente da alcuni compilatori a certi livelli di ottimizzazione. Naturalmente questa accortezza è del tutto superflua nel caso in cui si acceda agli elementi dell'array in maniera contigua (ossia per colonne) o nel caso in cui si lavori sugli array in forma globale.

3.8.2 *Loop skewing*

Un'altra operazione che può tornare spesso utile quando si lavora con cicli `DO` innestati è il cosiddetto *loop skewing* la cui applicazione è spesso associata al *loop interchange*. Lo *skewing* ha lo scopo di consentire calcoli di tipo *wavefront*, i quali sono così chiamati perché l'accesso agli elementi dell'array può essere assimilata proprio alla propagazione di un'onda che attraversi idealmente l'array.

Lo *skewing* è realizzato semplicemente sommando il valore dell'indice del ciclo esterno, moltiplicato per un *fattore di skew f*, agli estremi dell'indice del ciclo interno e sottraendo la stessa quantità ad ogni ricorrenza dell'indice del ciclo interno all'interno del *loop body*. Il risultato quando $f = 1$ è mostrato nel seguente esempio in cui sono riportati, in sequenza, un comune doppio ciclo innestato e la sua versione "avvitata":

```
! ciclo originale
DO i=2,n-1
   DO j=2,m-1
      a(i,j) = (a(i-1,j)+(a(i,j-1)+a(i+1,j)+a(i,j+1))/4.
   END DO
END DO
```

```
! ciclo ottimizzato con loop skewing
DO i=2,n-1
   DO j=i+2,i+m-1
      a(i,j-i) = (a(i-1,j-i)+(a(i,j-i-1)+a(i+1,j-i)+a(i,j-i+1))/4.
   END DO
END DO
```

Naturalmente il codice trasformato è del tutto equivalente a quello originale ma con l'unica sostanziale differenza che per ogni valore di `j` tutte le operazioni in `i` possono essere eseguite in parallelo. Si noti, infine, che il ciclo originale può certamente essere sottoposto a *loop interchange* ove mai se ne realizzasse la necessità, ma di certo non potrebbe essere parallelizzato. Viceversa, al ciclo sottoposto a *skewing* può essere facilmente applicato l'*interchange loop* e parallelizzato:

```
! ciclo sottoposto a skewing e interchange
DO j=4,m+n-2
   DO i=MAX(2,j-m+1),MIN(n-1,j-2)
      a(i,j-i) = (a(i-1,j-i)+(a(i,j-i-1)+a(i+1,j-i)+a(i,j-i+1))/4.
   END DO
END DO
```

3.8.3 *Loop unswitching*

La presenza di istruzioni o costrutti condizionali all'interno di un ciclo può pregiudicare notevolmente l'utilizzo della cache memory e spesso impedisce la parallelizzazione del codice. In questi casi può essere opportuno applicare una semplice trasformazione, detta *loop unswitching*, che consiste nel decomporre le istruzioni `IF` e di "clonare" il ciclo di conseguenza al fine di pervenire a due o più cicli privi di condizioni logiche da analizzare ad ogni passaggio. Ciò, naturalmente può aver luogo solo quando la condizione logica del costrutto `IF` è *loop invariant* in quanto è soltanto in questo caso che essa potrà essere estratta dal ciclo.

A titolo di esempio, nel ciclo che segue la variabile `x` non dipende dal ciclo per cui la condizione logica sul suo valore può essere senz'altro portata fuori dal loop così che il ciclo stesso potrà essere duplicato in modo tale che le istruzioni del ramo `.TRUE.` del costrutto `IF` potranno essere eseguite in parallelo.

```
! ciclo originale
DO i=2,n
   a(i) = a(i)+c
   IF(x<7) THEN
      b(i) = a(i)*c(i)
   ELSE
      b(i) = a(i-1)*b(i-1)
   END IF
END DO
```

```
! ciclo dopo l'unswitching
IF(n>0) THEN
   IF(x<7) THEN
      DO i=2,n
         a(i) = a(i)+c
         b(i) = a(i)*c(i)
      END DO
   ELSE
      DO i=2,n
         a(i) = a(i)+c
         b(i) = a(i-1)*b(i-1)
      END DO
   END IF
END IF
```

Si noti che in presenza di cicli innestati si dovrà cercare, *ove possibile*, di trasferire la verifica della condizione logica all'esterno del ciclo più esterno al fine di evitare che il test abbia luogo ad ogni spazzata dei cicli interni.

3.8.4 *Loop unrolling*

Spesso è possibile effettuare operazioni sugli array in modo molto più rapido semplicemente "srotolando" gli array stessi. Per poter meglio comprendere questo concetto (noto in letteratura come *loop unrolling*) si consideri il seguente costrutto:

```
somma = 0.
DO i = 1,5
   somma = somma+x(i)
END DO
```

Questo loop, come è oltremodo evidente, si limita a sommare i cinque elementi dell'array `x` e ad assegnare questo valore alla variabile `somma`. Ebbene, da un punto di vista di velocità di esecuzione, sarebbe molto più efficiente sostituire al ciclo `DO` la "brutale" ed esplicita sommatoria:

```
somma = x(1)+x(2)+x(3)+x(4)+x(5)
```

Il motivo di ciò è oltremodo semplice: in presenza del ciclo `DO`, infatti, ad ogni spazzata ha luogo la valutazione dell'espressione logica `i==n` che ha lo scopo di controllare la chiusura del loop. Per poter quantificare la differenza in termini di tempo di calcolo fra le due soluzioni è necessario, vista l'esiguità del numero di calcoli effettuati, inserire i due costrutti ciascuno in un ciclo che esegue gli stessi calcoli alcune migliaia di volte. Così, ad esempio, il programma seguente:

```
PROGRAM test_loop_unrolling
```

```
! Sezione dichiarativa
  IMPLICIT NONE
  INTEGER , PARAMETER :: long=SELECTED_INT_KIND(9)
  INTEGER(KIND=long) :: j
  INTEGER :: i
  REAL(KIND(1.D0)), DIMENSION(5) :: x=1.0
  REAL :: inizio, fine, somma
! Sezione esecutiva
  somma=0.
  CALL CPU_TIME(inizio)
  DO j=1_long,1000000_long
     DO i=1,5
        somma=somma+x(i)
     END DO
     somma=0.
  END DO
  CALL CPU_TIME(fine)
  WRITE(*,'(1X,A17,G14.7)') "DO loop: ", fine-inizio
  somma=0.
  CALL CPU_TIME(inizio)
  DO j=1_long,1000000_long
     somma=x(1)+x(2)+x(3)+x(4)+x(5)
     somma=0.
  END DO
  CALL CPU_TIME(fine)
  WRITE(*,'(1X,A17,G14.7)') "Loop unrolling: ", fine-inizio
  STOP
END PROGRAM test_loop_unrolling
```

compilato con il compilatore Digital Visual Fortran 6.0 e mandato in esecuzione su un PC con processore Intel Pentium III, produce il seguente output:

```
         DO loop:  0.1301872
 Loop unrolling:  0.2002880E-01
```

che evidenzia i benefici associati al meccanismo di *loop unrolling*. Naturalmente l'operazione di srotolamento degli array può essere estesa anche ai casi in cui le dimensioni dell'array non consentono di esplicitarne tutti gli elementi. Allo scopo basta eseguire l'unrolling un certo numero di volte su un numero ben preciso di elementi. Ad esempio, dato il costrutto:

```
n = 500
somma = 0.
DO i = 1,n
   somma = somma+x(i)
END DO
```

l'array `x` può essere efficacemente "svolto" per una lunghezza, ad esempio, ancora pari a `5` e ripetendo tale operazione `m = n/5` volte:

```
m = n/5
somma = 0.
k = 1
DO i = 1,m
   somma = somma+x(k)+x(k+1)+x(k+2)+x(k+3)+x(k+4)
   k = k+5
END DO
! adesso si valutano i termini eventualmente "avanzati"
! il seguente ciclo ha effetto solo se MOD(n,5)/=0
DO i = k,n
   sum = sum+x(i)
END DO
```

Sebbene il *loop unrolling* possa produrre codici molto efficienti, è spesso oneroso effettuare questa operazione "a mano" a parte il fatto che il codice, in questo modo, si presenterebbe in una forma meno compatta e, quindi, meno facile da gestire (ciò aumentando, ad esempio, la possibilità di commettere errori di battitura). Fortunatamente molti moderni compilatori offrono degli strumenti di ottimizzazione che eseguono tale operazione in maniera automatica. Ad esempio, il citato compilatore della Digital prevede l'opzione di ottimizzazione `/unroll:`*`count`* in cui il valore intero *`count`* (che rappresenta il *fattore di unrolling*) specifica quante volte i cicli debbano essere "svolti". Così, ad esempio, ricompilando il programma `test_loop_unrolling` con la direttiva di compilazione `/unroll:5` si ottiene, anche per il costrutto `DO`, un risultato molto simile a quello ottenuto esplicitando il calcolo della sommatoria:

```
Loop unrolling: 1.0014400E-02
```

I benefici associati al loop unrolling possono essere riguardati anche in termini di incremento di efficienza nell'accesso ai registri di memoria e della possibilità di parallelizzare le istruzioni. Per chiarire questo concetto si guardi l'esempio che segue in cui, in successione, sono riportati un comune ciclo `DO` operante su un array `a` e lo stesso ciclo sottoposto a loop unrolling:

```
! ciclo originale
DO i=2,n-1
   a(i) = a(i) + a(i-1)*a(i+1)
END DO

! ciclo ottimizzato
DO i=2,n-2,2
   a(i) = a(i) + a(i-1)*a(i+1)
   a(i+1) = a(i+1) + a(i)*a(i+2)
END DO
IF (MOD(n-2.2)==1) THEN
```

```
   a(n-1) = a(n-1) + a(n-2)*a(n)
END IF
```

Come si può facilmente verificare la seconda soluzione presenta tutti i vantaggi associati al loop unrolling. Infatti:

- Dal momento che ad ogni iterazione vengono eseguite due istruzioni invece di una, il numero di cicli risulta dimezzato e, con esso, anche il numero di volte che il programma dovrà verificare la condizione di fine ciclo.
- L'accesso ai registri è resa più efficiente dal fatto che ad ogni iterazione i valori di `a(i)` e `a(i+1)` vengono usati due volte (una volta per ciascuna delle due istruzioni del *loop body*) il che porta da tre a due il numero dei registri interrogati per ogni istruzione. In altre parole, mentre nel primo caso è necessario leggere il contenuto dei tre registri `a(i)`, `a(i-1)` e `a(i+1)` per ciascuna delle `n`-2 iterazioni del ciclo, nel secondo caso è necessario leggere il contenuto dei quattro registri `a(i)`, `a(i-1)`, `a(i+1)` e `a(i+2)` ma soltanto `n`$/2-1$ volte.
- Il parallelismo del codice può essere sfruttato per far sì che la seconda istruzione venga eseguita non appena il risultato della prima istruzione sia stato memorizzato sicché contemporaneamente all'esecuzione della seconda istruzione può aver luogo l'aggiornamento dell'indice del ciclo e l'esecuzione della successiva prima istruzione del blocco.

3.8.5 *Loop blocking*

Un'altra operazione spesso assai efficace per migliorare le prestazioni di un programma è quella nota come *loop blocking* (o *loop tiling*) la quale consiste nello scomporre un grosso nido di `DO` in una sequenza di cicli innestati più piccoli che operino su blocchi di dati di dimensioni tali che ciascuno possa essere contenuto per intero nella *cache memory*. Ma il *blocking* può servire anche ad incrementare l'efficienza dell'accesso ai dati contenuti nella cache. La necessità del *loop blocking* è illustrata dall'esempio che segue. Si consideri il seguente costrutto la cui funzione è quella di assegnare alla matrice `a` la trasposta di `b`:

```
DO i=1,n
   DO j=1,n
      a(i,j) = b(j,i)
   END DO
END DO
```

Come si può notare, l'accesso agli elementi di `b` avviene con passo unitario mentre l'accesso agli elementi di `a` si caratterizza per un passo pari ad `n`. In questo caso, chiaramente, anche scambiando i cicli interno ed esterno (ossia effettuando un *interchange*) non si risolverebbe il problema ma semplicemente lo si "invertirebbe". Ciò che invece può essere fatto è spezzettare il doppio ciclo in parti più piccole, ad esempio al modo seguente:

```
DO ti=1,n,64
   DO tj=1,n,64
```

```
      DO i=ti,MIN(ti+63,n)
         DO j=tj,MIN(tj+63,n)
            a(i,j) = b(j,i)
         END DO
      END DO
   END DO
END DO
```

decomponendo quindi il cosiddetto *spazio delle iterazioni* (ossia il dominio in cui scorrono gli indici del ciclo) in sotto-rettangoli tali che la cache è impiegata al meglio.

3.8.6 *Cycle shrinking*

Quando un ciclo si caratterizza per alcune dipendenze fra le sue istruzioni che impediscono il parallelismo è talvolta ancora possibile operare una certa manipolazione che garantisca un certo grado di parallelismo delle istruzioni. Una operazione di questo tipo è quella nota come *cycle shrinking* e consiste nel convertire un ciclo seriale in un doppio ciclo caratterizzato da un ciclo esterno seriale e da un ciclo interno parallelo. Ad esempio, considerato il seguente ciclo:

```
DO i=1,n
   a(i+k) = b(i)
   b(i+k) = a(i)+c(i)
END DO
```

appare subito evidente che, a causa della dipendenza della seconda istruzione dalla prima, le operazioni non possono essere parallelizzate *in toto*. Tuttavia, dal momento che il termine `a(i+k)` viene assegnato all'iterazione `i` ma utilizzato solo all'iterazione `i+k` (si dice, allora, in questo caso che la *distanza di dipendenza* è `k`) è possibile effettuare in parallelo blocchi di `k` istruzioni ed eseguire globalmente questi blocchi in serie:

```
DO ti=1,n,k
   DO i=ti,ti+k-1
      a(i+k) = b(i)
      b(i+k) = a(i)+c(i)
   END DO
END DO
```

3.8.7 *Loop fission*

Quando due cicli interni sono controllati da uno stesso ciclo esterno, è spesso soluzione più efficiente scomporre il ciclo esterno in due parti più piccole in modo da lavorare con due coppie di cicli innestati del tutto indipendenti l'uno rispetto all'altro. Ciò consente di applicare in maniera più semplice ed efficace altri provvedimenti ottimizzanti quali il *cache blocking*, il *loop fusion* o il *loop interchange*. Questa operazione, detta di *loop fission* (o *loop distribution* o anche *loop splitting*) in generale consiste nello scomporre un singolo ciclo in più cicli all'interno dello

stesso spazio di iterazione ma ciascuno racchiudente soltanto un sottoinsieme delle operazioni del ciclo originale. Tale operazione può avere diversi effetti benefici sul codice:

- Può dare luogo a cicli perfettamente innestati.
- Può creare *sub-loop* privi (o con ridotto numero) di dipendenze fra le relative istruzioni.
- Consentendo, ad esempio, di operare con array più piccoli che possono essere ospitati per intero nella *cache* può migliorare l'utilizzo della *cache memory*.
- Diminuendo la pressione di registrazione può aumentare la frequenza di riutilizzo degli stessi registri di memoria.

L'esempio che segue mostra come il *loop fission* possa rimuovere le dipendenze e consentire a una parte di ciclo di essere eseguita in parallelo:

```
! ciclo originale
DO i=1,n
   a(i) = a(i)*c
   x(i+1) = x(i)*7+x(i+1)+a(i)
END DO

! dopo il loop fission
DO i=1,n
   a(i) = a(i)*c
END DO
DO i=1,n
   x(i+1) = x(i)*7+x(i+1)+a(i)
END DO
```

Un ulteriore esempio, leggermente più complesso, può servire a chiarire ulteriormente il concetto. Si consideri il seguente ciclo:

```
! ciclo originale
DO i=1,n
   a(i) = a(i)+b(i-1)   ! S1
   b(i) = c(i-1)*x + y  ! S2
   c(i) = 1./b(i)       ! S3
   d(i) = SQRT(c(i))    ! S4
END DO
```

Per poter comprendere le dipendenze fra le varie istruzioni sarà bene riportare in successione l'effetto di ciascuna istruzione ad ogni iterazione:

- `S2` assegna un valore a `b(i)` che sarà usato da `S1` all'iterazione successiva.
- `S2` assegna un valore a `b(i)` che sarà usato da `S3` all'iterazione corrente.

- S3 assegna un valore a c(i) che sarà usato dalla stessa S3 all'iterazione successiva.
- S3 assegna un valore a c(i) che sarà usato da S4 all'iterazione corrente.

Da quanto detto si può subito notare che il ciclo originale può essere "splittato" nelle tre componenti S1, (S2, S3) e S4. Naturalmente, poiché le operazioni eseguite da S1 e da S4 sono subordinate a quelle eseguite dal blocco (S2, S3) è necessario che nella versione "ottimizzata" del ciclo il blocco (S2, S3) preceda S1 e S4.

```
! dopo il loop fission
DO i=1,n
   b(i) = c(i-1)*x + y  ! S2
   c(i) = 1./b(i)       ! S3
END DO

DO i=1,n
   a(i) = a(i)+b(i-1)   ! S1
END DO

DO i=1,n
   d(i) = SQRT(c(i))    ! S4
END DO
```

3.8.8 *Loop fusion*

Se due cicli vicini operano sugli stessi dati, conviene di solito combinarli in un unico ciclo operando in tal modo l'operazione nota come *loop fusion*. Questo procedimento, che rappresenta un pò l'inverso del *loop fission*, non solo rende un codice più compatto e leggibile ma, al tempo stesso, consente di utilizzare più efficientemente la *cache memory* limitando al massimo il numero delle operazioni di lettura dei dati in memoria. Inoltre esso può aumentare la velocità di esecuzione dei cicli in quanto, fondendo insieme diversi cicli controllati dal medesimo indice, riduce il numero di condizioni logiche da verificare per la chiusura del loop. Un esempio di applicazione dell'operazione di fusione è riportata di seguito:

```
! ciclo originale
DO i=1,n
   a(i) = a(i) + k
END DO
DO i=1,n
   b(i+1) = b(i) + a(i)
END DO

! dopo il loop fusion
DO i=1,n
   a(i) = a(i) + k
```

```
   b(i+1) = b(i) + a(i)
END DO
```

In questo caso, ad esempio, uno dei benefici connessi alla fusione in termini di ottimizzazione dell'accesso ai registri di memoria consiste nel fatto che ciascun termine `a(i)` deve essere letto una sola volta e non due come accadeva nella versione originale.

Per fondere due cicli è necessario che i loro indici scorrano fra gli stessi estremi. Se ciò non accade è talvolta possibile attraverso alcune operazioni di manipolazione (come il *peeling* o il *normalizing* che verranno discussi di seguito) adattare gli estremi in modo da rendere i cicli compatibili. Due cicli con gli stessi estremi possono essere fusi se non esiste alcuna istruzione `S2` del secondo ciclo che dipenda da una istruzione `S1` del primo ciclo. Il seguente esempio, quindi, mostra due cicli che non possono essere sottoposti a *fusion* in quanto l'istruzione del secondo ciclo non può avvenire in parallelo con l'istruzione del primo ciclo ma deve essere ad essa subordinata:

```
! ciclo originale
DO i=1,n
   a(i) = a(i) + k
END DO
DO i=1,n
   b(i+1) = b(i) + a(i+1)
END DO
```

Naturalmente tutto quanto di buono si dica circa dal *loop fusion* potrebbe sembrare automaticamente una critica al *loop fission* e viceversa. In realtà molto dipende da quale aspetto si vuole ottimizzare, ad esempio la fissione potrebbe essere la scelta giusta su un processore con una *cache* limitata o in presenza di array di grandissime dimensioni (in questi casi appare fondamentale la riduzione della pressione sui registri) o ancora per favorire l'applicazione di altre operazioni talvolta indispensabili come il *loop interchange*; la fusione, dal canto suo, permette una migliore parallelizzazione delle istruzioni e può consentire una drastica riduzione dell'accesso ai registri di memoria (riducendo così il tempo speso nelle operazioni di lettura/scrittura "interni"). In ogni caso, del tutto in generale si può dire che in presenza di cicli onerosi (cioè per grossi valori di `n`) un codice sottoposto a *loop fission* verrà eseguito molto più rapidamente su una macchina seriale, lo stesso ciclo ottimizzato con un *loop fusion* sarà eseguito molto più velocemente su un sistema parallelo.

3.8.9 *Loop pushing*

Il *loop pushing* è un tipo di trasformazione che consiste nello spostare un ciclo dalla procedura chiamante ad una versione clonata della procedura invocata. Questa operazione può risultare molto utile per ridurre i tempi di calcolo di un programma (si noti, a tal proposito, che le chiamate di procedura sono sempre operazioni assai lente) e deve comunque essere effettuata quasi sempre "a mano" in quanto sono ben pochi i compilatori Fortran che eseguono questo tipo di ottimizzazione. Un semplice esempio di *loop pushing* è riportato di seguito:

```
! loop e procedura originali
DO i=1,n
   CALL mysub(x,i)
END DO

SUBROUTINE mysub(a,j)
   IMPLICIT NONE
   REAL, DIMENSION(:) :: a
   INTEGER :: j
   ...
      a(j) = a(j)+c
      ...
      RETURN
END SUBROUTINE mysub

! invocazione della procedura e
! procedura clonata dopo il loop pushing
CALL mysub2(x)

SUBROUTINE mysub2(a)
   IMPLICIT NONE
   REAL, DIMENSION(:) :: a
   INTEGER :: i
   ...
      DO i=1,n
         a(i) = a(i) + c
      END DO
      ...
      RETURN
END SUBROUTINE mysub2
```

Naturalmente se nel ciclo ci sono altre istruzioni oltre alla chiamata di procedura, allora un prerequisito per il *pushing* dovrà essere il *loop fission*.

3.8.10 *Loop peeling*

Un'altra operazione utile in molti casi è il cosiddetto *loop peeling*. Il *peeling* ha due applicazioni principali: rimuovere le dipendenze create dalle prime o dalle ultime iterazioni di un ciclo (consentendo così la parallelizzazione delle istruzioni) e creare un *match* tra gli estremi di variazione degli indici di due o più cicli in modo da consentire una efficace operazione di fusione. Come esempio di applicazione della presente operazione, si guardi il seguente frammento di programma:

```
DO i=2,n
```

```
   b(i) = b(i)+b(2)
END DO
DO i=3,n
   a(i) = a(i)+k
END DO
```

Come si può notare, il primo ciclo non può essere parallelizzato a causa della dipendenza fra l'iterazione `i=2` e le iterazioni `i=3,...,n`. Inoltre i due cicli non possono essere fusi sotto il controllo di un'unica variabile indice a causa della differenza degli intervalli di variazione degli indici stessi. Tuttavia, se a `b` si applica l'unrolling relativamente al solo primo componente si risolvono contemporaneamente entrambi i problemi, come banalmente mostrato di seguito:

```
! dopo il peeling
IF(n>=2) THEN
   b(2) = b(2) + b(2)
END IF
DO i=3,n
   b(i) = b(i)+b(2)
END DO
DO i=3,n
   a(i) = a(i)+k
END DO

! dopo il peeling e il fusion
IF(n>=2) THEN
   b(2) = b(2) + b(2)
END IF
DO i=3,n
   b(i) = b(i)+b(2)
   a(i) = a(i)+k
END DO
```

3.8.11 *Loop reversal*

Il *loop reversal* è un tipo di trasformazione che consiste nell'invertire il verso di avanzamento dell'indice di un ciclo all'interno del range di iterazione. Questo tipo di trasformazione si rivela spesso indispensabile per poter consentire l'applicazione di altre forme di ottimizzazione come il *loop fusion* o il *loop interchange*. Come esempio, si consideri il seguente costrutto:

```
DO i=1,n
   DO j=1,n
      a(i,j) = a(i-1,j+1)+1
   END DO
END DO
```

Come è facilmente verificabile, per esso non è possibile scambiare gli indici `i` e `j` per cui non è possibile applicare il *loop interchange* pertanto se il valore `n` è sufficientemente elevato l'utilizzo della memoria si caratterizzerà per una scarsissima efficienza. Se, tuttavia, si invertono gli estremi del ciclo interno si ottiene un costrutto formalmente analogo:

```
DO i=1,n
   DO j=n,1,-1
      a(i,j) = a(i-1,j+1)+1
   END DO
END DO
```

ma caratterizzato dalla possibilità di poter essere sottoposto ad *interchange*.

Analogamente, se si osservano i cicli seguenti ci si accorge subito che essi non possono essere "fusi" direttamente a causa della dipendenza fra le istruzioni `S2` ed `S3`:

```
! cicli originali
DO i=1,n
   a(i) = b(i)+1    ! S1
   c(i) = a(i)/2    ! S2
END DO
DO i=1,n
   d(i) = 1/c(i+1)  ! S3
END DO
```

La dipendenza fra le istruzioni anzidette può essere facilmente riconosciuta, infatti considerando ad esempio `i=5` ci si accorge subito che nella versione "fusa" del costrutto:

```
! dopo l'applicazione del loop fusion
! (versione scorretta)
DO i=1,n
   a(i) = b(i)+1    ! S1
   c(i) = a(i)/2    ! S2
   d(i) = 1/c(i+1)  ! S3
END DO
```

l'istruzione `S3` farebbe uso del vecchio valore di `c(6)` piuttosto che del nuovo valore calcolato da `S2`. La soluzione al problema è ovviamente offerta dalla possibilità di invertire il verso di scorrimento degli indici `i` e `j` (ossia applicando un *loop reversal*):

```
! dopo l'applicazione del loop reversal
DO i=n,1,-1
   a(i) = b(i)+1    ! S1
   c(i) = a(i)/2    ! S2
END DO
DO i=n,1,-1
   d(i) = 1/c(i+1)  ! S3
END DO
```

I cicli così riscritti potranno essere fusi in un unico corpo senza possibilità di equivoco:

```
! dopo l'applicazione del loop reversal e
! del loop fusion (versione corretta)
DO i=n,1,-1
   a(i) = b(i)+1    ! S1
   c(i) = a(i)/2    ! S2
   d(i) = 1/c(i+1)  ! S3
END DO
```

Infine, il *reversal* può anche eliminare la necessità di creare degli array temporanei quando si eseguano operazioni su array in forma globale. Difatti, anche quando un ciclo è espresso nella notazione vettoriale il compilatore può decidere di applicare una scalarizzazione ossia può eseguire le operazioni in maniera seriale secondo uno o più cicli. Tuttavia tale conversione non sempre è banale in quanto la notazione vettoriale implica sempre una certa contemporaneità delle operazioni. A titolo di esempio, si riportano di seguito in sequenza una operazione su array in forma globale (ossia un ciclo vettorizzato), la sua ovvia (e scorretta) conversione nella forma seriale e la sua meno ovvia ma corretta conversione:

```
! A: espressione vettoriale
a(2:n-1) = a(2:n-1)+a(1:n-2)

! B: scalarizzazione non corretta
DO i=2,n-1
   a(i) = a(i)+a(i-1)
END DO

! C: scalarizzazione corretta
DO i=2,n-1
   temp(i) = a(i)+a(i-1)
END DO
DO i=2,n-1
   a(i) = temp(i)
END DO
```

Il motivo per cui la prima forma di scalarizzazione è sbagliata dipende dal fatto che nella versione originale del ciclo ogni elemento di `a` deve essere incrementato del valore del precedente elemento e questo deve accadere simultaneamente per ciascun elemento. Ciò che invece accade nella versione errata `B` è l'incremento di ciascun elemento di `a` del valore "aggiornato" del precedente elemento. La soluzione generale è quella mostrata come versione `C` consiste nell'introdurre un array temporaneo `temp` e di disporre di un loop a parte per poter riversare i valori calcolati nell'array `a`. Naturalmente l'esigenza dell'array temporaneo viene meno quando i due cicli così scritti possono essere sottoposti a *fusion*. Non è questo il caso per cui la necessità del vettore `temp` sembrerebbe irremovibile. Tuttavia le condizioni cambiano se si inverte il verso di scorrimento dell'indice `i`, pervenendo in tal modo alla migliore soluzione scalarizzata possibile:

```
! D: scalarizzazione corretta e
!    loop reversal
DO i=n-1,2,-1
   a(i) = a(i)+a(i-1)
END DO
```

3.8.12 *Loop normalization*

Il *loop normalization* converte tutti i cicli in modo tale che l'indice del ciclo valga inizialmente 1 (oppure 0) e sia incrementato di un fattore unitario ad ogni iterazione. Normalizzare un ciclo può consentire altri tipi di trasformazione come il *peeling* o la fusione sebbene la sua maggiore utilità risiede nel fatto di consentire al compilatore di eseguire test di coerenza sulle variabili indiciate nei cicli, test che di solito possono aver luogo soltanto in presenza di cicli con range normalizzati. Un esempio di *loop normalization* è il seguente:

```
! ciclo originale
DO i=1,n
   a(i) = a(i)+k
END DO
DO i=2,n+1
   b(i) = a(i-1)+b(i)
END DO

! dopo la normalizzazione
DO i=1,n
   a(i) = a(i)+k
END DO
DO i=1,n
   b(i+1) = a(i)+b(i+1)
END DO
```

Si noti che è soltanto dopo l'operazione di normalizzazione che i due cicli potranno essere eventualmente fusi:

```
! dopo normalizzazione e fusione
DO i=1,n
   a(i) = a(i)+k
   b(i+1) = a(i)+b(i+1)
END DO
```

E' da notare che tutte queste operazioni vengono svolte automaticamente da alcuni compilatori a determinati livelli di ottimizzazione il che lascia al programmatore come unica preoccupazione quella di scrivere programmi che utilizzino cicli nel modo più semplice, chiaro e compatto possibile, mentre è il compilatore a farsi carico del compito di ottimizzare il codice "dietro alle quinte". Nondimeno è importante capire i problemi coinvolti nell'ottimizzazione dei

cicli in quanto non sempre si può fare completo affidamento sulle scelte operate dal compilatore per cui una buona conoscenza delle problematiche in esame deve essere sempre alla base, da un lato, dello sviluppo del codice sorgente, dall'altro della scelta (sempre ponderata) delle direttive di compilazione da impiegare.

3.9 Array e istruzioni di controllo

Esistono molti casi in cui l'uso degli array può aiutare a rendere il codice molto più semplice ed efficiente. Già si è visto, a proposito delle operazioni globali, come in taluni casi sia possibile semplificare se non addirittura fare a meno dei cicli a conteggio grazie ad un uso maturo degli array; spesso è possibile trarre giovamento dall'uso di array "di appoggio" anche per snellire e rendere più efficienti altri tipi di costrutti, come quelli condizionali. Si supponga, ad esempio, di dover assegnare alla variabile `stato` un valore che dipende, a sua volta, dal valore assunto dalla variabile intera `x`. Si supponga, inoltre, che `x` possa assumere tutti i valori compresi in un certo intervallo (a tal riguardo, le variabili `x` e `stato` potrebbero essere riguardate, rispettivamente, come il dominio ed il codominio di una funzione di argomento intero). Tanto per fissare le idee si può fare riferimento al seguente ramo di programma:

```
INTEGER :: x, stato
...
IF (x == -1) THEN
   stato = 1
ELSE IF (x == +1)
   stato = 5
ELSE IF (x == 0) then
   stato = 0
ELSE
   PRINT*, " Valore di x non significativo! "
END IF
```

Il precedente frammento di programma assegna alla variabile `stato` un diverso valore a seconda che la variabile intera `x` valga, rispettivamente, `-1`, `+1` o `0`; se, invece, il valore assoluto di `x` risulta maggiore di `1` viene segnalata una condizione di errore. In casi come questi si può ridurre notevolmente il numero di condizioni da analizzare (e, con esse, il numero di clausole `ELSE IF` da introdurre) inserendo i possibili valori di `x` in un array ed utilizzando questo array alla destra dell'*unica* istruzione di assegnazione:

```
INTEGER x, i, stato
INTEGER, DIMENSION(-1:1) :: lookup = (/1,0,5/)
IF ((ABS(x)<2) THEN
   stato = lookup(x)
ELSE
   PRINT*, " Valore di x non significativo! "
END IF
```

3.10 Operazioni di I/O con array

Il Fortran 90/95 consente di effettuare operazioni di input/output sia sui *singoli elementi* di un array sia, in forma globale, sull'*intero array*.

3.10.1 Input e output degli elementi di un array

Come è noto, un elemento di array è da riguardarsi come una comune variabile per cui è possibile utilizzarlo nelle istruzioni `READ`, `WRITE` e `PRINT` nel modo consueto. Ad esempio, per visualizzare il contenuto di determinati elementi di un array basta specificarli nella lista degli argomenti di una istruzione `WRITE`, come dimostra il seguente blocco di istruzioni in cui l'istruzione `WRITE` è utilizzata per riprodurre a video i primi cinque elementi dell'array reale `a`:

```
REAL, DIMENSION(10) :: a
...
WRITE(*,100) a(1),a(2),a(3),a(4),a(5)
100 FORMAT(1X,'a = ',5F10.2)
```

Allo stesso modo, l'istruzione:

```
READ(*,*) a(3)
```

permette di inserire da tastiera il valore dell'elemento `a(3)`.

Il ciclo `DO` implicito

Nelle istruzioni di I/O è possibile inserire un *ciclo* `DO` *implicito*. Questo costrutto permette di visualizzare più volte una lista di argomenti in funzione di un indice. Ogni argomento della lista viene visualizzato una volta per ogni valore dell'indice del ciclo implicito.

La forma generale di un'istruzione `WRITE` in presenza di un ciclo `DO` implicito è la seguente:

```
WRITE(...)  (arg_1, arg_2, ..., indice = inizio, fine [, incremento])
```

dove gli argomenti *arg_1*, *arg_2*, ... sono i valori da leggere in input o da visualizzare in output. La variabile intera *indice* è l'*indice* del ciclo implicito, mentre *inizio*, *fine* e *incremento* rappresentano, rispettivamente, il *limite inferiore*, *superiore* e l'*incremento* della variabile indice (si noti che la variabile *opzionale* *incremento* è pari ad uno per default). Chiaramente, l'indice e tutti i parametri di controllo del ciclo devono avere valori di *tipo intero*. (La sintassi è perfettamente analoga nel caso di istruzione `READ` o `PRINT`).

Pertanto, facendo uso di un ciclo `DO` implicito, l'esempio precedente può essere riscritto in forma più compatta al modo seguente:

```
REAL, DIMENSION(10) :: a
...
WRITE(*,100) (a(i), i=1,5 )
100 FORMAT(1X,'a = ',5F10.2)
```

In questo caso la lista degli argomenti contiene un solo elemento, `a(i)`, e la lista stessa viene ripetuta una volta per ogni valore assunto dalla variabile `i` che rappresenta un indice variabile da `1` a `5`.

Come tutti i normali cicli `DO`, anche i cicli impliciti possono essere *annidati*, la qual cosa torna particolarmente utile quando si opera con array di rango maggiore di uno. Nel caso di due cicli `DO` impliciti annidati, è necessario ricordare che il ciclo *interno* sarà eseguito *completamente* ad ogni iterazione del ciclo *esterno*: quest'ordine viene osservato indipendentemente dal numero di cicli nidificati. A titolo di esempio, si consideri il seguente frammento di programma:

```
INTEGER, DIMENSION(4,4) :: mat=(/(i=1,16)/)
mat=RESHAPE((/(i, i=1,SIZE(mat,DIM=1))/),SHAPE(mat))
...
WRITE(*,100) ((mat(i,j), j=1,3 ), i=1,2)
100 FORMAT(1X,I5,1X,I5)
```

in cui l'istruzione `WRITE` contiene due cicli impliciti nidificati. La variabile indice del ciclo interno è `j`, mentre quella del ciclo esterno è `i`. Quando l'istruzione `WRITE` viene eseguita, la variabile `j` assume i valori `1`, `2` e `3` mentre `i` vale `1`. Quindi, la variabile `i` assume valore `2` e il ciclo interno viene ripetuto, con `j` che assume nuovamente i valori `1`, `2` e `3`. Pertanto l'output di questo blocco di istruzioni sarà:

```
 1   5   9
 2   6  10
```

Le funzioni intrinseche `SIZE` e `SHAPE` usate nel precedente esempio forniscono, rispettivamente, l'*estensione* (nella dimensione specificata), e la *forma* dell'array argomento. Per i dettagli relativi al loro utilizzo si rimanda alla sezione del capitolo 6 dedicata alle funzioni intrinseche di array.

3.10.2 Input e output con interi array o con sezioni di array

Oltre agli *elementi* di un array, anche un *intero array* o una sua *sezione* possono essere utilizzati con le istruzioni `READ`, `WRITE` e `PRINT`. Se il nome di un array viene specificato *senza* indici in un'istruzione di I/O, il compilatore suppone che *tutti* gli elementi dell'array debbano essere letti o visualizzati. Analogamente, se ad essere specificata in una istruzione di I/O è una sezione di array, allora il compilatore suppone che l'intera sezione debba essere letta o visualizzata.

Il seguente frammento di programma mostra un semplice esempio di utilizzo delle istruzioni di scrittura per array in forma globale per sezioni:

```
REAL, DIMENSION(5) :: a=(/1.,2.,3.,20.,10./)
INTEGER, DIMENSION(4) :: vett=(/4,3,4,5/)
WRITE(*,100) a        ! array globale
WRITE(*,100) a(2::2)  ! sezione di array con tripletta di indici
WRITE(*,100) a(vett)  ! sezione di array con indice vettoriale
100 FORMAT(1X,5F8.3)
```

Il risultato delle precedenti istruzioni è, chiaramente:

```
 1.000   2.000   3.000  20.000  10.000
 2.000  20.000
20.000   3.000  20.000  10.000
```

Un ulteriore esempio è fornito dal seguente, semplice, programma:

```
PROGRAM prova_print
  IMPLICIT NONE
  INTEGER :: i, j
  INTEGER, DIMENSION(3,3) :: a=RESHAPE((/1,2,3,4,5,6,7,8,9/),(/3,3/))
  PRINT*, a
  PRINT*
  DO i=1,3
     PRINT*, (a(i,j), j=1,3)
  END DO
  PRINT*
  PRINT*, a(:2,:2)
  PRINT*
  DO i=1,2
     PRINT*, (a(i,j), j=1,2)
  END DO
END PROGRAM prova_print
```

che fornisce in output il seguente risultato:

```
1  2  3  4  5  6  7  8  9

1  4  7
2  5  8
3  6  9

1  2  4  5

1  4
2  5
```

Come si può notare, l'output è stato volutamente "raddoppiato", nel senso che il programma stampa per due volte l'array `a` e per due volte la stessa sezione `a(1:2,1:2)`, e ciò allo scopo di fornire un ulteriore esempio di *ordinamento degli elementi di array* che, come si ricorderà, è definito *per colonne*.

Allo stesso modo, gli elementi di un array `b` possono essere introdotti (secondo l'ordine degli elementi di array) secondo la semplice istruzione:

```
READ(*,*) b
```

con cui l'introduzione di ciascun valore deve essere seguito dalla pressione del tasto *return* (che verrà semplicemente ignorata). Sempre con lo stesso criterio è possibile leggere gli elementi di una sezione di array, ad esempio con istruzioni del tipo:

```
READ(*,*) c(::2,::2)
```

3.11 Array come componenti di tipi di dati derivati

Fra i componenti di un *tipo di dati derivato* è possibile avere uno o più array. Questa possibilità può spesso servire a ridurre drasticamente il numero delle componenti scalari del tipo *user defined*, come dimostra il seguente esempio:

```
TYPE studente
   CHARACTER(LEN=15) :: nome, cognome
   INTEGER, DIMENSION(30) :: esami
END TYPE studente
```

In questo *record*, l'array `esami` potrebbe contenere i voti di 30 esami sostenuti da uno `studente`. Chiaramente, è possibile dichiarare un intero gruppo di elementi di tipo `studente`, (ad esempio per classificare il curriculum di tutti quanti gli iscritti ad un corso):

```
TYPE(studente), DIMENSION(50) :: iscritti
```

In questo caso, il `nome` ed il `cognome` dell'`i_`*mo* `studente` fra gli `iscritti` possono essere facilmente estratti come:

```
iscritti(i)%nome
iscritti(i)%cognome
```

così come il voto del terzo esame da questi sostenuto può essere letto accedendo al seguente componente:

```
iscritti(i)%voto(3)
```

Un vincolo che debbono rispettare gli array componenti di un tipo di dati derivato è il fatto che essi possono essere soltanto *array di forma esplicita* aventi *limiti costanti*, oppure *array di forma rimandata* (per la definizione di queste due forme di array si rimanda al capitolo 6).

Un'importante restrizione che si ha quando si estrae un componente di un oggetto di tipo derivato avente array come componenti, è rappresentata dal fatto che al massimo *uno* degli array abbia rango maggiore di zero. In altre parole, tutti gli array componenti, ad eccezione (al massimo) di uno, devono presentarsi come variabili "indiciate". Ad esempio, è possibile scrivere:

```
iscritti%voto(i) = 28
```

per porre uguale a 28 il `voto` dell'`i_`*mo* esame di ciascun studente iscritto, ed è possibile scrivere:

```
iscritti(j)%voto = 28
```

per porre uguale a 28 ciascun `voto` del `j_`*mo* studente iscritto. E', invece, illegale scrivere:

```
iscritti%voto = 28
```

poiché, in questo caso, sia `iscritti` che `voto` sono array di rango uno.

Il prossimo programma costituisce un ulteriore esempio di utilizzo di array e sezioni di array come componenti di strutture.

```
PROGRAM array_in_strut
! Sezione dichiarativa
   IMPLICIT NONE
   TYPE :: small
      INTEGER :: i, j
      CHARACTER(LEN=5), DIMENSION(3) :: word
   END TYPE small
   TYPE :: big
      INTEGER :: count
      TYPE(small), DIMENSION(8) :: data
   END TYPE big
   TYPE(big) :: a
   INTEGER, DIMENSION(5,2,4) :: x
! Sezione esecutiva
   READ(*,*) a
   WRITE(*,*) a%count,a%data(3)
   READ(*,*) x
   WRITE(*,*) x(2:4,:,3)
   STOP
END PROGRAM array_in_strut
```

3.12 Array di elementi di un tipo di dati derivati

La definizione e l'utilizzo di un array di tipo di dati derivati è abbastanza intuitiva per cui si preferisce ricorrere direttamente ad un esempio allo scopo di illustrare in maniera più diretta la sintassi della struttura.

```
PROGRAM chimica
! Demo di array di strutture
   IMPLICIT NONE
   INTEGER, PARAMETER :: lunghezza_nome=18, numero_elementi=109
   TYPE :: elemento_chimico
      CHARACTER(LEN=lunghezza_nome) :: nome_elemento
      CHARACTER(LEN=3) :: simbolo
```

```
      INTEGER :: numero_atomico
      REAL :: massa_atomica
   END TYPE elemento_chimico
   TYPE(elemento_chimico), DIMENSION(numero_elementi) :: tavola_periodica = &
      elemento_chimico("xxx","X",0,0.0)
   INTEGER :: loop
! start
   tavola_periodica(1) = elemento_chimico("Idrogeno","H",1,1.01)
   tavola_periodica(2) = elemento_chimico("Elio","He",2,4.00)
! ...
   DO loop = 1,numero_elementi
      WRITE(*,*) tavola_periodica(loop)
   END DO
   STOP
END PROGRAM chimica
```

Il precedente programma definisce una struttura di nome `elemento_chimico` avente due componenti letterali atte ad ospitare il nome ed il simbolo chimico di un elemento, e due componenti numeriche atte ad ospitare il numero atomico e la massa atomica. Quindi, viene dichiarato l'array `tavola_periodica` di `109` elementi (tanti essendo gli elementi chimici presenti in natura) di tipo `elemento_chimico` e, infine, viene avviata una fase di immagazzinamento dei dati relativi agli elementi chimici e la loro successiva stampa a video.

Si guardi, ora, questa definizione di tipo:

```
TYPE triplet
   REAL :: u
   REAL, DIMENSION(3) :: du
   REAL, DIMENSION(3,3) :: d2u
END TYPE triplet
```

le cui le componenti possono essere pensate, ad esempio, come il valore di una funzione scalare tridimensionale (`u`), la sua derivata prima (`du`) e la sua derivata seconda (`d2u`), rispettivamente. Con la seguente dichiarazione:

```
TYPE(triplet) :: t
TYPE(triplet), DIMENSION(10,10,10) :: a
```

`t%u` rappresenterà un valore reale; `t%du` un array di elementi `t%du(1)`, `t%du(2)` e `t%du(3)`; `t%d2u`, infine, sarà un array bidimensionale di elementi `t%d2u(i,j)`. Allo stesso modo, l'elemento `a(1,2,5)` rappresenta un oggetto di tipo `triplet` mentre l'espressione `a(1,2,5)%d2u(1,2)` si riferisce all'elemento `(1,2)` dell'array `d2u`, secondo campo dell'oggetto `triplet` a sua volta elemento `(1,2,5)` dell'array `a`.

3.13 Costrutto e istruzione WHERE

Come si è potuto apprendere nelle pagine precedenti, il Fortran 90/95 consente di utilizzare, nelle istruzioni di assegnazione, singoli elementi di un array, una sua sezione o l'intero array. Ad esempio, per estrarre il logaritmo degli elementi dell'array bidimensionale `matrice` di `m`×`n` elementi, si può utilizzare la semplice istruzione:

```
log_matrice = LOG(matrice)
```

ben più compatta del doppio ciclo `DO` innestato:

```
DO i=1,m
   DO j=1,n
      log_matrice(i,j) = LOG(matrice(i,j))
   END DO
END DO
```

Tuttavia, supponendo di voler estrarre il logaritmo soltanto di *alcuni* elementi dell'array `matrice`, ad esempio (come appare oltremodo chiaro) dei soli elementi positivi, un modo per svolgere questo compito è quello di applicare l'operatore logaritmo ad un elemento per volta, combinando i cicli `DO` con il costrutto `IF`, al modo seguente:

```
DO i=1,m
   DO j=1,n
      IF(matrice(i,j)>0.) THEN
         log_matrice(i,j) = LOG(matrice(i,j))
      ELSE
         log_matrice(i,j) = -HUGE(a)
      END IF
   END DO
END DO
```

dove, evidentemente, lo scopo della funzione intrinseca `HUGE` è quello di produrre un numero reale di valore assoluto molto alto (per l'esattezza, il valore più grande rappresentabile in macchina) al fine di simulare numericamente il concetto di "infinito". Lo stesso risultato può, tuttavia, essere ottenuto in maniera più efficace con una forma speciale di istruzione di assegnazione: la cosiddetta *assegnazione con schema* o *maschera* (*masked array assignment*). Si tratta di un'istruzione la cui operazione è controllata da un *array logico* della *stessa forma* dell'array specificato nell'assegnazione. L'operazione di assegnazione è svolta soltanto per gli elementi dell'array che corrispondono ai valori dello *schema di assegnazione*.

Nel Fortran 90/95 questo tipo di assegnazione è implementato mediante il *costrutto* `WHERE`, la cui forma generale è la seguente:

```
[nome:]  WHERE (espressione_schema)
  istruzioni_di_assegnazione_di_array    !  primo blocco
ELSEWHERE [nome]
  istruzioni_di_assegnazione_di_array    !  secondo blocco
END WHERE [nome]
```

in cui *espressione_schema* è un array logico avente la stessa forma dell'array da manipolare nelle istruzioni di assegnazione incluse nel costrutto `WHERE`. Questo costrutto applica le operazioni del `primo blocco` a tutti gli elementi dell'array per i quali *espressione_schema* risulti `.TRUE.`, altrimenti applica le operazioni del `secondo blocco`.

E' possibile assegnare un *nome* al costrutto `WHERE`, in tal caso anche la corrispondente istruzione `END WHERE` *deve* avere lo stesso nome, mentre questo è *facoltativo* nella clausola `ELSEWHERE`.

La clausola `ELSEWHERE` è, in realtà, opzionale; in sua assenza il costrutto `WHERE` assume la forma particolarmente semplice:

```
[nome:]  WHERE (espressione_schema)
   istruzioni_di_assegnazione_di_array
END WHERE [nome]
```

Il precedente esempio può, pertanto, essere implementato in maniera estremamente elegante ed efficiente mediante il seguente costrutto `WHERE`:

```
WHERE (matrice>0.)
   log_matrice = LOG(matrice)
ELSEWHERE
   log_matrice = valore_fittizio
END WHERE
```

Una particolare forma di costrutto `WHERE` è quello cosiddetto *con maschera*, in cui l'*espressione_schema* è rappresentata da un array di tipo logico. Un valido esempio è il seguente:

```
PROGRAM prova
! Demo WHERE con maschera
   IMPLICIT NONE
   LOGICAL, DIMENSION(5) :: mask=(/.TRUE.,.FALSE.,.TRUE.,.TRUE.,.FALSE./)
   REAL, DIMENSION(5) :: x=0.0
! start
   WHERE(mask)
      x = 1.0
   ELSEWHERE
      x = 2.0
   END WHERE
   WRITE(*,*) x
   STOP
END PROGRAM prova
```

il cui output è:

```
   1.000000       2.000000       1.000000       1.000000       2.000000
```

Il Fortran 95 ha esteso il costrutto `WHERE` dando la possibilità di lavorare anche con più clausole `ELSEWHERE` e di assegnare un *nome* al costrutto. La forma generale di un costrutto `WHERE`, secondo lo standard del Fortran 95, è pertanto:

```
[nome:]  WHERE (espressione_schema_1)
  istruzioni di assegnazione di array    !  primo blocco
ELSEWHERE (espressione_schema_2)[nome]
  istruzioni_di_assegnazione_di_array    !  secondo blocco
ELSEWHERE [nome]
  istruzioni_di_assegnazione_di_array    !  terzo blocco
...
END WHERE [nome]
```

Un costrutto `WHERE` può avere, così, tutte le clausole `ELSEWHERE` *con maschera* che si desiderano ma ad ogni elemento di array saranno applicate le operazioni di uno ed un solo blocco. Questo costrutto applica le operazioni del `primo blocco` a tutti gli elementi dell'array per i quali *espressione_schema_1* risulti `.TRUE.` e *espressione_schema_2* risulti `.FALSE.`, applica le operazioni del `secondo blocco` a tutti gli elementi dell'array per i quali *espressione_schema_1* risulti `.FALSE.` e *espressione_schema_2* risulti `.TRUE.`, e applica, infine, le operazioni del `terzo blocco` a tutti gli elementi dell'array per i quali risultino `.FALSE.` contemporaneamente *espressione_schema_1* ed *espressione_schema_2*. Inoltre, il Fortran 95 ha anche introdotto la possibilità di lavorare con costrutti `WHERE` innestati, possibilità questa non prevista dal Fortran 90 standard, potendo così dare vita a costrutti del tipo:

```
assign1: WHERE(cnd1)
   ...               ! maschera: cnd1
ELSEWHERE(cnd2)
   ...               ! maschera: cnd2.AND.(.NOT.cnd1)
   assign2: WHERE(cnd4)
      ...            ! maschera: cnd2.AND.(.NOT.cnd1).AND.cnd4
   ELSEWHERE
      ...            ! maschera: cnd2.AND.(.NOT.cnd1).AND.(.NOT.cnd4)
   END WHERE assign2
   ...
ELSEWHERE(cnd3) assign1
   ...               ! maschera: cnd3.AND.(.NOT.cnd1).AND.(.NOT.cnd2)
ELSEWHERE assign1
   ...               ! maschera: .NOT.cnd1.AND.(.NOT.cnd2).AND.(.NOT.cnd3)
END WHERE assgn1
```

Oltre che il *costrutto* `WHERE`, il Fortran 90/95 mette a disposizione anche l'*istruzione* `WHERE` che permette di controllare casi meno complessi. Anche l'istruzione `WHERE` può essere usata per eseguire una istruzione di assegnazione "controllata", ossia un'istruzione che venga applicata soltanto se una condizione logica risulti vera. L'istruzione `WHERE` è, pertanto, un altro modo

per far sì che una operazione "globale" sia applicata soltanto ad *alcuni* elementi di un array, e *non* a *tutti*. La sua sintassi è:

```
WHERE(espressione_schema) istruzione_di_assegnazione_di_array
```

Un classico esempio di utilizzo dell'istruzione `WHERE` è quello che permette di evitare, in corrispondenza di un rapporto, l'errore associato ad una divisione per zero:

```
INTEGER, DIMENSION(5,5) : num, den
REAL, DIMENSION(5,5) : rapp
...
WHERE(den/=0) rapp = REAL(num)/REAL(den)
```

oppure consente di applicare un "filtro" ad una serie di dati:

```
REAL, DIMENSION(100) : aaa
...
WHERE(aaa<0.) aaa=0.
```

Naturalmente sia il costrutto che l'istruzione `WHERE` non si applicano soltanto ad interi array ma anche a sezioni di array, come dimostra la seguente istruzione:

```
WHERE(matr1(j:k,j:k)/=0.) matr2(j+1:k+1,j-1:k-1) = &
                          matr2(j+1:k+1,j-1:k-1)/matr1(j:k,j:k)
```

3.14 Costrutto e istruzione `FORALL`

Il Fortran 95 ha introdotto un nuovo costrutto per consentire l'applicazione di un insieme di istruzioni soltanto ad un sottoinsieme di elementi di un array. Gli elementi su cui operare vengono scelti in base ad un *indice* ed al rispetto di una *condizione logica*. Questo nuovo costrutto è chiamato `FORALL`. La sua forma generale è la seguente:

```
[nome:]  FORALL (ind_1=tripl_1 [,ind_2=tripl_2,...,espressione_logica])
   lista di istruzioni di assegnazione di array
END FORALL [nome]
```

Ogni *indice* nel costrutto `FORALL` è specificato da una *tripletta* avente la forma:

```
inizio:fine:incremento
```

in cui *inizio* è il valore iniziale di *indice*, *fine* ne è il valore finale mentre *incremento* rappresenta lo *step*. Le operazioni della *lista di istruzioni di assegnazione di array* verranno applicate a tutti gli elementi dell'array aventi indici compresi negli intervalli specificati e che soddisfano alla condizione determinata da *espressione_logica*.

Il *nome* del costrutto `FORALL` è opzionale ma nel caso in cui sia presente esso dovrà essere specificato anche nell'istruzione `END FORALL`.

Un semplice esempio permetterà di comprendere meglio l'utilizzo del costrutto in esame. Il prossimo frammento di programma crea una matrice identica (ossia una matrice avente valore uno sulla diagonale principale e zero in tutti gli altri posti) di 10×10 elementi:

```
REAL, DIMENSION(10,10) :: matrice_identica=0.0
...
FORALL (i=1:10)
   matrice_identica(i,i) = 1.0
END FORALL
```

Un esempio appena più complicato è rappresentato dal seguente costrutto, che consente di calcolare il reciproco di un array `mat` evitando di operare divisioni per zero:

```
FORALL (i=1:n, j=1:m, mat(i,j)/=0.)
   mat(i,j) = 1./mat(i,j)
END FORALL
```

In generale, ogni espressione che può essere scritta a mezzo di un costrutto `FORALL` potrebbe essere scritta, in modo perfettamente equivalente, con una serie di *cicli* `DO` *innestati* combinati con uno *blocco* `IF`. Ad esempio, l'esempio precedente potrebbe essere riscritto come:

```
DO i=1,n
   DO j=1,m
      IF (mat(i,j)/=0.)
         mat(i,j) = 1./mat(i,j)
      END IF
   END DO
END DO
```

La differenza sostanziale fra le due soluzioni consiste nel fatto che mentre le istruzioni in un ciclo `DO` vengono eseguite in un *ordine stretto*, le istruzioni contenute in un costrutto `FORALL` possono essere eseguite in un *ordine qualsiasi* dipendente dal processore. Questa libertà consente alle architetture parallele di ottimizzare un programma in termini di velocità di esecuzione: ogni elemento può essere "manipolato" da un processore differente e i diversi processori possono terminare il loro lavoro in un ordine qualsiasi ed indipendentemente l'uno dall'altro senza che ciò abbia influenza sul risultato finale.

Se il corpo di un costrutto `FORALL` contiene più di una istruzione, allora il processore opererà prima su tutti gli elementi coinvolti dalla prima istruzione, per successivamente operare sugli elementi coinvolti nella seguente istruzione, e così via. Ad esempio, nel caso seguente:

```
FORALL (i=2:n-1, j=2:n-1)
   a(i,j) = SQRT(a(i,j))
   b(i,j) = 1./a(i,j)
END FORALL
```

gli elementi di `a` saranno valutati *tutti prima* degli elementi di `b`.

Si osservi che, proprio a causa del fatto che ciascun elemento di un array può essere processato indipendentemente dagli altri, il corpo di un costrutti `FORALL` *non* può contenere funzioni di "trasformazione" il cui risultato dipenda dai valori dell'*intero* array. Si noti, inoltre, che, come per tutti gli altri costrutti, anche per i costrutti `FORALL` è permessa la presenza di forme innestate. Ad esempio, la sequenza:

```
FORALL (i=1:n-1)
   FORALL (j=i+1:n)
      a(i,j) = a(j,i)
   END FORALL
END FORALL
```

assegna la trasposta della parte triangolare bassa di `a` alla parte triangolare alta.

Inoltre, un costrutto `FORALL` può includere un costrutto o un'istruzione `WHERE`, con il risultato che ciascuna istruzione di un costrutto `WHERE` sono eseguite in sequenza. Come si esempio, si osservi il seguente costrutto:

```
FORALL (i=1:n)
   WHERE (a(i,:)==0) a(i,:)=i
   b(i,:) = i/a(i,:)
END FORALL
```

In questo esempio, ciascun elemento nullo di `a` viene sostituito dal suo indice di riga e, *una volta completata questa operazione*, a tutti gli elementi delle righe di `b` vengono assegnati i reciproci dei corrispondenti elementi di `a` moltiplicati per il corrispondente indice di riga.

Oltre che il *costrutto* `FORALL`, il Fortran 95 prevede anche una *istruzione* `FORALL`, la cui sintassi è la seguente:

```
FORALL (ind_1=tripl_1 [,...,espressione_logica]) assegnazione
```

in cui l'istruzione di *`assegnazione`* viene eseguita soltanto per quegli elementi che soddisfino i parametri di controllo di `FORALL`.

Validi esempi di istruzioni `FORALL` sono:

```
FORALL (i=1:20, j=1:20) a(i,j) = 3*i + j**2
FORALL (i=1:100, j=1:100, (i>=j.AND.x(i,j)/=0)) x(i,j) = 1.0/x(i,j)
```

Di queste due istruzioni, la prima esegue l'assegnazione:

$$a_{i,j} = 3 \times i + j^2 \qquad \forall\, i, j \in \{1, \ldots, 20\}$$

mentre la seconda sostituisce ciascun elemento non nullo al di sotto della diagonale principale di `x` o sulla diagonale stessa con il suo reciproco.

3.15 Gestione di array di grandi dimensioni

Piuttosto che immagazzinare un array di enormi dimensioni, può essere talvolta conveniente la scelta di ricalcolarne i valori degli elementi tutte le volte che servono. Questa scelta è senz'altro infruttuosa per quanto concerne la velocità di esecuzione del programma ma potrebbe essere vantaggiosa in termini di ingombro "medio" di memoria.

Un'altra soluzione, più efficiente, al problema in esame è l'uso di strutture dati "impaccate". Un esempio è rappresentato dalla gestione di matrici sparse le quali si incontrano assai spesso

in molte aree delle scienze computazionali. Si supponga, ad esempio, di avere una matrice `A` di dimensioni `n`×`m` (dove almeno una fra le dimensioni `n` ed `m` sia molto grande) e si supponga, inoltre, che la maggior parte degli elementi di questa matrice siano nulli. Comprimere questa matrice secondo un opportuno algoritmo può eliminare lo spreco di memoria associato all'immagazzinamento di un enorme numero di zeri. A tale scopo, detto `ne` il numero di elementi non nulli della matrice, un primo metodo per comprimere la struttura potrebbe essere quello di usare tre array monodimensionali:

1. Un primo array, `ar`, contenente tutti e soli gli `ne` elementi diversi da zero di `A`, immagazzinandoli consecutivamente ed alternamente per colonne.

2. Un altro array, `ia`, anch'esso di `ne` elementi, contenente gli indici di riga corrispondenti ordinatamente agli elementi di `ia`.

3. Il terzo array, `ja`, di `N+1` elementi, contenente la posizione iniziale relativa di ogni colonna di `A` all'interno dell'array `ar`, vale a dire ogni elemento `ja(j)` indica dove parte la `j_`*ma* colonna di `A` nell'array `ar`. Se tutti gli elementi della colonna `j_`*ma* sono nulli, si porrà `ja(j)=ja(j+1)`. L'ultimo elemento, `ja(N+1)`, indica la posizione che segue l'ultimo elemento dell'array `ar`, vale a dire `ne+1`.

A titolo di esempio, si supponga che `A` rappresenti la seguente matrice di interi:

$$\begin{bmatrix} 11 & 0 & 13 & 0 & 0 & 0 & 0 \\ 21 & 22 & 0 & 0 & 25 & 0 & 0 \\ 0 & 32 & 33 & 0 & 0 & 0 & 0 \\ 0 & 0 & 43 & 0 & 45 & 0 & 47 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 61 & 62 & 0 & 0 & 0 & 0 & 67 \end{bmatrix}$$

Ebbene, secondo l'algoritmo appena illustrato, questa matrice può essere compattata a mezzo dei tre array:

```
ar = (/11,21,61,62,32,22,13,33,43,25,45,47,67/)
ia = (/1,2,6,6,3,2,1,3,4,2,4,4,6/)
ja = (/1,4,7,7,10,10,12,14/)
```

Un'altra possibilità è quella di definire un nuovo tipo di dati che comprenda, ad esempio, il valore del generico elemento non nullo di `A` ed i corrispondenti indici di riga e di colonna, e dichiarare un array di tale tipo e di dimensione tale da contenere tutti gli elementi di `A`:

```
TYPE nonzero
   INTEGER :: value
   INTEGER :: row, column
END TYPE

TYPE (nonzero), DIMENSION(ne) :: sparsa
```

Così, ad esempio, con riferimento all'esempio precedente si avrebbe `ne=13` e

```
sparsa(1)=nonzero(11,1,1)
sparsa(2)=nonzero(21,2,1)
sparsa(3)=nonzero(11,1,1)
sparsa(4)=nonzero(22,2,2)
sparsa(5)=nonzero(25,2,5)
sparsa(6)=nonzero(32,3,2)
sparsa(7)=nonzero(33,3,3)
sparsa(8)=nonzero(43,4,3)
sparsa(9)=nonzero(45,4,5)
sparsa(10)=nonzero(47,4,7)
sparsa(11)=nonzero(61,6,1)
sparsa(12)=nonzero(62,6,2)
sparsa(13)=nonzero(67,6,7)
```

E' importante osservare che per ciascun elemento non nullo della matrice occorre fornire tre informazioni distinte (il valore dell'elemento e degli indici di riga e di colonna) per cui va attentamente esaminata l'opportunità di ricorrere a tale tipo di rappresentazione, in particolar modo quando il numero `ne` di elementi diversi da zero non soddisfi alla condizione: `3×ne>m×n`.

Esistono molti altri modi per comprimere matrici sparse o *bandate* (come le matrici tridiagonali, pentadiagonali, etc.) e per tutti gli altri tipi di matrici "particolari" come quelle simmetriche o emisimmetriche, unitarie etc. Ad esempio, una matrice quadrata simmetrica A di ordine m potrà essere rappresentata dai soli elementi $A_{i,j}$ con $i = 1, \ldots, m$ e $j = i, \ldots, m$ ossia mediante i soli elementi della diagonale principale e quelli al di sopra di essa. Si potrà allo scopo utilizzare un unico vettore `a` di $n = m \times (m+1)/2$ elementi secondo la funzione di accesso:

$$k = f(i, j, k) = (i-1) \times m + j - 1 + (i-1) \times (i-2)/2$$

Tale relazione va applicata per $j \geq i$; nel caso in cui, invece, risulti $j < i$ gli indici i e j potranno essere scambiati essendo la matrice simmetrica. Ad esempio, per $m = 4$ si riportano gli elementi della matrice e la relativa posizione:

posizione:	1	2	3	4	5	6	7	8	9	10
vettore `a`:	`a(1)`	`a(2)`	`a(3)`	`a(4)`	`a(5)`	`a(6)`	`a(7)`	`a(8)`	`a(9)`	`a(10)`
elemento di A:	A_{11}	A_{12}	A_{13}	A_{14}	A_{22}	A_{23}	A_{24}	A_{33}	A_{34}	A_{44}

Secondo questo algoritmo, pertanto, l'accesso all'elemento $A_{i,j}$ avverrà mediante le istruzioni:

```
IF (j>i) swap(i,j)
k = f(i,j,k)
! istruzioni che coinvolgono a(k)
```

E' ovvio che una procedura che operi su matrici immagazzinate in maniera compressa deve essere in grado di gestire in maniera opportuna la particolare struttura dei dati. Procedure sviluppate in modo appropriato possono permettere di risparmiare spazio in memoria e tempo

di macchina. Esistono, comunque, delle situazioni in cui lo sforzo computazionale aggiuntivo richiesto per ”spacchettare” una struttura dati di questo tipo può rallentare l'esecuzione del programma in maniera inaccettabile; in questi casi sarà necessario trovare un giusto compromesso fra le risorse disponibili in termini di memoria e le prestazioni che si vogliono ottenere dal codice di calcolo.

Capitolo 4

Operazioni di I/O interattive

4.1 Generalità

Le operazioni di ingresso/uscita (sovente indicate semplicemente come I/O) costituiscono il mezzo offerto dal linguaggio per realizzare un'interfaccia fra il programma e il mondo esterno.

Del tutto in generale, si può affermare che le operazioni I/O forniscono il mezzo per trasferire *dati* da o verso uno o più *file*. Questo canale di comunicazione opera attraverso una struttura cui si dà il nome di *record*. Come si avrà modo di ripetere più volte nel prosieguo, un record rappresenta niente più che una sequenza di dati (che possono essere *valori* o *caratteri*) mentre un file è una sequenza di tali record. Un file può essere *interno* (ossia ospitato nella memoria centrale dell'elaboratore), oppure *esterno* (e ospitato su un dispositivo di memoria di massa quale una unità disco). Per poter accedere ad un file esterno è necessario operare una *connessione formale* tra una *unità logica*, definita nel programma, ed il file in esame. Un'unità file esterno viene così ad essere identificata, nel programma, da una sorta di *alias* che può essere una espressione costante intera positiva senza segno il cui valore indica l'unità logica, oppure un asterisco ("∗") che identifica una *unità standard* di ingresso o di uscita. Per unità di I/O standard si intende una categoria di unità file da ritenersi implicitamente *preconnesse*: esse sono, tipicamente ma non necessariamente, la tastiera (dispositivo standard di input) ed il terminale video (dispositivo standard di output).

Nel capitolo 8 verranno trattate in maniera esaustiva tutte le diverse possibilità offerte dal Fortran 90/95 per la gestione dei meccanismi di I/O con particolare riferimento alle operazioni su file. Per il momento ci si limiterà ad introdurre i meccanismi cosiddetti *interattivi* di input/output: essi permettono di leggere i dati di cui necessita il programma oppure di visualizzare i risultati del calcolo servendosi unicamente delle unità di ingresso/uscita *standard* definite nel sistema di elaborazione su cui si lavora. Prima, però, di addentrarsi nello studio di tali meccanismi è forse utile premettere una classificazione delle possibili forme di I/O previste dal Fortran, approfittando dell'occasione per dare una definizione di alcuni "aggettivi" che nel prosieguo serviranno a specificare la natura delle operazioni di ingresso/uscita.

Esistono quattro forme di istruzioni di input/output che è possibile usare per trasferire dati da o verso un file:

- *formattate*: i dati sono "editati" o "interpretati" conformemente ad una *specificazione di formato* esplicitamente definita, e le operazioni di I/O avvengono attraverso un flusso di *caratteri* tra la *lista di* I/O ed uno o più record sul file.

- *non formattate*: i dati sono trasferiti tra la lista di I/O ed un record alla volta del file, sottoforma di un flusso di *valori binari*; tale operazione può avvenire solo mediante connessione di un file esterno per cui il meccanismo di I/O non formattato verrà discusso direttamente nel capitolo 8.

- *dirette da lista*: è una forma semplificata di I/O formattata che *non* fa uso di una specificazione di formato dichiarata dal programmatore. Dipendentemente dal tipo di dati da leggere o da scrivere, il trasferimento di dati da o verso una unità file avviene sotto il controllo di una specificazione di formato *di default* (non sempre accurata).

- *con meccanismo namelist*: si tratta di un ulteriore caso particolare di I/O formattato a mezzo del quale i dati sono trasferiti tra una "lista con nome" di dati (*gruppo namelist*) ed uno o più record del file.

Di tutti questi meccanismi si parlerà nelle prossime pagine limitatamente alle operazioni che coinvolgono le unità di ingresso/uscita standard, rimandando lo studio delle operazioni di gestione dei file esterni al capitolo 8.

4.2 Istruzioni di lettura e scrittura

Le operazioni di lettura e scrittura dei dati si realizzano, in Fortran, attraverso le istruzioni `READ` (input), `WRITE` e `PRINT` (output).

In Fortran si possono usare istruzioni di ingresso/uscita nelle quali è possibile indicare esplicitamente una *specificazione di formato*, ossia una descrizione del modo in cui devono essere forniti e interpretati i dati (se si tratta di istruzioni di input) o in cui devono essere visualizzati i risultati (se si tratta di istruzioni di output).

Nel seguito si userà spesso il termine *record* ad indicare un insieme di caratteri rappresentato su un qualsiasi supporto fisico quale, ad esempio, una riga di stampa o una linea di video.

Le più semplici istruzioni di I/O con formato si presentano con la seguente sintassi:

```
READ([UNIT=]unità_logica,[FMT=]formato ) lista_di_ingresso
WRITE([UNIT=]unità_logica,[FMT=]formato ) lista_di_uscita
```

oppure:

```
PRINTformato, lista_di_uscita
```

dove:

- *lista_di_ingresso* è una lista di *variabili* separate da virgole;

- *lista_di_uscita* è una lista di *espressioni* Fortran separate da virgole;

- *unità_logica* rappresenta il "puntatore" ad una unità di input o di output e può essere:
  - un valore intero, nel qual caso le istruzioni `READ` e `WRITE` utilizzeranno le unità di ingresso e di uscita *associate* a tale numero;
  - il simbolo asterisco ("*"), nel qual caso le operazioni di I/O verranno effettuate utilizzando i dispositivi di I/O standard definiti dal sistema di elaborazione utilizzato.

  Si noti che alcuni numeri validi per l'*unità logica* sono riservati proprio alle unità standard (tipicamente l'unità `5` è associata al dispositivo standard di input, `6` al dispositivo standard di output). Ma dal momento che queste associazioni possono variare da un sistema all'altro è bene utilizzare sempre l'asterisco quando si vuole operare con i dispositivi standard, in modo da garantire il corretto funzionamento delle istruzioni di I/O indipendentemente dal sistema di elaborazione adottato.

- *formato* è un *identificatore di formato* che può essere:
  - un asterisco ("*"), in questo caso si dice che la frase di I/O è *guidata da lista*;
  - l'etichetta di una istruzione `FORMAT` che sia presente nella stessa unità di programma in cui compare la frase di I/O;
  - una espressione carattere;
  - il nome di un array di tipo carattere.

La frase `FORMAT` è una istruzione *non eseguibile* che si presenta con la seguente sintassi:

label `FORMAT(`s_f`)`

dove:

- *label* è l'etichetta, obbligatoria, dell'istruzione;
- s_f è una *specificazione di formato* che risulta associata a tutte le istruzioni di ingresso/uscita della stessa unità di programma in cui figura *label* come identificatore di formato.

Alle regole relative alla creazione e ad un uso "maturo" delle specificazioni di formato sarà dedicato il prossimo paragrafo, prima però si anticiperanno alcuni esempi allo scopo di illustrare come sia possibile associare una istruzione di I/O ad una specificazione di formato.

L'effetto della esecuzione di una istruzione `READ` è quello di leggere tanti valori quanti sono i nomi simbolici presenti nella *lista_di_ingresso* e di assegnare gli stessi, ordinatamente, ciascuno alla variabile cui corrisponde secondo la modalità imposta dalla specificazione di formato (quando presente). Così, ad esempio, l'esecuzione del seguente frammento di programma:

```
REAL :: x, y INTEGER :: m ... READ(*,100) m, x, y 100
FORMAT(I2,2(1X,F4.2))
```

consente di inserire, attraverso il dispositivo di ingresso standard (ad esempio la tastiera), tre valori numerici (uno intero e due reali) che verranno interpretati secondo quanto specificato dalla istruzione `FORMAT` ed assegnati ordinatamente alle variabili `m`, `x` e `y`. Così, se ad esempio, i dati forniti dall'utente rappresentassero la seguente *linea* (o *record*) *di ingresso*:

```
11, 3.14, 2.71
```

allora l'istruzione di lettura avrebbe, ai fini della definizione dei valori di `m`, `x` e `y`, lo stesso effetto delle tre istruzioni di assegnazione:

```
m = 11 x = 3.14 y = 2.71
```

Allo stesso modo, l'effetto della esecuzione di una istruzione `WRITE` o `PRINT` è quello di valutare tante espressioni quante ne compaiono nella *`lista_di_uscita`* e di produrre poi detti valori, ordinatamente, sul mezzo di uscita standard (ad esempio il terminale video) in un formato che dipende dal tipo di ciascuno di essi ed in accordo con la specificazione di formato (quando presente). Così, ad esempio, l'esecuzione del seguente frammento di programma:

```
REAL :: x=3.14, y=2.17 CHARACTER(LEN=4) :: str1, str2 str1="Star"
str2="Wars" WRITE(UNIT=*,FMT=100) x, y, x-2*y, str1//str2 100
FORMAT(1X,2(1X,F6.3),1X,E9.2/,A)
```

produce il seguente output:

```
3.140  2.170 -0.12E+01 StarWars
```

Come precedentemente anticipato, un identificatore di formato può essere costituito non solo dalla etichetta di una istruzione `FORMAT` ma anche da una espressione carattere o da un array di caratteri. Si parla in questi casi di *formato contestuale* in quanto la specificazione di formato viene mutuata direttamente dal contesto della istruzione di I/O, senza che venga fatto alcun riferimento ad una apposita frase `FORMAT`. In particolare, se l'identificatore di formato è una espressione carattere, il valore dell'espressione viene valutato in fase di esecuzione e detto valore deve costituire una specificazione di formato valida. Un possibile esempio è il seguente:

```
READ(*,'(I3)') m    ! formato contestuale ''costante''
```

che è una forma perfettamente equivalente alla coppia di istruzioni:

```
READ(*,100) m 100 FORMAT(I3)
```

mentre un ulteriore esempio potrebbe essere questo:

```
CHARACTER(LEN=10) :: formato ... formato = '(1X,F5.2)'
WRITE(*,formato) x    ! formato contestuale ''variabile''
```

Si noti che un formato contestuale variabile è molto più flessibile di quanto possa apparire dal precedente esempio, basti pensare al fatto che la variabile `formato`, che assume il valore `(1X,F5.2)` sul ramo di programma mostrato, su altri rami potrebbe assumere valori diversi. Naturalmente è possibile costruire una lista di specificatori di formato anche attraverso l'esecuzione di operazioni su stringhe:

```
CHARACTER(LEN=12) :: formato ... formato='(1X,F5.2,1X)'
WRITE(*,'(3X//formato(4:11)//',I3)') x,m
```

il cui risultato è analogo a quello che si avrebbe con l'uso della frase `FORMAT`:

```
WRITE(*,250) x,m 250 FORMAT(3X,F5.2,1X,I3)
```

Se, invece, l'identificatore di formato in una frase di I/O è il nome di un array di caratteri, allora la specificazione di formato è il risultato della concatenazione dei valori dei singoli elementi presi secondo l'ordinamento degli array. Un esempio, seppur banale, potrebbe essere il seguente:

```
CHARACTER(LEN=7), DIMENSION(2) :: a ... a(1)='(E10.3,' a(2)='1X,I5)'
WRITE(*,a) x,m   ! identico a WRITE(*,'(E10.3,1X,I5)')
```

Si noti, infine, che le *costanti intere* presenti negli specificatori di formato visti finora possono anche essere sostituite dal nome simbolico di opportune *variabili intere* (con valore definito), a patto che questi nomi siano racchiusi fra i simboli < e >. Così, ad esempio, il seguente frammento di programma:

```
INTEGER :: m=7, n=3, val=50 REAL :: x=3.14 WRITE(*,100) val, x 100
FORMAT(1X,I<m>,1X,F<m>.<n>)
```

risulta perfettamente equivalente a:

```
INTEGER :: val=50 REAL :: x=3.14 WRITE(*,100) val, x 100
FORMAT(1X,I3,1X,F7.3)
```

Dagli esempi proposti si evince che il formato contestuale può risultare molto utile limitatamente a casi di formati semplici oppure in applicazioni sofisticate qualora occorra definire formati variabili.

4.3 Specificazioni di formato

Le informazioni trasmesse durante una operazione di I/O sono rappresentate su record. La struttura di tali record può essere determinata mediante una *specificazione di formato* che, in particolare, individua una suddivisione di ciascun record in uno o più *campi*, ossia gruppi di caratteri ciascuno dei quali contiene un singolo dato; l'*ampiezza di un campo* rappresenta il numero di caratteri che lo compongono. La specificazione di formato descrive completamente il modo in cui ogni dato deve essere scritto (o letto) nel campo che gli è destinato e può inoltre contenere altre informazioni utili per l'interpretazione dei record.

Una specificazione di formato ha la seguente forma:

$(e_1, e_2, \ldots, e_n)$

in cui ciascun termine e_i può essere del tipo:

- $[r]\,d_r$

- d_{nr}
- $[r]s_f$

in cui:

- r è un *contatore di ripetizione* opzionale che, se presente, deve essere una costante intera positiva senza segno; se omesso, il suo valore è assunto pari ad uno;
- d_r è un *descrittore ripetibile*, ossia una stringa di caratteri che specifica l'ampiezza del campo su cui deve essere rappresentata una informazione, il tipo di informazione e la modalità con cui essa deve essere scritta sul record;
- d_{nr} è un *descrittore non ripetibile*, ossia una stringa di caratteri che fornisce al compilatore particolari informazioni sulla composizione e l'interpretazione di un record;
- s_f è a sua volta una *specificazione di formato* (è possibile, pertanto, avere specificazioni di formato *innestate*).

Ciascun termine e_i nella forma rd_r equivale a r descrittori d_r separati da virgole, mentre ciascun termine e_i nella forma rs_f equivale ad una specificazione di formato costituita da r ripetizioni separate da virgole della specificazione di formato s_f privata delle parentesi. Così, ad esempio, la specificazione:

$(e_1,2d_r,e_3)$

equivale a:

(e_1,d_r,d_r,e_3)

mentre la specificazione:

$(e_1,2s_f,e_3)$

in cui s_f indica, a sua volta, la specificazione (e_1,e_2), equivale a:

$(e_1,e_1,e_2,e_1,e_2,e_3)$

In base al *tipo* di informazione rappresentabile nel campo da essi descritto, i descrittori ripetibili previsti nel Fortran 90/95 possono essere suddivisi in *descrittori di tipo numerico*, *carattere* o *logico*. In tabella 4.1 è riportato l'elenco dei descrittori di formato previsti dal linguaggio. Per ciascun descrittore, il simbolo `w` rappresenta una costante intera positiva senza segno indicante l'ampiezza del campo individuato dal descrittore mentre la lettera `I`, `F`, `E`, `G`, `B`, `O`, `Z`, `A`, `L`, indica il tipo di informazione contenuta nel campo (numerico per `I`, `F`, `E`, `G`, `B`, `O`, `Z`, carattere per `A`, logico per `L`). Il significato dei simboli `d` ed `m` (che sono, comunque, costanti intere non negative senza segno) e di `e` (costante intera positiva senza segno) sarà chiarito nei prossimi paragrafi. In tabella 4.2, invece, è riportato l'elenco dei descrittori non ripetibili previsti dal linguaggio; anche a questi sarà dedicato uno dei prossimi paragrafi.

Si vuole concludere questa premessa facendo notare che ogni elemento della lista di una frase di I/O nella quale sia indicata una specificazione di formato è associato ad un descrittore ripetibile; il descrittore stabilisce la modalità secondo cui deve avvenire la *conversione* del dato dalla sua *rappresentazione interna* in memoria come valore dell'elemento della lista di I/O alla sua *rappresentazione esterna* sul record, o viceversa. I descrittori non ripetibili presenti nella specificazione di formato non sono, invece, associati ad alcun elemento della lista di I/O bensì servono a controllare la *posizione* in un record o in un file.

4.3.1 Rappresentazione dei dati numerici sui record di I/O

I valori numerici vengono tipicamente rappresentati sui record di I/O in *base dieci* mentre sono rappresentati in memoria nel sistema di numerazione adottato dall'elaboratore. Pertanto le operazioni di I/O di un dato numerico implica l'esecuzione di un algoritmo di conversione da un sistema di numerazione ad un altro e l'eventuale troncamento o arrotondamento della mantissa per i valori reali.

Descrittori ripetibili di tipo numerico	*Descrittori ripetibili di tipo logico*	*Descrittori ripetibili di tipo carattere*
`Iw [.m]`	`Lw`	`A`
`Fw.d`		`Aw`
`Ew.d [Ee]`		
`ENw.d [Ee]`		
`ESw.d [Ee]`		
`Gw.d [Ee]`		
`Bw [.d]`		
`Ow [.d]`		
`Zw [.d]`		

Tabella 4.1: Descrittori ripetibili nel Fortran 90/95

n`X`	salto di caratteri
'$h_1 \ldots h_n$'	inserimento di stringa
$h_1 \ldots h_n$	di caratteri
/	fine record
Tn	descrittori di tabulazione
TLn	
TRn	
:	interruzione per esaurimento lista

Tabella 4.2: Descrittori non ripetibili nel Fortran 90/95

In questo paragrafo si esporranno alcune regole di carattere generale riguardanti la rappresentazione esterna dei dati numerici sui record di I/O.

- *In input* i caratteri *blank* nelle prime colonne di un campo destinato alla rappresentazione di un dato numerico *non* sono significativi così come *non* è significativo un campo di soli *blank*. Gli eventuali spazi vuoti presenti nel campo *dopo* il primo carattere non vuoto vengono ignorati a meno che la interpretazione di tali caratteri non venga opportunamente specificata dal programmatore.

- *In output* i dati numerici vengono sempre scritti allineati *a destra* nel campo loro riservato, eventualmente preceduti da spazi vuoti, ossia sono rappresentati in modo tale che l'ultima cifra del numero intero occupi l'ultima colonna a destra del campo. Il segno precede il primo carattere non vuoto; in particolare, il segno `+` è considerato opzionale e la sua presenza dipende dal particolare sistema di elaborazione.

- *In output* un campo destinato alla rappresentazione di un numero viene riempito di asterischi se, tolti i caratteri opzionali, il dato non può essere rappresentato secondo le modalità dettate dal descrittore. Così, ad esempio, un campo di ampiezza due destinato a contenere il valore `-12` oppure `134` verrebbe riempito di asterischi dal momento che in entrambi i casi il dato da rappresentare richiede tre caratteri (se però il dato da rappresentare fosse `+12` allora il campo sul record di uscita conterrebbe la stringa `12` ed il segno `+` non verrebbe riportato indipendentemente dalla convenzione in atto al riguardo).

4.3.2 Interazione fra lista di ingresso/uscita e specificazione di formati

L'esecuzione di una frase di I/O con formato avviene scandendo insieme, da sinistra a destra, la lista di I/O e la specificazione di formato. L'acquisizione dei dati o la creazione di record di uscita è contemporanea alla scansione della lista e della specificazione di formato.

Per quanto concerne una operazione di ingresso, essa inizia con la trasmissione in memoria del record che deve essere letto (*record corrente*). Successivamente inizia la scansione della specificazione di formato. Possono presentarsi i seguenti casi:

- Se, durante la scansione della specificazione di formato, si incontra un descrittore ripetibile che definisce un campo di ampiezza `w`, si controlla se è stato letto ed assegnato un valore a tutti gli elementi della lista. In caso affermativo l'operazione di ingresso ha termine; altrimenti essa prosegue con l'acquisizione di un dato dal record corrente ossia viene letta una stringa di `w` caratteri che, interpretata in base al tipo di descrittore, costituisce il valore da assegnare al primo elemento della lista di ingresso il cui contenuto non è stato ancora definito. Il procedimento testé descritto può essere realizzato correttamente soltanto se il dato sul record di ingresso è nella forma prevista dal descrittore e se quest'ultimo è del tipo opportuno in relazione al tipo dell'elemento della lista di ingresso. Tutte le situazioni che impediscono il corretto svolgimento dell'operazione provocano l'interruzione dell'intero programma a meno che tali condizioni di errore non vengano opportunamente previste e gestite.

- Se, durante la scansione della specificazione di formato, viene incontrato un descrittore non ripetibile viene eseguita l'operazione da esso descritta e la scansione della specificazione di formato procede con l'esame del successivo descrittore.

- Quando, durante la scansione della specificazione di formato, si incontra la parentesi chiusa finale, l'operazione di ingresso termina se la sua esecuzione ha permesso di definire il contenuto di tutti gli elementi della lista. In caso contrario, l'acquisizione dei dati prosegue dal record che, sull'unità di ingresso, segue immediatamente il record corrente. La scansione della specificazione di formato riprende a partire dal *punto di riscansione* che coincide con l'inizio della specificazione di formato se questa non contiene specificazioni di formato interne; altrimenti esso coincide con l'inizio della specificazione di formato che si chiude con l'ultima parentesi interna destra.

Al fine di chiarire quest'ultimo punto, si guardino le seguenti specificazioni di formato in cui la freccia indica proprio il punto di riscansione:

```
( I 3,2A3,I2)
(I3, 2 (A3,I2))
(I2, 3 (1X,A3,I2),F5.3)
(L1, 2 (2(1X,A3,I5),F5.3))
(1X,2(F5.7,I3,A6), 3 (1X,A3,I5),E10.4)
```

Per quanto concerne, invece, l'esecuzione di una frase di uscita formatata, essa consiste nella creazione in memoria di uno o più record che successivamente vengono trasmessi all'unità di uscita. Il numero e la struttura dei record creati dipendono dalla lista di uscita, dalla specificazione di formato e dalla loro interazione. La scansione della lista e del formato procede nello stesso modo visto per le operazioni di input; in particolare, ogni qualvolta nella specificazione di formato si incontra un descrittore ripetibile, il valore dell'elemento della lista ad esso corrispondente viene convertito e rappresentato sul record corrente secondo le modalità imposte dal descrittore. Anche per le operazioni di uscita deve essere rispettata la corrispondenza di tipo fra elemento della lista di uscita e descrittore ripetibile ad esso associato.

4.4 Descrittori ripetibili

4.4.1 Il descrittore `I`

Il descrittore di formato `I` è utilizzato per controllare il formato di rappresentazione dei valori interi. La sua forma generale è:

```
Iw [.m]
```

dove `w` rappresenta la *larghezza di campo*, ossia il numero di caratteri da utilizzare per rappresentare i dati di input e output, mentre `m` rappresenta il *numero minimo* di cifre da visualizzare.

In input non vi è alcuna differenza fra i descrittori `Iw.m` e `Iw`; inoltre non ha importanza la posizione in cui viene inserito il valore intero (nel senso che eventuali bianchi di testa vengono completamente ignorati) purché il valore venga inserito *all'interno* del campo di `w` caratteri ad esso riservato.

In output il dato intero viene rappresentato in un campo di `w` caratteri con almeno `m` cifre, di cui le prime uguali a zero se il numero da rappresentare è composto da meno di `m` cifre. Se, però, l'ampiezza di campo `w` non è sufficiente per contenere tutte le cifre del valore intero da produrre, il campo sarà riempito con *w* asterischi.

Come esempio di istruzione di uscita formattata si consideri il seguente ramo di codice:

```
INTEGER :: a = 3, b = -5, c = 128 ... WRITE(*,"(3I4.2)")  a, b, c
```

Il descrittore di formato utilizzato è `3I4.2` che è equivalente a `(I4.2,I4.2,I4.2)` per via della presenza del contatore di ripetizione `3`. Pertanto, ciascuna delle tre variabili intere verrà stampata in accordo con lo specificatore `I4.2` ottenendosi il risultato seguente:

```
bb03b-05b128
```

dove con il simbolo *b* si è indicato uno spazio bianco mentre la presenza degli zeri è giustificata dalla presenza del termine ".2" in coda allo specificatore che richiede un minimo di `2` cifre significative.

Come ulteriore esempio, si osservino le seguenti istruzioni:

```
INTEGER :: i=-12, j=4, number=-12345 WRITE(*,100) i, i+12, j, number
WRITE(*,110) i, i+12, j, number WRITE(*,120) i, i+12, j, number 100
FORMAT(1X,2I5,I7,I10) 110 FORMAT(1X,2I5.0,I7,I10.8) 120
FORMAT(1X,2I5.3,I7,I5)
```

Il risultato prodotto questa volta è, chiaramente, il seguente:

```
  -12    0      4    -12345
  -12           4 -00012345
 -012  000      4*****
<-5-><-5-><--7--><-5->
                <---10--->
```

dove la presenza degli asterischi che riempiono l'intero campo finale dipende dal fatto che il valore intero `-12345` richiede sei caratteri ossia uno in più rispetto a quanti ne riserva lo specificatore `I5` utilizzato per la variabile `number` nell'ultima istruzione `FORMAT` (a margine dell'ultimo record si sono evidenziate graficamente le ampiezze dei campi associati a ciascuno specificatore di formato in modo da rendere più chiara l'interpretazione del risultato).

Nelle istruzioni di ingresso, invece, i valori interi possono essere inseriti ovunque all'interno dei campi ad essi destinati e saranno letti sempre correttamente. Così, ad esempio, data l'istruzione di lettura:

```
INTEGER :: i, j, k ... READ(*,'(2I5,I10)')
```

l'inserimento del seguente record di ingresso:

```
10    25      50000 <-5-><-5-><---10--->
```

ha l'effetto di assegnare alle variabili `i`, `j` e `k`, rispettivamente, i valori `10`, `25` e `50000`.

4.4.2 I descrittori `B`, `O` e `Z`

I descrittori di formato `B` , `O` e `Z` definiscono un campo per dati da rappresentare in forma, rispettivamente, binaria (ossia in termini delle sole cifre `0` e `1`), esadecimale (ossia in termini delle sole cifre da `0` a `7`) oppure ottale (ossia in termini delle cifre da `0` a `9` o dei caratteri da `A` ad `F` o da `a` ad `f`). Essi forniscono, così, una conversione fra la rappresentazione binaria *interna* e le corrispondenti rappresentazioni *esterne* binaria, esadecimale oppure ottale. Le rispettive forme di questi specificatori sono:

```
Bw [.m]
Ow [.m]
Zw [.m]
```

dove `w` rappresenta la larghezza di campo, ossia il numero di caratteri da utilizzare per rappresentare i dati di input e output, mentre `m` rappresenta il numero minimo di cifre da visualizzare. Il valore di `m` è ignorato in input ed ha effetto solo per operazioni di uscita. Se `m` non viene specificato si assume per esso un valore di default unitario; se `m` è maggiore di `w` il campo sarà riempito di asterischi.

Il campo di uscita relativo al descrittore `Bw` [risp. `Ow`, `Zw`] è una stringa di zero o più *bianchi di testa* seguiti dalle cifre corrispondenti alla rappresentazione binaria [risp. esadecimale, ottale] del valore da rappresentare esclusi gli *zeri di testa*.

Il campo di uscita relativo al descrittore `Bw.m` [risp. `Ow.m`, `Zw.m`] è identico a quello relativo al corrispondente descrittore `Bw` [risp. `Ow`, `Zw`] salvo che per il fatto che la stringa di uscita è composta di *almeno* `m` cifre; se necessario, detta stringa sarà riempita di *zeri di testa*.

Il prossimo frammento di programma rappresenta un esempio di istruzioni di output formattate facenti uso degli specificatori in esame:

```
INTEGER :: a=21, b=-1 WRITE(*,100) "Formato binario:     ", a, b
WRITE(*,200) "Formato ottale:      ", a, b WRITE(*,300) "Formato
esadecimale: ", a, b 100 FORMAT(1X,A,2(1X,B12)) 200
FORMAT(1X,A,2(1X,O12.4)) 300 FORMAT(1X,A,2(1X,Z12))
```

Il corrispondente risultato è:

```
Formato binario:             10101 111111111111 Formato ottale:
0025  37777777777 Formato esadecimale:            15     FFFFFFFF
                        <----12----><----12---->
```

4.4.3 Il descrittore F

Il descrittore di formato F è utilizzato per controllare il formato di rappresentazione dei valori reali secondo la *notazione decimale*. La sua sintassi è:

```
Fw.d
```

dove w rappresenta la larghezza di campo, ossia il numero di caratteri da utilizzare per rappresentare i dati di input e output, mentre d rappresenta il numero di cifre da visualizzare a destra del punto decimale. Se necessario, il numero sarà arrotondato prima di essere rappresentato; ciò accade, chiaramente, quando il numero rappresentato richiede meno cifre significative della sua rappresentazione interna. Al contrario, se il numero rappresentato richiede più cifre significative della sua rappresentazione interna, saranno aggiunti degli zeri extra alla destra del numero.

La dimensione del campo w deve essere valutata con molta attenzione quando si fa uso del descrittore F nelle operazioni di uscita; in particolare, detto n il numero di cifre della parte intera del valore da produrre, si deve far sì che w e d soddisfino alla condizione:

$$\mathtt{w} \geq n + \mathtt{d} + 2$$

ossia, fatte salve le n cifre per la parte intera, per stampare un valore con d cifre decimali occorre un campo di almeno d+2 caratteri (infatti occorrono, oltre ai d caratteri per le cifre che seguono la virgola, anche *un carattere* per l'eventuale *segno*, ed *uno* per il *punto decimale*). Nel caso in cui questa condizione non sia soddisfatta, il campo destinato all'output del valore reale sarà riempito di asterischi.

Un esempio di utilizzo del descrittore F in operazioni di uscita è fornito dalle seguenti istruzioni:

```
REAL :: x = 12.34, y = -0.945, z = 100.0 ... WRITE(*,"(3F6.2)")  x,
y, z
```

Il descrittore di formato utilizzato è 3F6.2 che è equivalente a (F6.2,F6.2,F6.2) per via della presenza del contatore di ripetizione 3. Pertanto, ciascuna delle tre variabili reali verrà stampata in accordo con lo specificatore F6.2 ossia nel formato a virgola fissa in un campo di sei caratteri di cui due destinati ad ospitare le prime due cifre dopo la virgola. Il risultato prodotto sarà, pertanto:

```
 b12.34b-0.95100.00
```

dove, evidentemente, il valore di y viene troncato dell'ultima cifra decimale in accordo con quanto imposto dallo specificatore di formato adottato, così come al valore di z viene aggiunto "in coda" uno zero supplementare. Ancora una volta con il simbolo b si è voluto rappresentare un carattere *blank*. Come ulteriore esempio di operazioni di uscita formattate che impieghino il descrittore F si guardi il seguente frammento di codice:

```
REAL :: x=-12.3, y=.123, z=123.456 WRITE(*,100) x,y,z WRITE(*,200)
x,y,z WRITE(*,300) x,y,z 100 FORMAT(1X,2F6.3,F8.3) 200
FORMAT(1X,3F10.2) 300 FORMAT(1X,F7.2,F6.3,F5.0)
```

il cui risultato è, evidentemente, il seguente:

```
****** 0.123 123.456
    -12.30      0.12    123.46
 -12.30 0.123 123.
```

Il primo campo, riempito di asterischi, si spiega considerando che il valore da rappresentare, `-12.3` richiede tre caratteri per la rappresentazione della parte intera (`-12`) mentre il particolare specificatore adottato dalla prima istruzione `FORMAT` ne riserva appena due (dei `6` associati allo specificatore `F6.3`, infatti, ne vanno sottratti `3` per la parte decimale ed uno per il punto decimale). L'ultimo valore prodotto, invece, si caratterizza per la totale assenza di una parte decimale, in accordo con lo specificatore `F5.0` che, per la stampa del valore cui è riferito, riserva un campo di `5` caratteri di cui `0` per la parte decimale.

L'interpretazione dei dati reali in istruzioni di ingresso è lievemente più complessa. Il valore di input in un un campo `F` può essere un numero reale qualsiasi espresso secondo la notazione in virgola fissa o mobile o un numero senza punto decimale. In particolare, se il valore è inserito nelle prime due forme, esso è interpretato sempre correttamente indipendentemente dalla posizione che occupa all'interno del campo di input. Così, ad esempio, considerata la seguente istruzione di input:

```
REAL :: x, y, z ... READ(*,150) x,y,z 150 FORMAT(3F10.4)
```

supponendo che il record di ingresso sia il seguente:

```
2.5         0.254E+01 25.E-01 <---10---><---10---><---10--->
```

tutte e tre le variabili di ingresso conterranno il valore `2.5` (ai piedi del precedente record di ingresso si sono indicate schematicamente anche le dimensioni di ciascun campo, al fine di rendere più evidente la formattazione del record). Se, invece, all'interno del campo appare un numero *senza* un punto decimale, si suppone che questo occupi la posizione specificata dal termine `d` del descrittore di formato. Ad esempio, se il descrittore di formato è `F10.4`, si suppone che le quattro cifre più a destra del numero costituiscano la parte decimale del valore di input mentre le restanti cifre sono considerate come la parte intera del valore di input. Così, ad esempio, considerata la seguente istruzione di input:

```
REAL :: x, y, z ... READ(*,150) x,y,z 150 FORMAT(3F10.4)
```

supponendo che il record di ingresso sia il seguente:

```
         25    250        25000
<---10---><---10---><---10--->
```

le variabili `x`, `y` e `z` conterrebbero, rispettivamente, i valori `0.0025`, `0.0250` e `2.5000` Questo semplice esempio dovrebbe essere sufficiente a far comprendere come l'impiego, nei record di ingresso, di valori reali senza il punto decimale possa generare una notevole confusione per cui non dovrebbe mai aver luogo.

4.4.4 Il descrittore `E`

Il descrittore di formato `E` è utilizzato per controllare il formato di rappresentazione dei valori reali secondo la *notazione esponenziale*. Il suo formato è:

`Ew.d [Ee]`

dove `w` rappresenta la larghezza di campo, ossia il numero di caratteri da utilizzare per rappresentare i dati di input e output, `d` rappresenta il numero di cifre da visualizzare a destra del punto decimale, mentre `e` rappresenta il numero di cifre nell'esponente (si noti che, per default, ad `e` viene attribuito valore due). Si faccia attenzione al fatto che i valori reali rappresentati secondo la notazione esponenziale con il descrittore `E` sono "normalizzati" ossia sono rappresentati con una mantissa compresa tra 0.1 e 1.0 moltiplicata per una potenza di 10. La dimensione dei campi dei numeri reali deve essere valutata con molta attenzione quando si fa uso del descrittore `E`. Infatti, nel caso in cui il *suffisso* `Ee` sia *assente* è necessario che `w` e `d` soddisfino alla condizione:

$\texttt{w} \geq \texttt{d} + 7$

ossia, per stampare una variabile con `d` cifre significative occorre un campo di almeno `d`+7 caratteri (infatti occorrono, oltre a `d` caratteri per le cifre significative della mantissa, anche un carattere per il segno della mantissa, due per lo zero e il punto decimale, uno per la lettera `E`, uno per il segno dell'esponente e due per l'esponente). Nel caso in cui questa condizione non sia soddisfatta, il campo destinato all'output del reale sarà riempito di asterischi. Analogamente, la forma "completa" del descrittore, che prevede anche il gruppo `Ee` implica il rispetto della condizione:

$\texttt{w} \geq \texttt{d} + \texttt{e} + 5$

Si noti che nel caso in cui il numero da rappresentare sia positivo la larghezza minima del campo può essere diminuita di un'unità (così, ad esempio, in assenza del suffisso `Ee` basta che la larghezza `w` del campo sia di soli sei caratteri più grande di `d`). Si noti, infine, che alcuni compilatori eliminano lo zero iniziale per cui, in questi casi, è richiesta una colonna in meno.

Come esempio, si guardino le seguenti istruzioni:

```
REAL :: a=1.2345E6, b=0.01, c=-77.7E10, d=-77.7E10 WRITE(*,100)
a,b,c,d 100 FORMAT(1X,2E14.4,E13.6,E11.6)
```

le quali, come è logico attendersi, generano il seguente risultato:

```
     0.1235E+07    0.1000E-01-0.777000E+12**********
 <-----14-----><-----14-----><-----13----><----11--->
```

(ai piedi del precedente record di uscita si sono indicate schematicamente anche le dimensioni di ciascun campo, al fine di rendere più evidente la formattazione del risultato).

Si supponga ora di voler rappresentare in uscita il valore complesso $\texttt{z} = 4.25 + j78.3$ Dal momento che un numero complesso è costituito da una coppia di valori reali, il relativo formato viene specificato mediante due successivi descrittori di formato validi per i reali, rispettivamente per la parte reale ed il coefficiente della parte immaginaria. Pertanto le istruzioni:

```
100 FORMAT(1X,E10.3,E14.4) PRINT100, z
```

costituiscono una valida possibilità per la rappresentazione del valore complesso in esame, producendo la stampa del seguente record:

```
 0.425E+01    0.7830E+02
<---10---><-----14----->
```

4.4.5 Nota sui descrittori reali

E' bene precisare che *nelle operazioni di input i descrittori reali sono fra loro equivalenti*: il dato può essere rappresentato, nel campo di ampiezza `w`, nella forma di costante Fortran intera o reale. Se il dato è nella forma di costante reale con esponente, l'esponente può essere scritto come una costante intera con segno non preceduta dalla lettera `E`. Se nel campo è presente il punto decimale allora il valore di `d` specificato nel descrittore non ha alcun significato; in caso contrario il dato viene interpretato come se fosse presente un punto decimale immediatamente prima delle `d` cifre più a destra della stringa di caratteri che rappresenta il numero, escluso l'esponente. A titolo esemplificativo, si riporta di seguito l'interpretazione che viene data del contenuto di un campo di ampiezza cinque in corrispondenza dei diversi descrittori reali (con il simbolo *b* si è indicato un carattere blank):

Campo	*Descrittore*	*Interpretazione*
*bb*2.5	F5.0	2.5
2.5*bb*	F5.2	2.5
-2.5*b*	F5.4	-2.5
.2E01	F5.2	2.
.7E-3	E5.1	0.0007
3.1+4	E5.1	31000.
*bb*3*bb*	F5.2	0.03

dove l'ultimo risultato si spiega considerando che, di default, gli spazi bianchi vengono ignorati.

La libertà consentita per la preparazione del record di ingresso suggerisce di impiegare per la lettura dei valori reali indifferentemente gli specificatori `E` o `F`, definendo in `w` il campo assegnato al dato e fissando arbitrariamente il valore di `d`: in tal modo si consente in pratica di preparare il supporto di ingresso in qualsiasi formato contenente il punto decimale. Così, ad esempio, i valori che seguono, tutti registrati in campi di `10` caratteri:

```
1.
1.2E-3
  1.2
  1.23E+01
 1.23456
<---10--->
```

possono essere letti indifferentemente con il formato `E10.1`, oppure `E10.2`, oppure ancora `E10.3`, etc.

Viceversa, *nelle operazioni di output i descrittori reali non sono fra loro equivalenti* in quanto definiscono modalità diverse di rappresentazione esterna di un numero reale nel campo di ampiezza `w`. In ogni caso, indipendentemente dal tipo di rappresentazione interna del dato, la sua rappresentazione esterna contiene `d` cifre dopo il punto decimale di cui l'ultima ottenuta per arrotondamento.

4.4.6 Il descrittore `ES`

Il descrittore di formato `ES` è utilizzato per controllare il formato di rappresentazione dei valori reali secondo la *notazione scientifica*. Secondo questa notazione un numero viene espresso con una mantissa compresa tra 1.0 e 10.0 per una potenza di 10 mentre, come si ricorderà, secondo la notazione esponenziale guidata dal descrittore `E`, un numero viene espresso con una mantissa compresa tra 0.1 e 1.0 per una potenza di 10. Fatta salva questa eccezione, non vi è altra differenza fra i descrittori `ES` ed `E`. Il descrittore di formato `ES` ha la seguente forma generale:

`ESw.d` [`Ee`]

dove, come sempre, `w` rappresenta la larghezza di campo, ossia il numero di caratteri da utilizzare per rappresentare i dati di input e output, `d` è il numero di cifre da visualizzare a destra del punto decimale mentre `e` rappresenta il numero di cifre nell'esponente (pari a due per default). La formula per ottenere la larghezza minima di un campo con formato `ES` è uguale a quella del descrittore `E`, con la differenza che `ES` può rappresentare una cifra significativa in più a parità di lunghezza dal momento che lo zero a sinistra del punto decimale viene sostituito da una cifra significativa. La larghezza del campo, `w`, deve soddisfare alla seguente condizione:

$\mathtt{w} \geq \mathtt{d} + 7$

in caso contrario il campo sarà riempito di asterischi. Se il numero da rappresentare è positivo, la condizione è "rilassata" nel senso che la larghezza `e` del campo deve risultare di almeno sei caratteri più grande di `d`. Chiaramente se il descrittore è presente nella sua forma completa con il suffisso `Ee` la nuova condizione cui deve soddisfare la larghezza del campo `w` diventa:

$\mathtt{w} \geq \mathtt{d} + \mathtt{e} + 5$

A titolo di esempio, si guardino le istruzioni:

```
REAL :: a=1.2345E6, b=0.01, c=-77.7E10 WRITE(*,100) a,b,c 100
FORMAT(1X,2ES14.4,ES12.6)
```

le quali forniscono il seguente output:

```
     1.2345E+06     1.0000E-02************
 <-----14-----><-----14-----><----12---->
```

4.4.7 Il descrittore EN

Il descrittore di formato EN è utilizzato per controllare il formato di rappresentazione dei valori reali secondo la *notazione ingegneristica*. Secondo questa notazione un numero viene espresso con una mantissa compresa tra 1.0 e 1000 per una potenza di 10 con esponente divisibile per 3. Il descrittore di formato EN ha la seguente forma generale:

```
ENw.d [Ee]
```

dove, come per gli altri descrittori di formato E ed ES, w rappresenta la larghezza di campo, ossia il numero di caratteri da utilizzare per rappresentare i dati di input e output, d è il numero di cifre da visualizzare a destra del punto decimale mentre e rappresenta il numero di cifre nell'esponente (pari a due per default). Tutte le considerazioni svolte per i descrittori E ed ES a proposito della dimensione minima del campo e delle particolarità di rappresentazione valgono identicamente anche per il descrittore EN.

Come esempio, le seguenti istruzioni:

```
REAL :: a=1.2345E6, b=0.01, c=-77.7E10 WRITE(*,100) a,b,c 100
FORMAT(1X,2EN14.4,EN12.6)
```

forniscono in uscita il seguente record:

```
     1.2345E+06    10.000E-03***********
 <-----14-----><-----14-----><----12---->
```

4.4.8 Il descrittore L

Il descrittore L individua un campo di ampiezza w destinato alla rappresentazione di un valore logico. Il suo formato è:

```
L [.w]
```

In ingresso il valore "vero" deve essere introdotto in una delle forme T o TRUE mentre il valore "falso" in una delle forme F o FALSE; questi caratteri possono essere seguiti da un qualunque altro carattere nel campo di ampiezza w ad essi riservato. In uscita i valori "vero" e "falso" sono rappresentati rispettivamente dai caratteri T e F preceduti da $w - 1$ caratteri bianchi.

Un esempio di utilizzo del descrittore L è fornito dal seguente blocco di istruzioni:

```
LOGICAL :: output=.TRUE., debug=.FALSE. WRITE(*,10) output,debug 10
FORMAT(1X,2L5)
```

il cui risultato è, chiaramente:

```
    T    F
<-5-><-5->
```

dove i caratteri T ed F sono riportati entrambi nella forma normalizzata a destra ciascuno all'interno di un campo di cinque caratteri. Allo stesso modo, il seguente frammento di programma:

```
LOGICAL :: cond1,cond2 ... READ(*,"(2L10)")  cond1, cond2
```

mostra un esempio di utilizzo del descrittore `L` in istruzioni di ingresso. In tal caso sono attesi due valori logici in altrettanti campi di ampiezza `10` così, ad esempio, ciascuno dei seguenti record di ingresso è formalmente valido ed ha lo scopo di assegnare a `cond1` il valore `.TRUE.` e a `cond2` il valore `.FALSE.`

```
  T          February
.TRUE.        .FALSE.
 Truman    falso
 .T.             .F.
<---10---><---10--->
```

4.4.9 Il descrittore `A`

Il descrittore di formato `A` controlla la rappresentazione esterna dei dati di tipo carattere. Il suo formato è:

```
A [.w]
```

Il descrittore `A` rappresenta i caratteri in un campo la cui lunghezza è pari al numero di caratteri da rappresentare. Il descrittore `A.w` rappresenta i caratteri in un campo di lunghezza fissa pari a `w`. Se la larghezza `w` del campo è maggiore della lunghezza della variabile stringa da rappresentare, la variabile viene rappresentata con i caratteri *allineati a destra* all'interno del campo. Se, invece, la larghezza `w` del campo è minore della lunghezza della variabile, vengono rappresentati all'interno del campo soltanto i primi `w` caratteri della variabile. Quanto detto si applica sia in fase di ingresso che in fase di uscita.

Il descrittore `A` in cui non compare l'ampiezza del campo può risultare molto utile per l'output di dati di tipo carattere il cui valore viene riprodotto *per intero* sul record di uscita. Nelle istruzioni di ingresso, invece, il descrittore `A` legge esattamente tanti caratteri quanti ne può contenere la variabile letta.

L'esempio che segue aiuterà a comprendere meglio l'utilizzo dello specificatore `A` nelle frasi di uscita:

```
CHARACTER(LEN=17) :: stringa="Ecco una stringa" WRITE(*,10) stringa
WRITE(*,11) stringa WRITE(*,12) stringa 10 FORMAT(1X,A) 11
FORMAT(1X,A20) 12 FORMAT(1X,A6)
```

Il corrispondente output è:

```
 <------16------>
 Ecco una stringa
     Ecco una stringa
 Ecco u
 <--------20--------->
```

Come ulteriore esempio si può considerare il seguente blocco di istruzioni:

```
CHARACTER(LEN=7), PARAMETER :: form="(A5,A2)" CHARACTER(LEN=4) ::
parola CHARACTER(LEN=1) :: lettera !... parola = "casa" lettera =
"y" WRITE(*,form) parola, lettera
```

cui consegue la stampa del record:

```
bcasaby
```

dove, come di consueto, il simbolo `b` rappresenta un *blank*.

Analogamente a quanto avviene con le istruzioni di uscita, anche nelle operazioni di ingresso se la larghezza `w` del campo è maggiore della lunghezza della variabile di caratteri, nella variabile vengono registrati soltanto i caratteri più a destra del campo di input. Se, invece, la larghezza del campo è minore della lunghezza della variabile di caratteri, i caratteri del campo vengono registrati a partire da sinistra nella variabile e lo spazio restante sarà riempito di spazi bianchi. Ad esempio, date le seguenti istruzioni:

```
CHARACTER(LEN=6)  :: str1 CHARACTER(LEN=8)  :: str2
CHARACTER(LEN=10) :: str3 CHARACTER(LEN=12) :: str4 ...
READ(*,'(A)')   str1 READ(*,'(A10)') str2 READ(*,'(A10)') str3
READ(*,'(A10)') str4
```

nel caso in cui i rispettivi record di ingresso fossero:

```
abcdefghij abcdefghij abcdefghij abcdefghij <---10--->
```

l'effetto delle frasi di ingresso sarebbe equivalente a quello delle seguenti istruzioni di assegnazione:

```
str1="abcdef" str2="cdefghij" str3="abcdefghij" str4="abcdefghij  "
```

4.4.10 Il descrittore `G`

Il descrittore di formato `G` è un particolare descrittore (*generalizzato*) atto ad individuare un campo destinato alla rappresentazione di un dato di tipo qualsiasi. Per dati di tipo `INTEGER`, `LOGICAL` o `CHARACTER`, non c'è nessuna differenza fra il descrittore `Gw.d` e, rispettivamente, i descrittori `Iw`, `Lw` e `Aw`. Per quanto concerne, invece, la rappresentazione di dati di tipo `REAL` (o `COMPLEX`) le cose appaiono un pò più complicate. Nelle operazioni di ingresso il descrittore `G` è del tutto equivalente ai descrittori `F` ed `E`; in uscita, invece, esso si comporta in maniera differente a seconda dell'ordine di grandezza del valore numerico x da rappresentare nel campo di ampiezza `w`. In particolare:

- Se risulta $10^{\mathtt{d}} < |x| < 0.1$, i descrittori `Gw.d` e `Gw.dEe` sono equivalenti a `Ew.d` ed `Ew.dEe`, rispettivamente.

- Se risulta $10^{i-1} \leq |x| < 10^i$, con $0 \leq i \leq$ `d`, allora si ottiene una rappresentazione esterna uguale a quella prodotta da un descrittore `F`$\widetilde{\texttt{w}}$`.`$\widetilde{\texttt{d}}$ dove $\widetilde{\texttt{d}}$ è pari a `d` $- i$ e $\widetilde{\texttt{w}}$ è pari a `w` $- k$, seguita da k caratteri vuoti, dove $k = 4$ per il descrittore `Gw.d`, mentre $k =$ `e` $+ 2$ per il descrittore `Gw.dEe`.

La seguente tabella mostra alcuni esempi di output prodotti usando il descrittore `Gw.d` confrontando gli stessi con quelli che si otterrebbero mediante l'utilizzo dell'equivalente descrittore `F` (con il carattere *b* si è indicato un carattere *blank*):

Valore	Formato	Output con `G`	Formato	Output con `F`
0.01234567	G13.6	*b*0.123457E-01	F13.6	*bbbbb*0.012346
-0.12345678	G13.6	-0.123457*bbbb*	F13.6	*bbbb*-0.123457
1.23456789	G13.6	*bb*1.23457*bbbb*	F13.6	*bbbbb*1.234568
12.34567890	G13.6	*bb*12.3457*bbbb*	F13.6	*bbbb*12.345679
123.45678901	G13.6	*bb*123.457*bbbb*	F13.6	*bbb*123.456789
-1234.56789012	G13.6	*b*-1234.57*bbbb*	F13.6	*b*-1234.567890
12345.67890123	G13.6	*bb*12345.7*bbbb*	F13.6	*b*12345.678901
123456.78901234	G13.6	*bb*123457.*bbbb*	F13.6	123456.789012
-1234567.89012345	G13.6	-0.123457E+07	F13.6	**********

4.5 Descrittori non ripetibili

4.5.1 Il descrittore *n*`X`

Questo descrittore, in cui ***n*** è una costante intera positiva senza segno che *non* può essere omessa neanche se uguale ad uno, consente di spostarsi esattamente di ***n*** caratteri sul record corrente. L'utilizzazione del descrittore ***n***`X` permette, dunque, di ignorare ***n*** caratteri su un record in ingresso o di creare un campo di ***n*** caratteri blank su un record di uscita. Normalmente, però, questo descrittore viene utilizzato nelle specificazioni di formato associate a istruzioni di uscita per spaziare opportunamente i risultati oppure per creare un blank come primo carattere del record. Ad esempio la specificazione di formato:

```
(1X,3(A7,I3))
```

consente di non perdere, in molti casi, il primo carattere nella stampa del record cui la specificazione di formato è riferita (il senso di quanto detto apparirà più chiaro leggendo il paragrafo relativo ai *caratteri di controllo*).

Come ulteriore esempio, si consideri la seguente istruzione di scrittura:

```
PRINT 111, n,x,m,y 111 FORMAT(5X,I4,5X,E13.6)
```

in cui si suppone che le variabili `n` ed `m` siano di tipo intero mentre `x` ed `y` siano di tipo reale. L'esecuzione dell'istruzione precedente avviene nel modo seguente:

- Il primo descrittore incontrato nella scansione di formato, `5X`, fa in modo che le prime cinque posizioni sul record di uscita siano riempite con caratteri blank.

- Il secondo descrittore è il descrittore ripetibile `I4`: ad esso viene associato il primo elemento della lista di uscita, `n`, il cui valore viene riprodotto sul record allineato a destra nel campo che va dalla sesta alla nona posizione.
- Il terzo descrittore è ancora `5X`; esso provoca l'inserimento di caratteri blank nelle posizioni dalla decima alla quattordicesima.
- Il quarto descrittore, `E13.6`, è ripetibile e viene associato al secondo elemento della lista, `x`, il cui valore viene opportunamente riprodotto sul record su un campo di ampiezza tredici a partire dalla quindicesima posizione.
- Si incontra infine la parentesi finale e siccome i valori di `m` e di `y` devono ancora essere riprodotti, la scansione della specificazione di formato riprende dall'inizio; viene così creato un secondo record che ha la stessa struttura del primo, sul quale vengono rappresentati i valori di `m` ed `y`.

4.5.2 I descrittori '$h_1 \ldots h_n$' e $h_1 \ldots h_n$

Durante un'operazione di uscita è possibile inserire sul record corrente una stringa $h_1 \ldots h_n$ costituita da caratteri codificabili dal sistema mediante uno dei descrittori:

'$h_1 \ldots h_n$'
$h_1 \ldots h_n$

entrambi aventi la forma di una *costante carattere*. Ad esempio le istruzioni:

```
WRITE(*,100) raggio, area 100
FORMAT(1X,"raggio:",1X,F5.2,3X,"area:",1X,F7.3)
```

produrrebbero, nel caso in cui `raggio` e `area` valessero rispettivamente `2` e `12.56`, il seguente output:

```
raggio:  2.00 area:  12.560
```

4.5.3 Il descrittore :

Quando, nella scansione di una specificazione di formato, si incontra il descrittore ":" l'operazione di I/O ha termine se è stato trasmesso il valore di tutti gli elementi della lista. Questo descrittore, che può *non* essere seguito e preceduto da virgole, è utile essenzialmente per evitare che messaggi "indesiderati" vengano riprodotti sui record di uscita. In altre parole, quando nell'interpretazione di una istruzione `FORMAT` si arriva ad un simbolo di due punti si valuta se ci sono altre variabili nella lista da stampare; se queste ci sono allora viene valutato anche il resto della frase `FORMAT`, in caso contrario l'istruzione ha termine. Il descrittore `:` è utile, ad esempio, per riutilizzare la stessa frase `FORMAT` con più istruzioni `WRITE` (o `PRINT`) aventi liste di output con differenti numeri di variabili. Un utile esempio di utilizzo di questo descrittore è il seguente:

```
WRITE(*,100) m,n,x WRITE(*,100) m,n 100 FORMAT(1X,"m =",I3,2X,"n
=",I3 : 2X,"x =",F7.3)
```

che produce il seguente output:

```
m =  1  n =  2  x =  1.000 m =  1  n =  2
```

4.5.4 Il descrittore /

Il descrittore di formato / (*slash*) indica la fine del trasferimento dei dati sul record corrente.

In operazioni di input questo descrittore di formato fa sì che venga saltato il buffer di input corrente e che venga acquisito un nuovo buffer dall'unità di input sicché l'acquisizione continuerà dall'inizio del nuovo buffer. Ad esempio, la seguente istruzione `READ` formattata legge i valori delle variabili `i` e `j` dalla prima riga di input, salta due righe e legge i valori delle variabili `k` e `l` dalla terza riga di input:

```
INTEGER :: i, j, k, l READ(*,100) i, j, k, l 100 FORMAT(2I3,//,2I3)
```

In operazioni di output il descrittore / trasmette al dispositivo di uscita il contenuto del buffer corrente di output e apre un nuovo buffer. Grazie a questo descrittore è possibile, ad esempio, visualizzare valori di output di una stessa istruzione `WRITE` in più righe. E' possibile usare più slash consecutivi per saltare più righe; inoltre, si noti che non è necessario fare uso di virgole per separare lo slash dagli altri descrittori. Come esempio si consideri la seguente istruzione `WRITE` formattata:

```
REAL :: massa, pressione, temperatura ... WRITE(*,100) massa,
pressione, temperatura 100 FORMAT("Stato del
gas:",20("="),//,"massa:",F7.3,/, &
           "pressione",F7.3,/,"temperatura",F7.3)
```

Se, ad esempio, i valori delle variabili `massa`, `pressione` e `temperatura` fossero, rispettivamente, `0.01`, `2.01` e `298.3` l'output del precedente frammento di programma sarebbe:

```
Stato del gas:
====================

massa:   0.010 pressione:   2.010 temperatura: 298.300
```

Un discorso del tutto analogo vale se le operazioni di I/O avvengono su file esterni. Ogni qualvolta un file viene connesso per operazioni di input usando l'accesso sequenziale, la presenza di questo descrittore nella frase di specificazione di formato posiziona il file all'inizio del successivo record da leggere. Analogamente, quando un file è connesso per operazioni di output usando l'accesso sequenziale, la presenza di uno slash nella frase di specificazione di formato posiziona il file all'inizio di un nuovo record da scrivere.

4.5.5 I descrittori Tn, TRn e TLn

I descrittori `Tn`, `TRn`, `TLn` sono chiamati *descrittori di tabulazione* in quanto permettono di specificare, mediante il valore della costante `n` (un intero positivo senza segno) che in essi compare, la posizione sul record corrente del successivo dato da trasmettere in I/O. In particolare, il descrittore `Tn` indica che l'operazione di lettura/scrittura deve continuare a partire dalla *n_ma* colonna del record corrente. Questo descrittore viene, pertanto, utilizzato per raggiungere direttamente una determinata posizione all'interno del buffer. Si noti che il descrittore `Tn` permette di raggiungere una posizione qualsiasi nella riga di output, anche oltre la posizione specificata dall'istruzione `FORMAT`. E', tuttavia, necessario fare attenzione a non sovrapporre accidentalmente i campi dei dati da rappresentare. In fase di lettura il descrittore di formato `Tn` permette, così come anche `nX`, di ignorare, all'interno di un campo di input, quei dati che non si vogliono leggere; esso, inoltre, può permettere di leggere gli stessi dati due volte in formati diversi. Ad esempio il seguente frammento di programma legge due volte i dati dalla prima alla sesta colonna del buffer di input, una volta come intero e una volta come stringa di caratteri:

```
INTEGER :: numero CHARACTER(LEN=6) :: parola READ(*,100) numero,
parola 100 FORMAT(I6,T1,A6)
```

In fase di scrittura, lo specificatore `Tn` consente di regolare la spaziatura e la posizione dei risultati prodotti nel record di uscita. Così, ad esempio, il seguente frammento di programma:

```
INTEGER :: a = 9, b = 99, c = 999 CHARACTER(LEN=5) :: str = "ciao!"
... WRITE(*,"(T8,I1,T5,I2,T1,I3,T25,A5)") a, b, c, str
```

produce la stampa del record:

```
999 99 9               ciao! ......................... 1   5  8
25
```

(per motivi di chiarezza nel record precedente si sono riportati graficamente anche i punti iniziali di ciascun valore prodotto).

Il descrittore `TRn` [risp: `TLn`], invece, fa sì che l'operazione di lettura/scrittura continui `n` colonne *dopo* [risp: *prima di*] quella attuale. In altre parole, detta p la posizione sul record corrente del carattere finale dell'ultimo campo utilizzato, il descrittore `TRn` indica che il campo destinato al prossimo dato inizia dalla posizione $p+1+n$ mentre il descrittore `TRn` indica che il campo destinato al prossimo dato inizia dalla posizione $p+1-n$. Dunque `TRn` e `TLn` specificano di quante posizioni deve essere spostato, a destra e a sinistra rispettivamente, l'inizio del campo successivo rispetto alla posizione $(p+1)$*_ma*. Il descrittore `TRn` è, pertanto, equivalente ad `nX` mentre `TLn` è equivalente a `T1` se risulta $p+1-n \leq 1$. Si deve precisare che se il codice `TL` indica uno spostamento a sinistra oltre il primo carattere del record, si produce comunque lo spostamento del puntatore sul *primo* carattere de record.

A titolo di esempio, si supponga che `i` sia una variabile intera e che `char` sia una variabile stringa di caratteri di lunghezza sette; l'istruzione di lettura:

```
READ(*,250) i,string 250 FORMAT(T5,I4,TL4,A7)
```

viene eseguita al modo seguente. Il descrittore `T5` causa un salto al quinto carattere del record corrente; il descrittore `I4` viene poi associato alla variabile `i` il cui valore viene letto nel campo che va dal quinto all'ottavo carattere del record. Il descrittore `TL4` indica che il campo relativo al dato successivo inizia a partire dalla quinta posizione del record corrente in quanto si ha p=8 e n=4. Il descrittore `A7` viene poi associato alla variabile `string` il cui valore viene letto nel campo che va dal quinto all'undicesimo carattere del record. In questo modo i caratteri presenti sul record nelle posizioni dalla quinta all'ottava colonna vengono letti due volte e vengono interpretati nei due casi in modi completamente diversi. Se, ad esempio, il record letto è composto dalle successioni di caratteri:

```
12342001/02ABCD
```

i valori assegnati alle variabili `i` e `string` sono, rispettivamente, il numero intero `2001` e la stringa di caratteri `2001/02`.

4.6 Alcune considerazioni sulle istruzioni di I/O formattate

4.6.1 Corrispondenza fra la lista di I/O e la lista di `FORMAT`

Come è noto, mentre il programma esamina da sinistra a destra la lista delle variabili di I/O, contemporaneamente scansiona nello stesso verso la corrispondente istruzione `FORMAT` (sia essa esplicita o contestuale). Così, il primo descrittore di formato ripetibile incontrato all'interno dell'istruzione `FORMAT` viene associato al primo valore della lista di I/O, il secondo descrittore al secondo valore e così via. Naturalmente, il *tipo di un descrittore di formato deve corrispondere al tipo di dato di ingresso o di uscita*. Nel seguente esempio, il descrittore `I3` è associato alla variabile intera `Age`, il descrittore `F10.2` alla variabile reale `Salary`, il descrittore `A` alla variabile stringa `State` e, infine, il descrittore `L2` alla variabile logica `InState`:

```
CHARACTER(LEN=8) :: State   = "Michigan" INTEGER          :: Age
= 30 REAL             :: Salary  = 32165.99 LOGICAL          ::
InState = .TRUE. ... WRITE(*,"(I4,F10.2,A,L2)")
Age,Salary,State,InState
```

Nel caso in cui le variabili nella lista di I/O non dovessero corrispondere nel tipo (o nell'ordine) ai corrispondenti descrittori di formato si verificherà un errore di *run-time*. Un caso del genere è rappresentato dal prossimo esempio ottenuto apportando una lieve (ma significativa) modifica al precedente frammento di programma:

```
CHARACTER(LEN=8) :: State   = "Michigan" INTEGER          :: Age
= 30 REAL             :: Salary  = 32165.99 LOGICAL          ::
InState = .TRUE. ... WRITE(*,"(I4,F10.2,A,L2)")  Age, State, Salary,
InState
```

In questo caso, infatti, l'istruzione `WRITE` tenterà di stampare una variabile stringa con un descrittore valido per dati di tipo reale, ed un valore reale con il descrittore `A` valido esclusivamente per gli oggetti di tipo carattere. Si badi che errori di questo tipo sono abbastanza

subdoli in quanto tipicamente non vengono rilevati in fase di compilazione ma si manifestano solo durante l'esecuzione del programma, provocando una inevitabile interruzione del run all'atto dell'esecuzione dell'istruzione "incriminata".

Se un descrittore di formato è associato a un fattore di ripetizione `r`, il descrittore sarà utilizzato il numero di volte specificato dal fattore di ripetizione, prima che venga utilizzato il successivo descrittore. Nel seguente esempio il descrittore `I3` è associato alle variabili `i`, `j` e `k` mentre il descrittore `F7.3` è associato alla sola variabile `x`:

```
INTEGER :: i,j,k REAL    :: x ... WRITE(*,"(1X,3(I4),F7.3)")
i,j,k,x
```

Se un gruppo di descrittori di formato racchiusi fra parentesi è associato a un fattore di ripetizione `r`, l'intero gruppo sarà utilizzato il numero di volte specificato dal fattore di ripetizione, prima che venga utilizzato il successivo descrittore. Ogni descrittore all'interno del gruppo sarà utilizzato ordinatamente da sinistra a destra durante ogni iterazione. Nel seguente esempio, il descrittore `F7.3` viene associato alla variabile `x`; poi viene utilizzato due volte il gruppo di descrittori `(I3,L2)` e, quindi, `I3` viene associato alla variabile `i`, `L2` alla variabile `y`, di nuovo `I3` alla variabile `j` e `L2` alla variabile `cond2`. Infine `A10` viene associato alla variabile `str`.

```
INTEGER           :: i, j REAL              :: x LOGICAL
:: cond1, cond2 CHARACTER(LEN=10) :: str ...
WRITE(*,"(1X,F7.3,2(I3,L2),A10)")  x,i,cond1,j,cond2,str
```

4.6.2 Lista di I/O più corta della lista di descrittori di formato

Se il numero di variabili della lista di I/O è minore del numero di descrittori di formato, l'applicazione del formato termina al primo descrittore che non può essere associato ad una variabile. In altre parole, una volta che tutte le variabili della lista di I/O sono state processate, l'istruzione di I/O si conclude e tutti i descrittori di formato inutilizzati vengono semplicemente ignorati. Così, ad esempio, la seguente istruzione di scrittura:

```
INTEGER           :: a = 1, b = 2, c = 4 CHARACTER(LEN=20) :: FMT =
"(2I5,I3,F7.3,A10)" ... WRITE(*,FMT) a, b, c
```

si caratterizza per la presenza di tre variabili di uscita e di cinque descrittori di formato. Ebbene, quando la lista di variabili di uscita è stata processata, il contenuto del buffer viene inviato al dispositivo di uscita e gli ultimi tre descrittori vengono del tutto ignorati. La scansione della lista di `FORMAT` si arresterà, pertanto, al descrittore `F7.3` che è il primo descrittore non associato ad una variabile.

4.6.3 Lista di I/O più lunga della lista di descrittori di formato

Se il numero di variabili della lista di I/O è maggiore del numero di descrittori di formato, il programma trasmette [risp: acquisisce] il contenuto corrente del buffer di output [risp: di input]

al dispositivo di uscita [risp: di ingresso] e "riutilizzerà" la lista degli specificatori di formato a partire dalla parentesi aperta più a destra nell'istruzione `FORMAT` incluso, eventualmente, il suo fattore di ripetizione. Per esempio, data la seguente istruzione di uscita:

```
INTEGER           :: i=1, j=2, k=4, l=8 REAL              :: x=1.1,
y=2.2, z=3.3, s=4.4 LOGICAL           :: cond1,cond2
CHARACTER(LEN=20) :: FMT = "(1X,2(I3,F5.2),L1)" ... WRITE(*,FMT)
i,x,j,y,cond1,k,z,l,s,cond2
```

il suo output sarebbe:

```
  1  1.10  2  2.20 T
 4  3.30  8  4.40 F
```

in quanto, una volta terminata la lista dei descrittori, dal momento che la lista di uscita è stata processata soltanto in parte, il programma deve riutilizzare la lista di `FORMAT` a partire dal punto di riscansione che, come sempre, coincide con la parentesi aperta che si trova più a destra nella lista di formato. Pertanto, una volta che il buffer contenete i valori dell variabili della lista parziale:

```
i,x,j,y,cond1
```

viene inviato al dispositivo di uscita per essere rappresentato secondo il formato specificato dalla lista:

```
(1X,I3,F5.2,I3,F5.2,L1)
```

il gruppo di descrittori `2(I3,F5.2),L1` dovrà essere riutilizzato per il processamento della restante parte della lista di output:

```
k,z,l,s,cond2
```

Pertanto la lista di descrittori di formato:

```
(1X,2(I3,F5.2),L1)
```

è, nel caso in esame, del tutto equivalente a:

```
(1X,I3,F5.2,I3,F5.2,A /
    I3,F5.2,I3,F5.2,A)
```

mentre, in generale, equivale a:

```
(1X,I3,F5.2,I3,F5.2,A /
    I3,F5.2,I3,F5.2,A /
    I3,F5.2,I3,F5.2,A /
    ...)
```

Si noti che la presenza dello specificatore non ripetibile *slash*, "/", è giustificata dal fatto che il Fortran impone l'avanzamento di un record nel file di ingresso o di uscita ogni qualvolta viene raggiunta la fine della lista di `FORMAT`.

Come si è avuto modo di notare, la presenza delle parentesi che raggruppano alcuni specificatori condiziona fortemente la posizione del punto di riscansione e, pertanto, può cambiare radicalmente l'interpretazione dell'istruzione `FORMAT`. Ad esempio, si consideri il seguente formato:

```
(1X,2(I5),A)
```

Poiché la riscansione comincerebbe a partire dal fattore di ripetizione `2` questa specificazione di formato è perfettamente equivalente a:

```
(1X,I5,I5,A /
    I5,I5,A /
    I5,I5,A /
    ...)
```

Rimuovendo le parentesi intorno al descrittore `2I5` si otterrebbe la riga di formato:

```
(1X,2I5,A)
```

che, invece, è equivalente a:

```
(1X,I5,I5,A /
 1X,I5,I5,A /
 1X,I5,I5,A /
 ...)
```

Pertanto l'effetto del formato `(1X,2I5,A)` è totalmente differente dall'effetto di `(1X,2(I5),A)`. Allo stesso modo il significato di questa specificazione di formato:

```
(1X,3(I2,I3),F5.0,2(1X,I3))
```

è il seguente:

```
(1X,I2,I3,I2,I3,F5.0,1X,I3,1X,I3 /
                     1X,I3,1X,I3 /
                     1X,I3,1X,I3 /
                     ...)
```

in quanto il punto di riscansione è ancora una volta il fattore di ripetizione `2` ed, inoltre, soltanto il gruppo `2(1X,I3)` viene ripetuto durante il processo di riscansione della frase `FORMAT`.

Naturalmente la posizione del punto di riscansione prescinde dalla presenza o meno di un fattore di ripetizione. Così, ad esempio, la specificazione di formato:

```
(1X,I1,(I2,I3))
```

ha come punto di riscansione il descrittore `I2` e, pertanto, è equivalente alla seguente specificazione:

```
(1X,I1,I2,I3 /
       I2,I3 /
       I2,I3 /
       ...)
```

mentre la stessa specificazione, in assenza di parentesi:

```
(1X,I1,I2,I3)
```

sarebbe equivalente a:

```
(1X,I1,I2,I3 /
 1X,I1,I2,I3 /
 ...)
```

e avrebbe, pertanto, un significato totalmente diverso.

4.6.4 Sovrapposizione fra campi di input ed interpretazione dei *blank*

E' molto importante, in presenza di istruzioni di input formattate, rispettare il vincolo dell'ampiezza `w` dei vari campi di ingresso nella preparazione del record di input. Nel caso, infatti, in cui esistesse un *overlap* i campi della lista ingresso e i valori del record di input, l'operazione di lettura potrebbe dar luogo a risultati davvero inaspettati (se non a condizioni di errore allorquando si perdesse la corrispondenza fra specificatore di formato e tipo della variabile). Ad esempio, il seguente record di input:

```
1234   56  78  90
```

verrebbe interpretato in maniera completamente diversa dalle istruzioni:

```
READ(*,100) a,b,c,d READ(*,200) a,b,c,d 100 FORMAT(I6,I5,I4,I3) 200
FORMAT(I3,I4,I5,I6)
```

producendo, nel primo caso, le assegnazioni:

```
a=1234; b=56; c=78; d=90
```

e nel secondo caso le assegnazioni:

```
a=123; b=4; c=567; d=890
```

Si noti che, generalmente, *tutti* gli eventuali spazi bianchi presenti nel campo di ampiezza `w` vengono ignorati, per cui essi vengono del tutto rimossi dalla stringa di cifre introdotta. Pertanto, data la seguente istruzione di ingresso:

```
INTEGER :: a, b, c, d ... READ(*,"(4I7)")  a, b, c, d
```

ed il seguente record di input:

```
1  3  5  1 35    135 13    5 <--7--><--7--><--7--><--7-->
```

l'effetto che si ottiene equivale ad assegnare il valore `135` a tutte e quattro le variabili della lista di ingresso.

4.6.5 Formato con ampiezza di campo minima

Al fine di minimizzare il numero di spazi bianchi nei record di uscita, il Fortran 95 consente di specificare un formato con una *ampiezza del campo nulla*. Questa particolarità si applica esclusivamente ai descrittori `I`, `F`, `O`, `B`, `Z`. Dunque, è possibile avere una specificazione di formato del tipo:

```
(I0,F0.3)
```

Questo, si badi, *non* significa imporre una ampiezza nulla per il campo destinato a contenere il valore intero o reale, bensì serve ad imporre che il campo abbia una ampiezza "minima" atta a contenere il valore in questione. L'uso di questa facility permette, così, al programmatore di non preoccuparsi eccessivamente della "topologia" del record che, altrimenti, gli imporrebbe spesso di lavorare con campi di dimensioni eccessive per non correre il rischio, in particolari situazioni, di vederlo riempito di asterischi.

4.7 Istruzioni di I/O guidate da lista

Le *istruzioni di* I/O *guidate da lista* costituiscono una forma semplificata delle istruzioni di lettura e scrittura formattate. Esse consentono di adottare un formato "libero" per i dati di ingresso ed un formato "standard" predefinito per i dati di uscita. Così, ad esempio, l'utente può scegliere di introdurre i dati di input in una colonna qualsiasi e l'istruzione `READ` sarà interpretata sempre in maniera corretta. Tali modalità sono utili per un molti casi semplici di interesse pratico e pertanto, anche in relazione alle notevoli semplificazioni così introdotte, le istruzioni in esame vengono largamente utilizzate.

Le operazioni di I/O guidate da lista sono definite unicamente sulle *unità standard precollegate* ed hanno la seguente forma semplificata:

```
READ(*,*) lista_di_ingresso
WRITE(*,*) lista_di_uscita
PRINT*, lista_di_uscita
```

Una istruzione `READ` produce la lettura di una o più linee fino a produrre l'ingresso di tanti valori quante sono le variabili della *`lista_di_ingresso`*. Un'istruzione `WRITE` (o `PRINT`) produce l'uscita di tanti valori quanti sono elencati nella *`lista_di_uscita`*, ciascuno con un formato standard predefinito dallo specifico compilatore e dipendente dal tipo di valore da rappresentare.

Si noti che ciascuna operazione di I/O produce la lettura o la scrittura di un nuovo record indipendentemente dal fatto che nell'operazione precedente l'ultimo record sia stato o meno scandito interamente. In altri termini, non è possibile leggere o scrivere parti successive di uno stesso record con operazioni di I/O distinte. In particolare, se la lista di input è vuota viene saltato il record successivo, se la lista di output è vuota viene prodotta una riga bianca.

4.7.1 Istruzione di input guidate da lista

La lista di ingresso può essere registrata su una o più linee, utilizzando quale carattere di separazione la virgola oppure uno o più spazi bianchi, secondo il seguente formalismo:

- I valori interi vanno denotati a mezzo di una stringa di cifre, gli eventuali spazi bianchi iniziali o finali vengono ignorati.
- I valori di tipo reale possono essere introdotti indifferentemente nella notazione fissa o scientifica.
- I valori di tipo complesso devono essere introdotti nella forma:

  (*parte_reale*,*parte_immaginaria*)

  dove *parte_reale* e *parte_immaginaria* sono entrambi valori reali.
- I valori di tipo stringa possono essere rappresentati o meno racchiusi da una coppia di apici o doppi apici.
- I valori logici devono essere rappresentati con una delle lettere `T` o `F` eventualmente precedute da un punto e seguite da uno o più caratteri.
- I valori di un oggetto di un tipo di dati derivato devono essere introdotti inserendo il valore di ciascun componente dell'oggetto, nello stesso ordine in cui esso figura nella definizione del tipo.

E', inoltre, ammessa una notazione sintetica per valori identici ripetuti, nella forma:

*n***valore*

in cui *n* è una costante intera positiva senza segno. Ad esempio, data la seguente istruzione di lettura:

```
INTEGER :: n REAL ::  x, y, z LOGICAL :: vero ... READ(*,*) n, x, y,
z, vero
```

il seguente record di input:

```
10  3*3.14  .T
```

produce le seguenti assegnazioni di valore:

```
n=10; x=3.14; y=3.14; z=3.14; vero=.TRUE.
```

Le istruzioni di lettura guidate da lista hanno, oltre al vantaggio della semplicità connessa al fatto di non richiedere alcuna istruzione `FORMAT` e alla libertà offerta all'utente di poter introdurre i dati di ingresso a partire da una qualsiasi colonna, la caratteristica di supportare i cosiddetti *valori nulli*. Se un record di input contiene due virgole consecutive allora la corrispondente variabile nella lista di input *non* sarà soggetta ad assegnazione. Questa caratteristica consente all'utente di lasciare inalterati i valori di una o più variabili. Si consideri il seguente esempio:

```
INTEGER :: i=1, j=2, k=3 WRITE(*,*) "Introduci i valori di i, j e k:
" READ(*,*) i,j,k WRITE(*,*) "i, j, k = ", i,j,k
```

Se, quando la precedente istruzione di lettura viene eseguita, il record di ingresso fosse:

```
1000 ,, 3000
```

la successiva istruzione di scrittura produrrebbe il record:

```
i, j, k = 1000 2 3000
```

Allo stesso modo è possibile lasciare inalterati i valori di un intero record o di una sua parte. A tale scopo si usa il separatore *slash* (`/`). In presenza di questo separatore nella lista di ingresso indica che agli elementi della lista che non sono ancora stati messi in corrispondenza con un valore, corrisponde un *valore nullo*. In altre parole, se gli elementi della lista di input sono n e prima dello slash sono stati letti $n-k$ valori, le k variabili residue restano non modificate. Sempre con riferimento all'esempio precedente, se la lista di input fosse stata:

```
1000 /
```

la successiva istruzione di scrittura avrebbe prodotto il record:

```
i, j, k = 1000 2 3
```

Si noti che, tranne che nelle costanti carattere, gli eventuali spazi bianchi nel record di ingresso sono sempre trattati come separatori e non sono, pertanto, "significativi.

4.7.2 Istruzione di output guidate da lista

La forma dei valori prodotti nelle operazioni nelle operazioni di output con formato diretto da lista è analoga a quella per i dati di ingresso tranne che per le eccezioni elencate di seguito. I valori vengono visualizzati separati da virgole o da spazi bianchi come fissato dal processore. In particolare si hanno le seguenti forme di rappresentazione:

- I valori interi prodotti in output sono quelli che sarebbero prodotti usando il descrittore `Iw` con `w` fissato dal processore.
- I valori reali prodotti in uscita sono quelli che verrebbero prodotti usando descrittori `F` o `E`, dove il formato scelto di volta in volta dipende dal valore numerico da rappresentare.
- I valori di tipo complesso vengono prodotti nella forma:

  (*parte_reale*,*parte_immaginaria*)

  dove *parte_reale* e *parte_immaginaria* sono entrambi valori reali prodotti secondo quanto visto al punto precedente.
- I valori di tipo stringa vengono prodotti secondo il formato `A` e, quindi, *non* precedute o seguite da separatori (con l'effetto che due stringhe consecutive possono risultare unite).
- I valori logici prodotti in output sono `T` e `F` (il formato corrispondente è, dunque, `L1`).

Si noti, infine, che se devono essere prodotte in uscita $n \geq 2$ costanti identiche il processore può scegliere di stamparle nella forma:

*n*valore*

dove *valore* è il valore delle costanti identiche.

4.8 Caratteri di controllo per la stampante

Fino ad ora si è appreso come controllare il layout di dati da o verso i dispositivi standard di I/O. Quando, tuttavia, durante operazioni di output i dati vengono inviati ad un dispositivo che il sistema identifica come *stampante*, è possibile esercitare un ulteriore (limitato) controllo della spaziatura verticale del foglio stampato.

Quando un record viene inviato ad un dispositivo di uscita designato come *stampante*, il primo carattere del record viene rimosso dalla linea ed interpretato come *carattere di controllo* che determina di quanto il foglio deve "salire" prima che i restanti caratteri del record vengano stampati. Al riguardo, esistono quattro caratteri che hanno un particolare significato. Se il primo carattere del record di uscita non è uno di essi, l'effetto sulla stampante risulta indefinito anche se, in pratica, qualsiasi carattere viene interpretato come uno spazio e, pertanto, produce l'avanzamento di un rigo. Tali caratteri sono elencati di seguito:

Carattere di controllo	*Avanzamento verticale*
`b` (*spazio*)	una linea
`0` (*zero*)	due linee
`1` (*uno*)	una pagina
`+` (*più*)	nessun avanzamento

Dal momento che il primo carattere è rimosso e, pertanto, non riprodotto è importante inserire un carattere extra ("di controllo") all'inizio di ogni record da stampare.

Esistono molti modi per inserire un carattere di controllo in testa ad un record, soprattutto se tale carattere è uno spazio bianco. E' preferibile, tuttavia, per motivi di leggibilità, inserire esplicitamente il suddetto carattere di controllo, come mostrano i seguenti esempi:

```
100 FORMAT(1X,...) 200 FORMAT(' ',...) 300 FORMAT(T2,...)
```

E' importante sottolineare che quanto detto si applica esclusivamente alla unità stampante e a tutte quelle unità che il compilatore identifica come stampanti, inclusa, in alcuni casi, l'unità video. Tutte le altre unità di uscita non richiedono alcun carattere di controllo e stamperanno, pertanto, il record completo del suo carattere di testa. Si noti, infine, che nel caso l'unità stampante sia l'unità standard di uscita, l'istruzione `PRINT` inserisce *automaticamente* il carattere di controllo *blank* all'inizio di ogni record.

Si consideri, ancora, il seguente esempio:

```
WRITE(*,100) 100 FORMAT('1',"Questo e' il titolo e viene stampato a
inizio pagina") WRITE(*,200) 200 FORMAT('0',"Ora si e' raddoppiata
```

```
l'interlinea") WRITE(*,300) 300 FORMAT(' ',"Adesso si stampa
'normalmente'")
```

L'effetto di questo segmento di programma, compilato utilizzando il compilatore Microsoft Power Station 4.0, è il seguente:

```
Questo e' il titolo e viene stampato a inizio pagina

Ora si e' raddoppiata l'interlinea Adesso si stampa 'normalmente'
```

Si faccia molta attenzione quando si lavora con istruzioni di output formattate, potendosi presentare problemi davvero subdoli risultanti in record di uscita quantomai "bizzarri". Ad esempio, anche le seguenti semplici istruzioni possono generare risultati imprevedibili:

```
WRITE(*,100) n 100 FORMAT(I3)
```

In questo caso, il descrittore `I3` specifica l'intenzione di stampare il valore della variabile intera `n` nei primi tre caratteri del buffer di output. Se il valore di `n` fosse, ad esempio, `25` le tre posizioni verrebbero riempite dalla stringa *b*`25` dove, al solito, il simbolo *b* indica uno spazio vuoto. Dal momento che il primo carattere viene interpretato come carattere di controllo, la stampante salta alla riga successiva e stampa il numero `25` nelle *prime due* colonne della nuova riga. Se, invece, `n` valesse `125`, i primi tre caratteri del buffer di input sarebbero riempiti con la stringa `125` e dal momento che il primo carattere del buffer (nella fattispecie il carattere `1`) è interpretato come carattere di controllo, la stampante salta alla nuova pagina e stampa il numero `25` nelle prime due colonne della nuova riga. Questo, ovviamente, non era il risultato atteso. Il problema, ovviamente, si risolve inserendo esplicitamente un carattere di controllo nella prima colonna:

```
WRITE(*,100) n 100 FORMAT(' ',I3)
```

Si fa notare, tuttavia, che i problemi di cui sopra si ravvisano soltanto con quei compilatori (come il citato Fortran Power Station) che considerano il terminale video alla stessa stregua di una stampante a linee. Viceversa, compilatori che non fanno questa assunzione, come il Digital Visual Fortran 6.0, sono immuni da questo problema, per cui compilando con quest'ultimo un programma di test con il seguente segmento di codice:

```
n=125 WRITE(*,100) n 100 FORMAT(I3)
```

il risultato sarebbe quello corretto:

```
125
```

4.9 Istruzioni di I/O con meccanismo *namelist*

Le istruzioni di I/O con meccanismo *namelist* rappresentano un metodo abbastanza "curioso" di lettura o scrittura di dati da o verso file esterni o dispositivi standard. Specificando una o più variabili come facenti parte di un gruppo *namelist* è possibile leggere o scrivere i valori di ciascuna di esse attraverso un'*unica* istruzione di I/O.

Un gruppo *namelist* può essere creato a mezzo di una istruzione `NAMELIST` la quale si presenta con la seguente sintassi:

`NAMELIST/`*`nome_della_lista`*`/`*`lista_di_variabili`*

in cui *`nome_della_lista`* è il nome che identifica l'intero gruppo, mentre *`lista_di_variabili`* è un elenco di nomi di variabili di tipi predefiniti, definiti dall'utente o di variabili di tipo array. Non possono, tuttavia, far parte di un gruppo *namelist* variabili array o stringhe di caratteri che siano parametri formali con dimensioni non costanti, array automatici o allocabili, puntatori. L'ordine con il quale le variabili sono elencate nella lista determina l'ordine con cui le stesse verranno riprodotte in una eventuale istruzione di uscita. Il nome di una stessa variabile può tranquillamente comparire in più di un gruppo `NAMELIST`. Infine, sebbene ciò risulterà chiaro solo dopo la lettura del capitolo dedicato alle *unità di programma*, si anticipa che se il gruppo `NAMELIST` è specificato con l'attributo `PUBLIC`, nessun componente della lista può avere l'attributo `PRIVATE`.

Un esempio di utilizzo dell'istruzione `NAMELIST` è il seguente:

```
REAL :: x, val INTEGER, DIMENSION(10) ::  indici
NAMELIST/mia_lista/x,val,indici
```

E' anche possibile dichiarare diversi gruppi *namelist* con la stessa istruzione:

`NAMELIST/`*`lista_1`*`/`*`variabili_lista_1`* `[[,] /`*`lista_2`*`/`*`variabili_lista_2`*`,...]`

sebbene è sempre raccomandabile scrivere ciascuna istruzione `NAMELIST` su una linea separata. Si noti inoltre che se due istruzioni `NAMELIST` hanno lo stesso nome (ossia se hanno in comune il termine *`nome_lista`*) allora la seconda lista sarà considerata come una continuazione della prima, ad esempio le istruzioni:

```
NAMELIST/mia_lista/x,y,z(3:5) NAMELIST/mia_lista/a,b,c
```

hanno esattamente lo stesso effetto dell'unica istruzione:

```
NAMELIST/mia_lista/x,y,z(3:5),a,b,c
```

Si faccia bene attenzione al fatto che `NAMELIST` rappresenta una istruzione di specificazione e pertanto dovrà essere opportunamente collocata all'interno della parte "dichiarativa" del codice sorgente, prima cioè di ogni istruzione esecutiva della unità di programma in cui figura.

Le operazioni di I/O con meccanismo *namelist* sono realizzate con istruzioni `READ` o `WRITE` che si caratterizzano per i seguenti aspetti:

- la *lista di ingresso* o *di uscita* è assente;
- l'identificatore di formato (`[FMT=]`*`formato`*) è rimpiazzato dal nome della lista secondo la sintassi:

  `[NML=]`*`nome_della_lista`*

Così, ad esempio, l'istruzione:

```
WRITE(*,NML=gruppo)
```

invierà al dispositivo di uscita standard i valori di tutte le variabili facenti parte della lista con nome `gruppo`, nell'ordine in cui, in fase dichiarativa, sono state inserite nella lista. L'output prodotto ha la seguente forma:

- Il primo carattere prodotto è un *ampersand* (`&`), opzionalmente preceduto da un certo numero di caratteri vuoti ed immediatamente seguito dal nome del gruppo `NAMELIST`. Detto nome apparirà scritto sempre in maiuscolo.
- Per ogni variabile del gruppo `NAMELIST` verrà prodotta la coppia *`nome-valore`* separati da un segno di uguaglianza. Il segno = può essere opzionalmente preceduto o seguito da uno o più spazi bianchi. Il formato di uscita dei dati è identico a quello che si avrebbe nel caso di output guidato da lista. Se una delle variabili è un array, il segno di uguaglianza sarà seguito dalla lista dei valori corrispondenti, ordinatamente, agli elementi dell'array. Se due o più valori consecutivi sono identici, il processore può scegliere di rappresentarli nella forma sintetica *`r*valore`* in cui *`valore`* è il valore da rappresentare mentre *`r`* rappresenta il numero di ripetizioni. I valori della lista possono essere, opzionalmente, seguiti da una virgola.
- L'ultimo carattere prodotto è uno slash (`/`).

Nel seguente esempio vengono dichiarate un certo numero di variabili e inserite all'interno di una gruppo `NAMELIST`, successivamente vengono inizializzate e infine riprodotte a video (o comunque inviate al dispositivo standard di uscita):

```
INTEGER :: int1, int2, int3 INTEGER, DIMENSION(3) :: array LOGICAL
:: log REAL(KIND=single) :: real1 REAL(KIND=double) :: real2 COMPLEX
:: z1, z2 CHARACTER(LEN=1) :: char1 CHARACTER(LEN=10) :: char2

NAMELIST /esempio/ int1,int2,int3,log1,real1,real2  &
                 & z1,z2,char1,char2,array
int1  = 11 int2  = 12 int3  = 14 log1  = .TRUE. real1 = 24.0 real2 =
28.0d0 z1    = (38.0,0.0) z2    = 2.0*z1 char1 = 'A' char2 =
'0123456789' array = (/41,42,43/) WRITE (*,esempio)
```

Un possibile output potrebbe essere il seguente:

```
 &ESEMPIO INT1=11,INT2=12,INT3=14,LOG=T,REAL1=24.0000,REAL2=
28.0000000000000,Z1=(38.0000,0.000000),Z2=(76.0000,0.000000),CHAR1=A,
CHAR2=0123456789,ARRAY=41,42,43 /
```

come anche possibile è la seguente forma di uscita:

```
&ESEMPIO INT1 =           11 INT2 =           12 INT3 =           14
LOG =  T REAL1 =        24.000000 REAL2 =        28.000000000000000
Z1 =           (38.000000,0.000000E+00) Z2 =
(76.000000,0.000000E+00) CHAR1 =  A CHAR2 =  0123456789 ARRAY =
41          42          43 /
```

la scelta dipende dalla specifica implementazione.

Dall'esempio precedente si può osservare come, in fase di output, normalmente le stringhe vengano riprodotte senza delimitatori. Si può, tuttavia, forzare, l'uso di delimitatori utilizzando lo specificatore `DELIM` nell'istruzione di connessione del file di lavoro. Ciò verrà spiegato in dettaglio nel capitolo 8 quando si parlerà proprio della gestione dei file, per ora si anticipa soltanto che è possibile connettere anche un file preconnesso allo scopo di modificarne alcune impostazioni. Ad esempio, supponendo che al terminale video, inteso come dispositivo standard di uscita, sia assegnato dal compilatore l'unità logica `6`, l'istruzione:

```
OPEN(UNIT=6,specificatore=valore)
```

consente di impostare a *`valore`* il valore di *`specificatore`*. Per quanto concerne, in particolare, lo specificatore `DELIM`, esso indica quali caratteri devono (eventualmente) delimitare una costante di tipo `CHARACTER` nelle operazioni di uscita dirette da lista o con meccanismo *namelist*. I suoi possibili valori sono:

- `'NONE'`: è il valore di default ed indica che nessun delimitatore "contornerà" le stringhe in uscita;
- `'APOSTROPHE'`: indica che tutte le costanti stringa prodotte in output saranno delimitate da apici ("apostrofi") per cui gli eventuali apostrofi presenti nella stringa verranno raddoppiati;
- `'QUOTE'`: indica che tutte le costanti stringa prodotte in output saranno delimitate da doppi apici (*quotation marks*) per cui gli eventuali doppi apici presenti nella stringa verranno raddoppiati.

Si osservi che lo specificatore `DELIM` ha effetto unicamente nelle operazioni di output mentre viene semplicemente ignorato in fase di input. Ponendo, pertanto, il valore di `DELIM` pari ad `'APOSTROPHE'` o a `'QUOTE'`, in uscita verranno prodotte stringhe delimitate, rispettivamente, da apici singoli o doppi. Così, ad esempio, se l'istruzione di lettura dell'esempio precedente fosse stata:

```
OPEN(UNIT=6,DELIM='QUOTE') WRITE(*,NML=esempio)
```

le variabili `char1` e `char2` sarebbero state riprodotte come:

```
CHAR1 =  "A" CHAR2 =  "0123456789"
```

Le operazioni di ingresso con `NAMELIST` avvengono in maniera molto simile a quanto visto per il meccanismo di uscita. La differenza sostanziale rispetto al caso precedente, tuttavia, è rappresentata dal fatto che l'utente non ha l'obbligo di inserire i valori di tutte le variabili della lista, in tal caso i valori delle variabili non introdotte resterà inalterato. Dunque, nelle operazioni di lettura vengono definiti soltanto gli oggetti specificati nel file di input mentre tutti gli eventuali altri elementi della lista restano nel loro preesistente stato di definizione (o, eventualmente, restano indefiniti). Allo stesso modo, è possibile definire il valore di un elemento di array o di un suo sottoinsieme senza alterare il valore delle altre porzioni dell'array. E' anche possibile inserire due o più volte il valore di una stessa variabile, con il risultato che il valore effettivamente attribuito alla variabile sarà l'ultimo introdotto. Nel caso di input di valori complessi o di tipo definito dall'utente è importante che il numero delle componenti introdotte non superi quello previsto per il tipo in esame (sebbene esso possa essere minore con il solito risultato che il valore delle componenti non introdotte resta inalterato).

Per quanto concerne il formato dei dati da introdurre in input e le regole di scrittura valgono, fatta salva qualche eccezione, le stesse considerazioni svolte a proposito dell'uso di `NAMELIST` nelle operazioni di uscita. Si ha, pertanto, quanto segue:

- Il primo carattere da introdurre deve essere un ampersand (`&`) immediatamente seguito dal nome della lista (senza blank interposti). Detto nome può essere scritto indifferentemente in caratteri maiuscoli o minuscoli.
- Per ogni variabile della lista deve essere riprodotta la coppia *`nome-valore`*. Non ha, tuttavia, importanza l'ordine con cui si inseriscono le coppie *`nome-valore`* né ha importanza il modo con cui le coppie vengono separate, potendo scegliere indifferentemente come separatori virgole o spazi bianchi, così come è possibile anche disporle su righe differenti.
- L'ultimo carattere da introdurre deve essere il carattere slash (`/`).

Si noti che, contrariamente a quanto avviene con le istruzioni di uscita, in fase di lettura gli oggetti stringa di caratteri *devono* essere delimitati da apici (indifferentemente singoli o doppi).

Un esempio di input con `NAMELIST` può essere il seguente:

```
INTEGER :: a, b CHARACTER(LEN=10) str NAMELIST /mynml/ a, b, str
 ...
READ(*,NML=mynml)
```

e tutte le seguenti possibilità rappresentano valide alternative per l'inserimento dei dati:

```
&mynml a=1, b=3, str="tutto ok"\

&MYNML a=1 b=3 str="tutto ok" \

&mynml str="tutto ok" b=3 a=1\
```

Si faccia però attenzione a non inserire nel file di input il nome di una variabile *non* appartenente al gruppo `NAMELIST`, in quanto ciò provocherebbe un errore al tempo di esecuzione.

In una lista di input un valore può essere ripetuto utilizzando un opportuno fattore di ripetizione (una costante intera senza segno seguita da un asterisco) inserito prima del valore da ripetere. Ad esempio, la frase:

```
7*"Hello"
```

ha come effetto quello di assegnare la stringa `Hello` a sette elementi consecutivi di tipo `CHARACTER` (potrebbero essere sette elementi consecutivi di un array o una lista di sette stringhe). Un fattore di ripetizione non seguito da alcun valore indica un *valore nullo* multiplo ed il valore delle corrispondenti variabili non sarà, pertanto, alterato. Al fine di chiarire quest'ultimo concetto, si consideri il seguente esempio:

```
INTEGER, DIMENSION(102) :: vettore vettore = 10, 50*25, 50*, -101
vettore(42) = 63
```

Questo frammento di programma ha come effetto quello di assegnare il valore `10` al primo elemento dell'array `vettore`, assegnare il valore `25` agli elementi dal secondo al cinquantunesimo, lasciare inalterati i valori dal cinquantaduesimo fino al penultimo, assegnare il valore `-101` all'ultimo elemento dell'array. Infine il valore del quarantaduesimo elemento di `vettore` viene aggiornato a `63`.

Un'altra possibilità per realizzare una assegnazione di valore nullo ad un elemento di array è quella di inserire, nella lista dei valori da assegnare ai componenti dell'array, due virgole consecutive in corrispondenza del componente da lasciare inalterato. Ad esempio, data la seguente definizione:

```
INTEGER, DIMENSION(5) :: array=0 NAMELIST/mialista/array,... ...
READ(*,NML=mialista)
```

se il file di input fosse:

```
&mialista array = 1,2,,4,5 ...
```

il valore di `array` sarebbe:

```
(1,2,0,4,5)
```

4.10 *Non-advancing* I/O

Le operazioni `READ` e `WRITE` su file ad accesso sequenziale sono per default di tipo *advancing*. Questo significa che il file è automaticamente posizionato all'inizio del record successivo a quello appena letto o scritto prima che il successivo trasferimento di dati abbia luogo, oppure alla fine del record quando l'operazione di I/O è stata completata.

Esistono, tuttavia, situazioni in cui è conveniente leggere o scrivere solo parte di un record, rimandando la lettura o la scrittura della restante parte. A tale scopo il Fortran 90/95 mette a

disposizione una forma speciale di I/O chiamata, appunto, *non-advancing*. Questa particolare forma di ingresso/uscita può avere luogo soltanto con file *esterni* connessi ad accesso *sequenziale* (inclusi i dispositivi standard di I/O) di tipo formattato e con formato esplicito (quindi non può essere usato nelle operazioni di I/O guidate da lista o con meccanismo *namelist*).

Le operazioni di I/O possono essere rese *non-advancing* includendo lo specificatore `ADVANCE` nelle istruzioni `READ` o `WRITE`. Questo specificatore ha la seguente forma:

`ADVANCE=`*`modalità_di_avanzamento`*

in cui *`modalità_di_avanzamento`* è una espressione carattere che può valere `'YES'` (che è il valore di default) oppure `'NO'`.

Una operazione di I/O di tipo *advancing* può essere scelta esplicitamente inserendo lo specificatore `ADVANCE='YES'` nella frase `READ` o `WRITE`, sebbene il risultato è lo stesso anche se si omette lo specificatore. Una operazione di I/O *non-advancing*, invece, ha luogo inserendo lo specificatore `ADVANCE='NO'` nella frase `READ` o `WRITE`.

Per quanto concerne le operazioni di input *non-advancing*, è bene precisare quanto segue:

- Una operazione di lettura continua sul record corrente fino a che non viene incontrata una condizione di *fine record*.
- Se una istruzione `READ` *non-advancing* tenta di leggere dati oltre il *marker di fine record* viene generato un errore di *run-time*. In questo caso l'esecuzione del programma si arresta, a meno che l'istruzione di lettura non sia stata descritta con lo specificatore `IOSTAT`, nel qual caso l'esecuzione del programma *non* viene interrotta e il file viene posizionato immediatamente dopo il record appena letto.

Come facilmente intuibile, l'uso dello specificatore `IOSTAT` può risultare particolarmente utile per controllare una condizione di errore. La sua sintassi è:

`IOSTAT=`*`stato`*

in cui *`stato`* è una variabile di tipo `INTEGER`. Al termine dell'operazione di lettura alla variabile *`stato`* viene assegnato un valore rappresentativo dell'occorrenza e della tipologia di un eventuale errore. I possibili casi che possono presentarsi sono:

- La variabile *`stato`* vale zero: ciò sta ad indicare che l'operazione di lettura è avvenuta con successo.
- La variabile *`stato`* ha un valore positivo (dipendente dal processore): l'operazione di lettura è stata "abortita".
- La variabile *`stato`* vale `-1`: è stata incontrata una condizione di *fine file*.
- La variabile *`stato`* vale `-2`: è stata incontrata una condizione di *fine record* in una operazione di I/O *non-advancing*.

Un uso particolarmente semplice dello specificatore `IOSTAT` è il seguente:

```
READ(UNIT=*,FMT=100,IOSTAT=ioerror) x,y,z IF(ioerror/=0) THEN
   ... !    Si puo' stampare un messaggio di errore
   ... !    e si possono prendere eventuali precauzioni
   ... !    prima di abortire l'operazione
END IF 100 FORMAT(...) ! Continua la normale esecuzione
```

Si noti che l'utilizzo dello specificatore `IOSTAT` *non* è limitato al controllo delle operazioni di *non-advancing* I/O ma, al contrario, è di applicabilità del tutto generale. Anche per questo motivo si ritornerà sull'argomento nel capitolo 8 dove si continuerà a parlare dei meccanismi di I/O.

Nel caso in cui abbia avuto luogo una condizione di errore può risultare utile conoscere il numero di caratteri che sono stati letti. Ciò può essere ottenuto a mezzo dello specificatore `SIZE` il quale ha la seguente forma:

`SIZE=`*`contatore`*

in cui *`contatore`* è una variabile di tipo `INTEGER`. Al termine dell'operazione di lettura alla variabile *`contatore`* viene assegnato il numero di caratteri letti ad eccezione dagli eventuali caratteri *blank*. Così, ad esempio, dato il seguente frammento di programma:

```
! Esempio di lettura ''non avanzante'' INTEGER recsize,stato
CHARACTER(50) stringa
READ(UNIT=*,FMT='(A15)',ADVANCE='NO',SIZE=recsize,IOSTAT=stato)
stringa IF(stato==-2) THEN
   PRINT*, "Ripeti l'immissione della stringa"
   READ(*,'(A<recsize>)',ADVANCE='NO') stringa
END IF WRITE(*,*) stringa
```

se la lunghezza del record di ingresso fosse minore di cinquanta caratteri, l'istruzione `READ` incontrerebbe una condizione di *fine record*, la variabile `stato` assumerebbe pertanto il valore `-2` mentre il valore della variabile `recsize` risulterebbe pari proprio al numero di caratteri letti ossia all'effettiva lunghezza del record introdotto. Pertanto potrebbe avere luogo una nuova operazione di lettura di un record di esattamente `recsize` caratteri che non genererebbe alcuna condizione di errore. Come detto, la variabile `contatore` dello specificatore `SIZE` ignora gli eventuali caratteri *blank* che contornano il record letto. Esiste, tuttavia, la possibilità di rendere significativi tali caratteri, connettendo, cioè, il file di lavoro (nella fattispecie, il dispositivo standard di lettura) e inserendo lo specificatore `PAD` posto pari a `'NO'`. Questo specificatore ha la forma:

`PAD=`*`modalità_di_padding`*

in cui *`modalità_di_avanzamento`* è una espressione carattere che può valere `'YES'` (il valore di default) oppure `'NO'`. Nel caso esso valga `'YES'` (oppure se è omesso) il processore aggiunge nelle operazioni di *input formattate* dei "bianchi di testa" ai record la cui lunghezza risulti minore di quanto specificato dalla specificazione di formato. Se il valore di `PAD` è `'NO'` la lunghezza del record di input deve essere pari almeno a quanto specificato dal formato. Così, ad esempio, se un programma contiene le seguenti istruzioni:

```
INTEGER :: int1, int2 ... OPEN(UNIT=6,PAD='YES') READ(*,FMT='(2I6)')
int1,int2
```

e il record di ingresso è inserito come:

```
12bbbb34
```

il risultato sarebbe l'assegnazione del valore `12` alla variabile `int1` e del valore `34` alla variabile `int2` (naturalmente lo stesso effetto si avrebbe omettendo lo specificatore `PAD` ed assumendo per esso il valore di default). Se, invece l'istruzione di connessione fosse:

```
OPEN(UNIT=6,PAD='NO')
```

la successiva istruzione di lettura fallirebbe in quanto la lunghezza del record di ingresso, di otto caratteri, risulta minore dei dodici caratteri che è quanto imposto dalla specificazione di formato.

Pertanto, connettere il dispositivo di ingresso standard specificando `PAD='NO'` fa in modo che gli eventuali caratteri bianchi inseriti come *padding* vengano tenuti in conto dallo specificatore `SIZE` in una eventuale istruzione di lettura *non-advancing*. Si fa notare, tuttavia, che l'uso dello specificatore `PAD` risulta poco significativo nelle operazioni di lettura da tastiera mentre può risultare utile nelle operazioni di lettura da file esterno scritti, ad esempio, da un altro programma; in tal caso, infatti, porre uguale a `'NO'` il valore di `PAD` può fornire uno strumento per verificare se la lettura dei dati avviene correttamente.

Per quanto concerne, infine, le operazioni di output *non-advancing*, esse sono molto più semplici rispetto a quelle di input. E' necessario prevedere soltanto due possibilità:

- Se non viene riscontrata alcuna condizione di errore, l'operazione di scritture continua sul medesimo record a partire dall'ultimo carattere inserito.
- Se viene incontrata una condizione di errore, il file viene posizionato immediatamente dopo il record appena scritto.

L'utilizzo più frequente di una istruzione di uscita *non-advancing* è quello di scrivere un *prompt* sul terminale video e di leggere il record di input sulla stessa riga dello schermo. Ad esempio, la seguente coppia di operazioni di I/O:

```
WRITE(*,'("Inserisci il valore di m: ")',ADVANCE='NO')
READ(*,'(I5)') m
```

visualizza sul terminale video la stringa:

```
Inserisci il valore di m:  _
```

in cui il simbolo *underscore* è rappresentativo della posizione del cursore che si posiziona immediatamente dopo l'ultimo carattere della stringa rappresentata e corrisponde al punto dove comincerà l'inserimento del record di input.

Capitolo 5

Unità di Programma

Un qualsiasi programma Fortran può comporsi di più unità distinte dette **unità di programma**. Per unità di programma si intende un **programma principale** (o **main**), una **procedura** (che può essere, indifferentemente, *esterna*, *interna* o *di modulo*) o un **modulo**. In ogni caso, un programma eseguibile consiste *sempre* di un *unico* main e di un numero qualsiasi (anche zero) di altre unità di programma. L'unico legame tra le diverse unità di programma è l'*interfaccia* attraverso la quale un'unità ne invoca un'altra *per nome*.

Lo "spacchettamento" di un programma in più unità indipendenti presenta numerosi vantaggi:

- Le singole unità di programma possono essere scritte e testate indipendentemente l'una dall'altra.
- Un'unità di programma che abbia un *solo compito* ben preciso da svolgere è molto più semplice da comprendere e da mantenere.
- Una volta che un modulo o una procedura che siano state sviluppate e testate, è possibile riutilizzarle in altri programmi dando, così, la possibilità al programmatore di costruirsi una propria libreria di programmi.
- Alcuni compilatori riescono meglio ad ottimizzare un codice quando il sorgente sia stato scritto in forma *modulare*.

5.1 Il programma principale

Tutti i programmi hanno uno ed un solo *programma principale*. L'esecuzione di un qualsiasi programma parte sempre con l'esecuzione della prima istruzione del main e procede da quel punto in poi.

La forma generale dell'unità di programma principale è la seguente:

```
[PROGRAM [nome_programma]]
  [istruzioni di definizione e dichiarazione di variabili]
  [istruzioni eseguibili]
```

```
[CONTAINS
  procedure interne]
END [PROGRAM [nome_programma]]
```

L'istruzione `PROGRAM` segna l'inizio dell'unità di programma principale mentre l'istruzione `END PROGRAM` non solo rappresenta la fine dell'unità ma anche la fine dell'*intero* programma. Specificare il nome del programma è opzionale ma è sempre consigliabile farlo. L'istruzione `CONTAINS` serve ad identificare le eventuali *procedure interne* all'unità di programma principale. Quando tutte le istruzioni eseguibili siano state completate, il controllo bypassa le eventuali procedure interne per eseguire l'istruzione di `END`.

Un programma può essere interrotto, durante la sua esecuzione, in un punto qualsiasi e da ogni unità di programma, attraverso l'istruzione `STOP` la quale ha la seguente sintassi:

```
STOP [etichetta]
```

in cui l'*etichetta* è una stringa di caratteri opzionale (racchiusa tra virgolette) avente lo scopo di informare l'utente del punto in cui l'esecuzione del programma è stata interrotta.

5.2 Procedure

Una *procedura* è un'unità di programma usata per raggruppare un insieme di istruzioni correlate aventi lo scopo di risolvere un problema specifico e, in qualche modo, "completo". Per procedura si intende una **subroutine** o una **function**. Una *function* restituisce un solo valore e, tipicamente, non altera il valore dei suoi argomenti; una *subroutine* di solito serve ad eseguire calcoli più complicati e può restituire, attraverso i suoi argomenti, zero o più risultati.

Le procedure possono essere di due tipi: *intrinseche* o *definite dall'utente*. Le prime sono residenti nel linguaggio e, pertanto, sono immediatamente utilizzabili dal programmatore (tipiche procedure intrinseche sono le *funzioni matematiche* `SIN`, `LOG`, etc.). Invece, le procedure definite dall'utente (anche dette *sottoprogrammi*) sono create dal programmatore per svolgere compiti speciali non previsti dalle procedure intrinseche.

Il Fortran 90/95 prevede tre tipologie di sottoprogrammi:

- **Procedure Interne**: sono contenute interamente in una unità di programma "ospite".
- **Procedure Esterne**: sono procedure "indipendenti", spesso *non* contenute nel file che contiene l'unità di programma chiamante e non necessariamente devono essere scritte in Fortran.
- **Procedure di Modulo**: sono procedure contenute all'interno di un'unità di programma *modulo*.

5.2.1 Subroutine

Una *subroutine* è una procedura Fortran che viene invocata specificandone il nome in un'istruzione `CALL`, riceve i valori di *input* e restituisce quelli di *output* attraverso una `lista_di_argomenti`. La forma generale di una subroutine è la seguente:

```
SUBROUTINE nome_subroutine [(lista_di_argomenti )]
  [istruzioni di definizione e dichiarazione di variabili ]
  [istruzioni eseguibili ]
[CONTAINS
  procedure interne ]
[RETURN]
END [SUBROUTINE [nome_subroutine ]]
```

L'istruzione `SUBROUTINE` identifica l'inizio della subroutine, specifica il nome della procedura e la lista degli argomenti.

La lista degli argomenti contiene una serie di variabili che vengono passate alla subroutine dall'unità di programma chiamante. Queste variabili sono chiamate **parametri formali** o **argomenti fittizi** (*dummy*) in quanto le subroutine non riservano alcuna memoria per questo tipo di variabili. Si tratta, quindi, di una specie di "segnaposti" per gli **argomenti effettivi** o **parametri attuali** che saranno passati dall'unità chiamante.

Allo stesso modo di un'unità di programma principale, anche una subroutine ha una *sezione dichiarativa* ed una *sezione esecutiva*. Quando un programma *chiama* una subroutine, l'esecuzione del programma viene temporaneamente sospesa per consentire l'esecuzione della subroutine. Quando viene raggiunta l'istruzione `RETURN` o l'istruzione `END SUBROUTINE` l'esecuzione dell'unità chiamante riprende dalla riga successiva a quella che ha effettuato la chiamata.

Qualsiasi unità di programma eseguibile può chiamare una subroutine, inclusa un'altra subroutine. La *chiamata* avviene attraverso l'istruzione `CALL`, la cui sintassi è:

```
CALL nome_subroutine ([lista_di_argomenti ])
```

Gli *argomenti effettivi* della lista degli argomenti devono corrispondere in *numero*, *ordine* e *tipo* agli *argomenti fittizi* dichiarati nella subroutine.

5.2.2 Function

Una *function* è una procedura Fortran il cui risultato è un numero, un valore logico, un array, una stringa, un puntatore, che può essere combinato con variabili e costanti a formare una espressione. Queste espressioni possono apparire sul lato destro di un'istruzione di assegnazione nell'unità di programma chiamante.

La forma generale di una funzione è la seguente:

```
tipo FUNCTION nome_function [(lista_di_argomenti )]
  [istruzioni di definizione e dichiarazione di variabili ]
  [istruzioni eseguibili ]
  nome_funzione = espressione
[CONTAINS
  procedure interne ]
END [FUNCTION [nome_function ]]
```

Il *tipo* del risultato fornito dalla function può anche essere specificato nella sezione dichiarativa, in tal caso la forma della function diventa:

```
[prefisso] FUNCTION nome_function[(lista di argomenti)]
  tipo nome_function
  [istruzioni di definizione e dichiarazione di variabili]
  [istruzioni eseguibili]
  nome_function = espressione
[CONTAINS
  procedure interne]
END [FUNCTION [nome_function]]
```

in cui lo specificatore opzionale *prefisso* può essere PURE, ELEMENTAL o RECURSIVE.

La function deve iniziare con l'istruzione FUNCTION e terminare con l'istruzione END FUNCTION. Il *nome* della function deve essere specificato nell'istruzione FUNCTION mentre è facoltativo nell'istruzione END FUNCTION.

Per eseguire una funzione basta specificarne il nome all'interno di un'espressione dell'unità chiamante. L'esecuzione della funzione termina quando viene raggiunta l'istruzione END FUNCTION. Il valore restituito dalla function viene inserito nell'espressione dell'unità chiamante che contiene il suo nome.

Il nome di una funzione *deve* apparire sul lato sinistro di almeno un'istruzione di assegnazione. Il valore che viene assegnato a *nome_function*, quando la funzione restituisce il controllo all'unità chiamante, è il *valore della funzione*.

La lista degli argomenti può essere vuota nel caso in cui la funzione può svolgere i suoi calcoli senza argomenti di input. Tuttavia, le parentesi che contengono la lista degli argomenti sono *obbligatorie* anche se la lista è vuota.

Poiché una funzione restituisce un valore, è sempre necessario assegnarle un *tipo*. Se è utilizzata l'istruzione IMPLICIT NONE il tipo di dati della funzione deve essere dichiarato sia nella procedura che nelle unità chiamanti. Viceversa, se l'istruzione IMPLICIT NONE non è utilizzata, alla funzione saranno applicate le convenzioni standard del Fortran per i tipi di dati predefiniti.

5.2.3 Argomenti attuali e argomenti fittizi

Le procedure sono utilizzate per assolvere a compiti ben precisi. L'attivazione di una procedura e la sua esecuzione presuppongono uno scambio di informazioni tra l'unità chiamante e la procedura. Il modo più comune per rendere dei dati disponibili ad una procedura è di passarli attraverso una *lista di argomenti* all'atto della chiamata di procedura.

Una lista di argomenti è semplicemente un insieme di variabili o di espressioni (o anche nomi di procedure). Gli argomenti di una istruzione di chiamata sono detti *argomenti attuali*, mentre quelli corrispondenti nell'istruzione di dichiarazione della procedura sono chiamati *argomenti fittizi*. Gli argomenti attuali e quelli fittizi vengono detti *argomenti associati*: essi si corrispondono *per posizione*, cioè il primo argomento attuale corrisponde al primo argomento fittizio, il secondo argomento attuale al secondo argomento fittizio, e così via. Oltre al *numero*, anche il *tipo* di dati e il *rango* degli argomenti attuali e fittizi devono corrispondere *esattamente*. *Non* è, invece, necessario che essi si corrispondano anche per nome.

Quando una procedura viene invocata, l'unità chiamante passa, per ciascun parametro formale, l'*indirizzo di memoria* del corrispondente parametro attuale. Pertanto diventano direttamente utilizzabili dal sottoprogramma gli indirizzi delle locazioni in cui sono memorizzati i valori dei parametri attuali. In altre parole, gli argomenti fittizi di una procedura sono completamente definiti soltanto al momento dell'attivazione della procedura, quando ognuno di essi identifica la stessa locazione di memoria del corrispondente argomento effettivo. Questa modalità di scambio delle informazioni dall'unità chiamante alla procedura chiamata è detto *passaggio di dati per riferimento* in quanto all'atto della chiamata ciò che viene effettivamente passato sono i *puntatori* ai valori attuali e non i *valori* in sé.

Quando un elemento della lista di argomenti è un *array*, ciò che viene passato dall'unità chiamante alla procedura invocata è la locazione di memoria del *primo* elemento dell'array. Gli indirizzi dei successivi elementi dell'array vengono determinati, a partire dal primo, in base al numero di dimensioni ed ai limiti di tali dimensioni, secondo le modalità di immagazzinamento delle variabili dimensionate tipica del Fortran.

Infine, nel caso in cui il parametro effettivo sia un'*espressione*, essa viene calcolata prima della chiamata di procedura ed il suo valore viene assegnato ad una variabile interna del programma chiamante. Tale variabile interna rappresenta il parametro effettivo al quale la procedura accede. Quanto detto si applica anche quando il parametro effettivo sia una costante senza nome.

E' chiaro che, alterando il valore di un argomento fittizio, una procedura può cambiare il valore di un argomento attuale.

- Una *subroutine* è usata per cambiare il valore di uno o più argomenti. Nel seguente esempio, la subroutine `swap` scambia i valori di due array reali:

```
REAL, DIMENSION(10) :: x, y
...
CALL swap(x,y)

SUBROUTINE swap(a,b)
   REAL, DIMENSION(10) :: a, b, temp
      temp = a
      a = b
      b = temp
END SUBROUTINE swap
```

- Una *function* è usata per generare un singolo risultato *in funzione* dei suoi argomenti. Nel seguente esempio la funzione `line` calcola il valore dell'ordinata, nota l'ascissa, di un punto di una retta:

```
REAL :: y, x, c
...
x=1.0; c=2.5
y=line(3.4,x,c)
```

```
FUNCTION line(m,x,cost)
   REAL :: line
   REAL :: m, x, cost
      line = m*x+cost
END FUNCTION line
```

Il *nome della function*, `line`, è trattato esattamente come una *variabile*: esso deve, perciò, essere dichiarato con lo stesso tipo della variabile `y` che serve ad immagazzinare il valore della funzione nell'unità chiamante.

Va sottolineato che ad ogni parametro formale *di uscita* deve corrispondere come parametro attuale una variabile oppure un elemento di variabile dimensionata. In altri termini, una *costante* o una *espressione* non devono essere usate come parametro attuale in corrispondenza di un parametro formale il cui valore venga definito oppure modificato nel corpo del sottoprogramma. Infatti, se l'argomento attuale è una costante, qualunque modifica del corrispondente argomento fittizio provoca un cambiamento del valore della costante che, invece, per definizione deve rimanere inalterato durante tutta l'esecuzione del programma. Il divieto di utilizzare un'espressione in corrispondenza di un argomento fittizio di uscita deriva, invece, dal fatto che l'associazione tra l'espressione ed il corrispondente parametro formale avviene mediante l'indirizzo della locazione nella quale, al momento della chiamata della procedura, è memorizzato il valore dell'espressione. Questo indirizzo viene trasmesso al sottoprogramma ma *non* è noto all'unità di programma attivante che, pertanto, non lo può utilizzare. Si osservi, infine, che se due parametri attuali coincidono, nessuno dei corrispondenti parametri formali può essere un argomento di uscita.

5.2.4 Alcune note sull'utilizzo delle procedure

A volte è possibile migliorare l'efficienza di un programma semplicemente evitando di invocare una procedura quando non strettamente necessario. Calcolare, infatti, il valore di una funzione o degli argomenti di uscita di una subroutine è, di norma, un'operazione molto più lenta rispetto alla valutazione di una comune espressione all'interno, ad esempio, del programma principale. Ad esempio, se lo stesso valore di una funzione è richiesto più volte durante l'esecuzione del programma, piuttosto che invocare ogni volta questa funzione può essere comodo precomputarne i valori "utili" e sostituire, nel corso del programma, le chiamate di procedura con la lettura dei valori precalcolati immagazzinati in una opportuna struttura (*table lookup*). Ciò, naturalmente, non deve essere visto soltanto in termini di tempo necessario affinché una funzione venga calcolata ma anche riguardo al numero di volte che la stessa funzione viene invocata con il medesimo input. C'è però da considerare il fatto che, nei computer di moderna concezione, l'esecuzione di istruzioni semplici (quale è, ad esempio, una comune assegnazione) è, di norma, notevolmente più rapida dell'acquisizione di un dato in memoria (talvolta questa differenza si può quantificare in diversi ordini di grandezza). Per cui, se è richiesto un "costoso" accesso alla memoria ogni qualvolta si necessita di un valore di questo tipo (ad esempio se l'accesso deve avvenire a dati contenuti su file esterni come quelli dell'*hard disk*) una soluzione

più rapida può essere calcolarlo "al volo". Se, tuttavia, questo valore è utilizzato molte volte mentre risiede nella *cache memory* del computer, può essere più conveniente precomputarlo.

Quando alcuni degli argomenti di una procedura sono costanti la procedura stessa può essere "clonata" nel senso che è possibile costruire una nuova routine da quella originale semplicemente eliminando i parametri formali superflui e sostituendo nel corpo della procedura il valore della costante ad ogni ricorrenza del parametro. Naturalmente questa *function cloning* può essere efficacemente sfruttata per semplificare alcune operazioni algebriche pervenendo così a funzioni o subroutine più snelle e dalla più rapida esecuzione. Un esempio di "clonazione" di subroutine è riportato di seguito e mostra come sia possibile sfruttare efficacemente questa operazione per semplificare alcune espressioni ad esempio secondo il meccanismo di *strength reduction*:

```
! codice originale
CALL mysub(a,2)
...
SUBROUTINE mysub(x,p)
   IMPLICIT NONE
   REAL,DIMENSION(:) :: x
   INTEGER :: p
      x = x**p + REAL(p-2)
      ...
END SUBROUTINE mysub

! codice ottimizzato
CALL mysub2(a)
...
SUBROUTINE mysub2(x)
   IMPLICIT NONE
   REAL,DIMENSION(:) :: x
      x = x*x
      ...
END SUBROUTINE mysub
```

In questo caso nel caso l'elevamento al quadrato è stato sostituito da una comune moltiplicazione del fattore per sé stesso. L'efficacia di questa soluzione risiede nel fatto che l'operazione di prodotto viene normalmente eseguita dieci volte più velocemente dell'elevamento a potenza.

5.3 Procedure Interne

Un'unità di programma può contenere una o più *procedure interne* le quali, tuttavia, non possono contenere ulteriori procedure interne (in altre parole non è possibile una loro nidificazione). Le procedure interne vengono elencate al termine dell'unità di programma ospite, e sono precedute dall'istruzione `CONTAINS`. Di seguito si riporta un esempio di procedura interna ospitata in un programma principale:

```
PROGRAM main
IMPLICIT NONE
REAL :: a, b, c, mainsum
...
mainsum = somma()
...
CONTAINS
   FUNCTION somma()
      IMPLICIT NONE
      REAL :: somma   ! N.B.: Le variabili a, b, c sono definite nel main
         somma = a+b+c
   END FUNCTION
END PROGRAM
```

Tutte le procedure interne hanno totale visibilità delle variabili definite nell'unità ospite pertanto non è necessario ridefinirle. Si dice che le variabili dell'unità ospite sono accessibili all'unità interna per *host association*. Allo stesso modo non sarebbe necessario ripetere l'istruzione `IMPLICIT NONE` nelle unità interne, tuttavia, per motivi di chiarezza è sempre preferibile farlo. Viceversa, l'unità ospite *non* ha visibilità alcuna sulle variabili definite localmente alla procedura interna.

Oltre che per *host association*, le variabili dell'unità ospite possono essere passate alla procedura interna a mezzo della comune *associazione per argomento*. Non c'è differenza sostanziale tra i due meccanismi, che possono anche essere combinati, come nel seguente esempio:

```
PROGRAM main
IMPLICIT NONE
REAL :: x, y, z1, z2
...
CALL sub(z1,z2)
...
CONTAINS
   SUBROUTINE sub(z1,z2)
      IMPLICIT NONE
      REAL, INTENT(OUT) :: z1, z2
         z1 = 2*x-y
         z2 = x**2
   END SUBROUTINE
END PROGRAM main
```

Come ulteriore esempio, si guardi il programma seguente il quale invoca due funzioni interne, entrambe facenti uso delle stesse variabili `a`, `b` e `c` definite nel main, con la differenza che alla prima procedura le variabili vengono passate attraverso una lista di parametri attuali, alla seconda per associazione di host:

```
PROGRAM  HeronFormula
```

```
! Scopo: Calcolare l'area di un triangolo nota la lunghezza
! dei suoi lati
  IMPLICIT  NONE
  REAL :: a, b, c, TriangleArea
  DO
     WRITE(*,*) "Inserire le lunghezze dei lati: "
     READ(*,*)  a, b, c
     WRITE(*,*) "I dati inseriti sono: ", a, b, c
     IF (TriangleTest(a,b,c))  THEN
        EXIT
     ELSE
        WRITE(*,*) "Attenzione: il triangolo e' degenere"
        WRITE(*,*) "Reinserire i dati:"
     END IF
  END DO
  TriangleArea = Area()
  WRITE(*,*)  "L'area del triangolo e':", TriangleArea
CONTAINS
! ------------------------------------------------------------
! LOGICAL FUNCTION  TriangleTest(a,b,c) :
! Queste funzione riceve, attraverso i suoi parametri formali,
! tre valori reali e verifica se questi possono essere lati di
! un triangolo. In particolare controlla se:
!    (1)  tutti gli argomenti sono positivi
!    (2)  la somma di due lati e' sempre maggiore del terzo
! Se il test e' superato questa funzione restituisce .TRUE.,
! altrimenti restituisce .FALSE.
! ------------------------------------------------------------
   LOGICAL FUNCTION  TriangleTest(a,b,c)
      IMPLICIT  NONE
      REAL, INTENT(IN) :: a, b, c
      LOGICAL          :: test1, test2
         test1 = (a>0.0).AND.(b >0.0).AND.(c>0.0)
         test2 = (a+b>c).AND.(a+c>b).AND.(b+c>a)
         TriangleTest = test1.AND.test2  ! NB: Entrambe le condizioni
                                         !     devono essere verificate
   END FUNCTION  TriangleTest
! ------------------------------------------------------------
! REAL FUNCTION  Area() :
!  Questa funzione calcola l'area di un triangolo a mezzo della
!  formula di Erone. Le lunghezze a, b e c dei lati del triangolo
!  sono mutuate per associazione di host.
! ------------------------------------------------------------
```

```
   REAL FUNCTION  Area()
      IMPLICIT  NONE
      REAL :: s
         s = (a+b+c)/2.0
         Area = SQRT(s*(s-a)*(s-b)*(s-c))
   END FUNCTION  Area
END PROGRAM  HeronFormula
```

Se una procedura interna dichiara un oggetto con lo stesso nome di una variabile *già dichiarata* nell'unità di programma ospite, questa nuova variabile si "sovrappone" a quella dichiarata nell'unità esterna per tutta la durata dell'esecuzione della procedura. Ad esempio, nel seguente programma le due variabili `x` dichiarate, rispettivamente, nel main e nella procedura interna, sono totalmente diverse ed indipendenti l'una dall'altra.

```
PROGRAM prog
IMPLICIT NONE
REAL :: x, y, z
...
CONTAINS
   REAL FUNCTION func()
      IMPLICIT NONE
      REAL :: a, b, x
         ...
   END FUNCTION func
END PROGRAM prog
```

5.4 Variabili locali

Vengono definite come *locali* tutte le variabili dichiarate all'interno di una procedura e che *non* facciano parte della lista di argomenti formali. Ad esempio, nella seguente procedura:

```
SUBROUTINE mia_sub(m,n)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: m, n
   REAL :: a
   REAL, DIMENSION(m,n) :: x
   ...
END SUBROUTINE mia_sub
```

le variabili `a` ed `x` sono oggetti locali. Essi si caratterizzano per il fatto che:

- Sono creati ogni qualvolta la procedura è invocata.
- Sono distrutti quando termina l'esecuzione della procedura.

- Il loro valore *non* è conservato tra due chiamate consecutive della procedura.
- "Non esistono" (ossia non è riservata loro memoria) tra due chiamate consecutive di procedura.

5.4.1 Attributo e istruzione SAVE

Secondo le specifiche standard del linguaggio, i valori di tutti gli array e delle variabili *locali* di una procedura diventano indefiniti all'atto dell'uscita dalla procedura. La volta successiva che detta procedura viene chiamata, i valori degli array e delle variabili locali potrebbero non essere stati conservati, ciò dipendendo esclusivamente dal particolare compilatore. Il Fortran 90/95 dispone, tuttavia, di uno strumento che permette di mantenere inalterati i valori degli array e delle variabili locali fra due chiamate consecutive di procedura. Questo strumento è l'*attributo* `SAVE` che, come qualsiasi altro attributo, deve essere specificato all'interno di una istruzione di dichiarazione di tipo: qualsiasi variabile locale dichiarata con l'attributo `SAVE` sarà salvata fra due chiamate di procedura. La sintassi di una istruzione di definizione di variabile con l'attributo `SAVE` è la seguente:

```
tipo, SAVE ::  lista_di_variabili
```

mentre un esempio valido è il seguente:

```
COMPLEX, SAVE :: x, y, z
```

Le variabili locali definite con l'attributo `SAVE`, pertanto, si comportano esattamente come variabili `static` del C.

Si noti che qualsiasi variabile locale che venga inizializzata in un'istruzione di dichiarazione di tipo viene automaticamente salvata, indipendentemente dal fatto che venga specificato o meno l'attributo `SAVE`. Ad esempio, le seguenti variabili saranno salvate entrambe fra due chiamate consecutive delle procedure che le contengono:

```
REAL, SAVE :: x1=0.
REAL :: x2=0.
```

Il Fortran 90/95 dispone, inoltre, dell'*istruzione* `SAVE`. Si tratta di un'istruzione non eseguibile che va inclusa nella sezione dichiarativa della procedura con le istruzioni di dichiarazione di tipo. Tutte le variabili locali elencate nell'istruzione `SAVE` restano inalterate fra due chiamate consecutive della procedura. Se nessuna variabile viene specificata nell'istruzione `SAVE` saranno salvate *tutte* le variabili locali. Il formato dell'istruzione `SAVE` è il seguente:

```
SAVE ::  lista_di_variabili
```

o semplicemente:

```
SAVE
```

A conclusione dell'argomento si sottolinea l'ovvia inutilità dell'uso dell'attributo o dell'istruzione `SAVE` relativamente ai dati dichiarati con l'attributo `PARAMETER`. E' altresì evidente che l'attributo e l'istruzione `SAVE` non hanno alcun significato in un *programma principale* poiché l'uscita dal main costituisce la fine dell'esecuzione del programma.

5.5 Moduli

Un *modulo* è un'unità di programma compilato separatamente che contiene le definizioni e i valori iniziali dei dati che devono essere condivisi fra varie unità di programma. Se il nome del modulo è incluso, all'interno di un'unità di programma, in un'istruzione `USE`, i dati dichiarati nel modulo potranno essere liberamente usati all'interno di quella unità di programma. In altre parole, ogni unità di programma che usa un modulo avrà accesso agli stessi dati.

Un modulo inizia con un'istruzione `MODULE`, che gli assegna un nome, e termina con l'istruzione `END MODULE` che, facoltativamente, può essere seguita dal nome del modulo. Fra queste due istruzioni vanno inserite le dichiarazioni dei dati da condividere.

Il formato di un generico modulo è il seguente:

```
MODULE nome_modulo
  [istruzioni di definizione di tipo]
  [istruzioni di dichiarazione di tipo]
[CONTAINS
  procedure interne]
END [MODULE [nome_modulo]]
```

Un esempio di modulo è il seguente:

```
MODULE mio_modulo
   IMPLICIT NONE
   SAVE
   REAL, PARAMETER :: pi=3.14
   INTEGER :: m=5, n=5
   REAL, DIMENSION(n) :: vettore
END MODULE mio_modulo
```

L'uso dell'istruzione `SAVE` assicura che tutti i dati dichiarati nel modulo restino inalterati fra due chiamate di procedura differenti: questa istruzione dovrebbe essere inclusa in qualsiasi modulo che dichiara dati da condividere. Per utilizzare i valori di un modulo, un'unità di programma deve dichiarare il nome del modulo in un'istruzione `USE`. La forma dell'istruzione `USE` è la seguente:

```
USE nome_modulo
```

In un'unità di programma le istruzioni `USE` devono essere poste prima di tutte le altre (fatta eccezione, ovviamente, per le istruzioni `PROGRAM`, `SUBROUTINE`, `FUNCTION` e dei commenti i quali ultimi possono essere inseriti ovunque). Il processo per accedere alle informazioni in un modulo con un'istruzione `USE` è chiamato *associazione di* `USE`.

Le righe seguenti mostrano un esempio di utilizzo del modulo `mio_modulo` allo scopo di creare una condivisione di dati tra un programma principale ed una subroutine:

```
PROGRAM test_mio_modulo
  USE mio_modulo
```

```
  IMPLICIT NONE
  vettore=pi*(/1.,-2.,3.,-4.,5./)
  DO i=1,m
     ...
  END DO
  CALL sub
END PROGRAM test_mio_modulo

SUBROUTINE sub
   USE mio_modulo
   IMPLICIT NONE
      WRITE(*,*) vettore
END SUBROUTINE sub
```

Un'applicazione particolarmente comune per un modulo è quella di realizzare un archivio di *costanti con nome* di interesse ed i cui valori possano essere attinti da svariate procedure. Un esempio di questo tipo di utilizzo è rappresentato dal codice seguente:

```
MODULE trig_consts
! Scopo: definire alcune costanti di interesse in trigonometria
! LEGENDA:
!  pi: pi greco
!  r2d: fattore di conversione radianti->gradi
!  rdg: fattore di conversione gradi->radianti
   IMPLICIT NONE
   SAVE
   REAL(KIND(1.D0)), PARAMETER :: pi = 3.141592653589, &
                                  r2d = 180.0d0/pi,    &
                                  d2r = pi/180.0d0
END MODULE trig_consts

PROGRAM calculate
  USE trig_consts
  IMPLICIT NONE
     WRITE(*,*) SIN(30.0*d2r)
END PROGRAM calculate
```

In particolare è possibile scrivere un modulo che contenga le definizioni dei *numeri di kind* degli oggetti che verranno utilizzati nel corso di un programma e delle procedure da esso invocate:

```
MODULE def_kind
INTEGER, PARAMETER :: &
    short = SELECTED_INT_KIND(4),      & ! interi ad almeno 4 cifre
    long  = SELECTED_INT_KIND(9),      & ! interi ad almeno 9 cifre
    dble  = SELECTED_REAL_KIND(15,200)   ! reali a 15 cifre fino a 10**200
```

```
END MODULE def_kind
```

Pertanto, in una qualsiasi unità di programma sede dell'istruzione:

```
USE def_kind
```

saranno perfettamente lecite le seguenti istruzioni di dichiarazione:

```
INTEGER(short) :: myimage(1024,1024)
INTEGER(long)  :: counter
REAL(double)   :: processed_data(2000,2000)
```

Chiaramente, su sistemi di elaborazione differenti le costanti `short`, `long` o `double` potranno avere valori diversi ma questo non condizionerà minimamente la portabilità del codice.

Un interessante esempio di modulo che sposa le caratteristiche anzidette di "archivio" di dati e di precisione portabile è riportato di seguito. Questo modulo, ben lungi dall'essere un mero esempio didattico, elenca dapprima i valori dei parametri di kind per oggetti di tipo `INTEGER` e `REAL` di differente range e precisione, quindi i valori delle principali costanti fisiche ed astronomiche di interesse.

```
MODULE Costanti_fisiche
!-----------------------------------------------------------------------
!   Descrizione:
!  ==============
!   Questo modulo contiene i valori delle piu' comuni costanti
!   fisiche ed astronomiche. Inoltre esso identifica i corretti
!   parametri di kind per la macchina in uso
!    Autore:      B. Raucci
!    E-mail:      biagio@raucci.net
!-----------------------------------------------------------------------
  IMPLICIT NONE
  SAVE
! Determina la precisione ed il range del sistema di elaborazione in uso
  INTEGER,PARAMETER :: r8 = SELECTED_REAL_KIND(15,307)
  INTEGER,PARAMETER :: r4 = SELECTED_REAL_KIND(6,37)
  INTEGER,PARAMETER :: i1 = SELECTED_INT_KIND(2)
  INTEGER,PARAMETER :: i2 = SELECTED_INT_KIND(4)
  INTEGER,PARAMETER :: i4 = SELECTED_INT_KIND(8)

! Identifica il numero piu' piccolo rappresentabile in macchina
! ed il numero di cifre significative per i numeri di kind r4 ed r8
  REAL(r4),PARAMETER :: tiny_r4    = TINY(1.0_r4)
  REAL(r8),PARAMETER :: tiny_r8    = TINY(1.0_r8)
  INTEGER, PARAMETER :: sig_fig_r4 = PRECISION(1.0_r4)
  INTEGER, PARAMETER :: sig_fig_r8 = PRECISION(1.0_r8)
```

```
  REAL(r4),PARAMETER :: eps_r4     = 10.0_r4**(-sig_fig_r4)
  REAL(r8),PARAMETER :: eps_r8     = 10.0_r8**(-sig_fig_r8)

! Valori di pi greco e della costante di Nepero
  REAL(r8),PARAMETER :: pi         = &
              3.14159265358979323846264338327950288419716939937510_r8
  REAL(r8),PARAMETER :: two_pi     = 2*pi
  REAL(r8),PARAMETER :: four_pi    = 4*pi
  REAL(r8),PARAMETER :: four_pi_o3 = four_pi/3
  REAL(r8),PARAMETER :: pi_over_2  = pi/2
  REAL(r8),PARAMETER :: natural_e  = &
              2.71828182845904523536028747135266249775724709369995_r8

! Fattori di conversione da gradi a radianti e da radianti a gradi
  REAL(r8),PARAMETER :: degrees_to_radians = pi/180
  REAL(r8),PARAMETER :: radians_to_degrees = 180/pi

! Costanti fisiche (espresse nel SI, salvo avviso contrario)
  REAL(r4),PARAMETER :: G = 6.673E-11_r4      !Costante di Gravitazione
                                              ! Universale (N m^2/kg^2)
  REAL(r8),PARAMETER :: c = 2.99792458E08_r8  !Velocita' della luce (m/s)
  REAL(r8),PARAMETER :: h = 6.62606876E-34_r8 !Costante di Planck (J s)
  REAL(r8),PARAMETER :: hbar = h/two_pi       !hbar (J s)
  REAL(r8),PARAMETER :: k_B  = 1.3806503E-23_r8  !Costante di Boltzmann (J/K)
  REAL(r8),PARAMETER :: sigma = &             !Costante di Stefan-Boltzmann
               2*pi**5*k_B**4/(15*c**2*h**3) ! (J/m^2/s/K^4)
  REAL(r8),PARAMETER :: a_rad = 4*sigma/c     !Costante di radiazione
                                              ! (J/m^3/K)
  REAL(r8),PARAMETER :: m_p = 1.67262158E-27_r8   !Massa del protone (kg)
  REAL(r8),PARAMETER :: m_n = 1.67492716E-27_r8   !Massa del neutrone (kg)
  REAL(r8),PARAMETER :: m_e = 9.1093818897E-31_r8 !Massa dell'elettrone (kg)
  REAL(r8),PARAMETER :: m_H = 1.673532499E-27_r8 !Massa dell'idrogeno (kg)
  REAL(r8),PARAMETER :: uma = 1.66053873E-27_r8  !Unita' di massa atomica (kg)
  REAL(r8),PARAMETER :: e   = 1.602176462E-19_r8 !Carica dell'elettrone (C)
  REAL(r8),PARAMETER :: eV  = 1.602176462E-19_r8 !1 elettron-volt (J)
  REAL(r8),PARAMETER :: keV = 1.0E3_r8*eV        !1 kilo eV (J)
  REAL(r8),PARAMETER :: MeV = 1.0E6_r8*eV        !1 Mega eV (J)
  REAL(r8),PARAMETER :: GeV = 1.0E9_r8*eV        !1 Giga eV (J)
  REAL(r8),PARAMETER :: N_A = 6.02214199E23_r8   !Numero di Avogadro
  REAL(r8),PARAMETER :: R   = 8.314472_r8        !Costante universale
                                                 ! dei gas (J/mol/K)
! Costanti di tempo
  REAL(r8),PARAMETER :: hr   = 3600           !1 ora (in s)
```

```
  REAL(r8),PARAMETER :: day  = 24*hr          !1 giorno solare(in s)
  REAL(r8),PARAMETER :: yr   = 3.155815E7_r8  !1 anno siderale (in s)
  REAL(r8),PARAMETER :: J_yr = 365.25*day     !1 anno giuliano (in s)

! Costanti di distanze astronomiche (in unita' del SI)
  REAL(r8),PARAMETER :: AU = 1.4959787066E11_r8  !Unita' astronomica (m)
  REAL(r8),PARAMETER :: pc = 206264.806_r8*AU    !Parsec (m)
  REAL(r8),PARAMETER :: ly = c*J_yr              !Anno luce (giuliano; m)

! Costanti di interesse astronomico (in unita' del SI)
  REAL(r4),PARAMETER :: M_Sun = 1.9891E30_r4   !Massa del Sole (kg)
  REAL(r4),PARAMETER :: R_Sun = 6.95508E8_r4   !Raggio del Sole (m)
  REAL(r4),PARAMETER :: S_Sun = 1.365E3_r4     !Radiazione solare(J/m^2/s)
  REAL(r4),PARAMETER :: L_Sun = &
                         four_pi*AU**2*S_Sun   !Luminosita' del Sole (W)
  REAL(r4),PARAMETER :: Te_Sun  = &
          (L_Sun/(four_pi*R_Sun**2*sigma))**(0.25) !Temperature del Sole (K)
  REAL(r4),PARAMETER :: M_Earth = 5.9736E24_r8     !Massa della Terra (kg)
  REAL(r8),PARAMETER :: R_Earth = 6.378136E6_r8    !Raggio della Terra (m)

END MODULE Costanti_fisiche
```

Come visto, i moduli sono particolarmente utili per condividere grandi quantità di dati fra più unità di programma e fra un gruppo di procedure correlate, nascondendoli a tutte le altre unità. In questo modo si creano delle variabili *globali*. Tuttavia, è possibile (e talvolta opportuno) limitare il numero delle variabili definite in un modulo a cui una unità di programma può avere accesso. Questa peculiarità aumenta la "sicurezza" di un codice assicurando che un'unità di programma che usi un modulo non ne cambi accidentalmente il valore di una variabile. A tale scopo è possibile associare all'istruzione `USE` il qualificatore `ONLY`:

```
USE nome_modulo, ONLY: lista_di_variabili
```

Ad esempio, se all'istruzione:

```
USE global
```

si sostituisce l'istruzione:

```
USE global, ONLY: a, b, c
```

l'unità di programma in cui questa istruzione è posta può avere accesso alle variabili `a`, `b`, `c`, quali che siano le altre variabili definite nel modulo `global`.

L'uso dello specificatore `ONLY` previene la possibilità che l'unità di programma che usi un modulo possa modificare per errore quei dati (contenuti nel modulo) dei quali, peraltro, *non* necessita. Come è noto, la maggior parte degli errori di battitura sono individuati dal compilatore grazie all'istruzione `IMPLICIT NONE` la quale rende illegale l'uso di variabili non dichiarate.

Tuttavia, se il nome digitato per errore coincide con il nome di una variabile del modulo usato, questo errore *non* sarà individuato dal compilatore. L'unico modo di ovviare a un simile inconveniente è, pertanto, quello di evitare che l'unità di programma, sede dell'associazione di `USE`, abbia visibilità di quelle variabili che non usa. Questa è, appunto, la funzione di `ONLY`.

Un problema che può insorgere con le variabili globali è quello noto col nome di *name clash*, ossia può accadere che uno stesso nome venga usato per identificare due differenti variabili in parti diverse di un programma. Questo problema può essere risolto facendo sì che una variabile *globale* possa essere riferita con un nome *locale*:

```
USE nome_modulo, nome_locale => nome_globale
```

Con il simbolo `=>` si fa semplicemente corrispondere un identificatore locale ad una variabile globale: a parte la similitudine formale, esso non ha nulla in comune con l'istruzione di associazione di un *puntatore*. A titolo di esempio, si consideri la seguente istruzione:

```
USE global, x=>a
```

In questo caso, l'unità di programma in cui questa istruzione è collocata avrà accesso a tutte le variabili definite nel modulo `global` ma sarà "costretto" a riferirsi alla variabile `a` con il nome locale `x`.

Chiaramente le due *facility*, `ONLY` e `=>`, possono essere combinate, come nel seguente esempio:

```
USE global, ONLY: a, b=>y
```

in modo da dare soluzioni particolarmente versatili ed efficienti.

Come si sarà potuto intuire leggendo queste pagine e, soprattutto, come risulterà ancora più chiaro proseguendo la lettura di queste note, l'uso di moduli in programma può tornare estremamente utile e può aiutare il programmatore a sviluppare codici efficienti, compatti e facilmente mantenibili. Tuttavia è doveroso segnalare anche alcuni "svantaggi" connessi al loro utilizzo e che sono, sostanzialmente, tutti quelli elencati di seguito:

- Ogni modulo deve essere compilato prima di ogni altra unità di programma che ne faccia uso; questo può significare, ad esempio, che il programma principale debba essere l'ultima unità ad essere compilata piuttosto che la prima, come sarebbe più logico.
- Un cambiamento apportato ad un modulo implica la ricompilazione di tutte le unità che lo usano, il che naturalmente, può condurre a processi di compilazioni assai lenti.
- Di solito la compilazione di un modulo produce un unico oggetto (di solito un file ".mod") che deve essere linkato *per intero* al resto del programma per cui una qualsiasi associazione di `USE` fa sì che tutto il codice del modulo debba essere presente nel programma eseguibile

5.5.1 Restrizione della visibilità di un modulo

In generale, *tutte* le entità definite in un modulo sono accessibili alle unità di programma che usano quel modulo. Tuttavia è spesso necessario restringere al minimo la visibilità delle entità

definite in un modulo facendo sì che le unità di programma che siano sede dell'associazione di `USE` vedano soltanto le entità strettamente necessarie. Restringere l'accesso al contenuto di un modulo porta, di norma, a scrivere programmi più facili da comprendere e mantenere. Questa operazione (detta di *data hiding*) può effettuarsi attraverso un uso, spesso combinato, degli attributi (o delle istruzioni) `PUBLIC` e `PRIVATE`. La sintassi degli attributi `PUBLIC` e `PRIVATE` è:

```
tipo, PRIVATE ::  lista_delle_variabili_''private''
tipo, PUBLIC ::  lista_delle_variabili_''pubbliche''
```

mentre la forma delle corrispondenti istruzioni `PUBLIC` e `PRIVATE` è:

```
tipo ::  lista_delle_variabili
PRIVATE [::]  lista_delle_variabili_''private''
PUBLIC [::]  lista_delle_variabili_''pubbliche''
```

Se ad una entità del modulo è applicato l'attributo (o l'istruzione) `PUBLIC`, allora detta entità sarà disponibile all'esterno del modulo a ciascuna unità di programma che usi il modulo. Viceversa, se ad una entità del modulo è applicato l'attributo (o l'istruzione) `PUBLIC`, allora detta entità *non* sarà visibile all'esterno del modulo ma sarà disponibile esclusivamente alle procedure *interne* al modulo. Si noti che *tutte* le entità di un modulo sono da ritenersi *pubbliche per default* sicché, in mancanza di una specifica di `PRIVATE`, *tutte* le variabili e le procedure di un modulo saranno accessibili esternamente. Le seguenti righe di programma rappresentano esempi di uso corretto degli attributi e delle istruzioni `PUBLIC` e `PRIVATE`:

```
INTEGER, PRIVATE :: m, n
REAL, PUBLIC :: x, y

INTEGER :: i, j
PRIVATE i, j
```

Se un modulo contiene una istruzione `PRIVATE` senza specificare una lista di variabili, allora *tutte* le variabili e le procedure definite nel modulo saranno da considerarsi locali al modulo. Ogni variabile o procedura che, invece, si vuole rendere accessibile all'esterno dovrà essere specificata appositamente con l'attributo o con un'istruzione `PUBLIC`. A chiarimento di quanto detto, si consideri il seguente esempio:

```
MODULE dati
   IMPLICIT NONE
   PRIVATE
   PUBLIC :: a, b, c
   REAL :: a, b, c
   ...
END MODULE dati
```

Particolare attenzione merita l'uso delle dichiarazioni `PUBLIC` e `PRIVATE` con oggetti di *tipo derivato*, come schematizzato di seguito.

- I componenti di un tipo di dati derivato definito in un modulo possono essere resi inaccessibili all'esterno del modulo includendo l'istruzione `PRIVATE` all'interno della definizione di tipo. In tal caso il tipo di dati definito può ancora essere usato al di fuori del modulo, ma solamente come entità "unica", non essendo accessibili separatamente i suoi componenti. In particolare, in un'unità di programma che richiami il modulo in esame sarà possibile dichiarare variabili di quel tipo ma *non* sarà possibile lavorare con i *componenti singoli*. Un esempio di tipo di dati derivato con i componenti *privati* è il seguente:

```
TYPE mio_tipo
   PRIVATE
   REAL :: x
   REAL :: y
   REAL :: z
END TYPE mio_tipo
```

- Chiaramente, anche un *intero* tipo di dati derivato può essere dichiarato come *privato*, come nell'esempio seguente:

```
TYPE, PRIVATE :: mio_tipo
   REAL :: x
   REAL :: y
   REAL :: z
END TYPE mio_tipo
```

  In questo caso l'intero tipo di dati derivato sarà *non* accessibile all'esterno del modulo nel quale è definito ma sarà visibile solo all'interno del modulo stesso.

- E' possibile, infine, dichiarare come *private* variabili di un tipo di dati derivato anche se questo è *pubblico*, come è mostrato nel seguente esempio:

```
TYPE :: mio_tipo
   REAL :: x
   REAL :: y
   REAL :: z
END TYPE mio_tipo
TYPE(mio_tipo), PRIVATE :: tripletta
```

  In questo caso il tipo di dati derivato `mio_tipo` è accessibile a tutte le unità di programma che facciano uso del modulo, mentre la variabile `tripletta` può essere usata soltanto all'interno del modulo.

Si noti, inoltre, che anche un tipo di dati derivato definito con l'attributo `SEQUENCE` può avere componenti privati, ad esempio:

```
TYPE :: storage
   PRIVATE
```

```
   SEQUENCE
   INTEGER :: num
   REAL, DIMENSION(100) :: vet
END TYPE storage
```

In tal caso, gli attributi `PRIVATE` e `SEQUENCE` possono essere scambiati di posto ma in ogni caso essi devono rappresentare la seconda e la terza istruzione della definizione.

E' interessante notare, in conclusione, che, così come le variabili (e le costanti) definite in un modulo, anche una *procedura di modulo* può essere dichiarata pubblica o privata (naturalmente tutte le procedure di modulo sono da considerarsi `PUBLIC` *per default*). Ad esempio, nel seguente modulo viene usata l'istruzione `PUBLIC` per consentire la visibilità della subroutine `lunghezza` alle unità di programma esterne al modulo, e l'istruzione `PRIVATE` per impedire, invece, la visibilità della subroutine `quadrato`:

```
MODULE lunghezza_vettore
   PRIVATE :: quadrato
   PUBLIC :: lunghezza
   CONTAINS
      SUBROUTINE lunghezza(x,y,z)
         REAL, INTENT(IN) :: x, y
         REAL, INTENT(OUT) :: z
            CALL quadrato(x,y)
            z = SQRT(x+y)
            RETURN
      END SUBROUTINE
      SUBROUTINE quadrato(x1,y1)
         REAL, INTENT(INOUT) :: x1, y1
            x1 = x1**2
            y1 = y1**2
            RETURN
      END SUBROUTINE
END MODULE
```

5.5.2 Procedure di Modulo

Oltre ai dati, un modulo può contenere anche intere subroutine e funzioni che vengono, pertanto, chiamate *procedure di modulo*. Queste procedure sono compilate come parte del modulo e vengono messe a disposizione dell'unità di programma che sia sede dell'associazione di `USE`. Le procedure che sono incluse all'interno di un modulo devono trovarsi *dopo* le dichiarazioni dei dati e devono essere precedute dall'istruzione `CONTAINS`. Questa istruzione indica al compilatore che le istruzioni successive sono procedure di modulo. Nel seguente esempio, la subroutine `sub` è contenuta all'interno del modulo `mio_modulo`:

```
MODULE mio_modulo
```

```
IMPLICIT NONE
SAVE
! dichiarazione dei dati condivisi
...
CONTAINS
   SUBROUTINE sub(a,b,c,m,conv)
      IMPLICIT NONE
      REAL, DIMENSION(3), INTENT(IN) :: a
      REAL, INTENT(IN) :: b, c
      INTEGER, INTENT(OUT) :: m
      LOGICAL, INTENT(OUT) :: conv
! sezione esecutiva
      ...
   END SUBROUTINE sub
END MODULE mio_modulo
```

La subroutine `sub` è "a disposizione" di una qualunque unità di programma che faccia uso del modulo `mio_modulo` e può da questa essere usata, come una normale subroutine, mediante l'istruzione `CALL`, così come illustrato dal seguente esempio:

```
PROGRAM mio_prog
  USE mio_modulo
  IMPLICIT NONE
  ...
  CALL sub(x,y,z,m,conv)
  ...
END PROGRAM
```

Si noti che anche per una procedura di modulo è possibile definire una procedura interna, come ben descritto dal seguente programma:

```
MODULE miomodulo
! Esempio di sottoprogrammi interni ad procedure di modulo
! Sezione dichiarativa
   IMPLICIT NONE
   TYPE :: new
      INTEGER :: j, k
   END TYPE new
! Inizializzazione di x a mezzo di costruttore:
   TYPE(new) :: x=new(1234,5678)
! Sezione "interna"
CONTAINS
   SUBROUTINE sub_mod_1()       ! sottoprogramma di modulo
! start subroutine sub_mod_1
      CALL sub_int()
```

```
      RETURN
   CONTAINS
      SUBROUTINE sub_int()      ! sottoprogramma interno
! start subroutine sub_int_1
         CALL sub_mod_2(x%j,x%k)
         RETURN
      END SUBROUTINE sub_int
   END SUBROUTINE sub_mod_1
   SUBROUTINE sub_mod_2(i1,i2)  ! sottoprogramma di modulo
      INTEGER, INTENT (IN OUT) :: i1, i2
      INTEGER :: x
! start subroutine sub_mod_2
         x = i1
         i1 = i2
         i2 = x
      RETURN
   END SUBROUTINE sub_mod_2
END MODULE miomodulo
!
PROGRAM prova
   USE miomodulo
   CALL sub_mod_1( )
   WRITE(*,*) x
   STOP
END PROGRAM prova
```

5.6 Procedure Esterne

Le *procedure esterne* sono unità di programma (subroutine o function) *indipendenti*, compilate separatamente, tipicamente scritte su file del tutto separati dal resto del programma. Il corpo di una procedura esterna deve contenere tutte le frasi (dichiarazioni e istruzioni) indispensabili per una sua corretta ed *autonoma* compilazione. Così come un programma principale e un modulo, anche una procedura esterna può ospitare procedure interne. Il seguente frammento di codice mostra un esempio di procedura esterna invocata dal programma principale:

```
PROGRAM first
  REAL :: x
  x=second()
  ...
END PROGRAM first

FUNCTION second()    ! funzione esterna
   REAL :: second
```

```
      ...              ! N.B. nessuna ''host association''
END FUNCTION second
```

Le procedure esterne *non* possono in alcun caso condividere dati tramite una *host association*. Il trasferimento dei dati attraverso gli argomenti è il modo più comune di condividere dati con una procedura esterna. Le procedure esterne possono essere invocate da tutti gli altri tipi di procedura.

5.7 Interfacce esplicite ed implicite

Una *interfaccia* ha luogo ogni qualvolta una unità di programma invoca un'altra unità di programma. Affinché il programma funzioni correttamente è necessario che gli argomenti attuali dell'unità di programma chiamante siano "consistenti" con i corrispondenti parametri formali della procedura chiamata: questa condizione è verificata dal compilatore attraverso un test sull'interfaccia.

Un problema serio può nascere quando una variabile occupa in una lista di argomenti attuali il posto che competerebbe ad un array. La procedura invocata non è, generalmente, in grado di comprendere la differenza tra una variabile scalare ed un array per cui essa tratterà quella variabile e quelle che seguono in termini di locazioni di memoria come se fossero parte di un unico array. In questo caso la procedura può modificare perfino il valore di una variabile che *non* le è stata passata *per riferimento* solo perché la sua locazione di memoria segue quella di un parametro attuale. Problemi come questo sono estremamente difficili da trovare e correggere quando si lavora con procedure esterne, a meno di non rendere *esplicita* l'interfaccia, come sarà chiaro tra breve.

Quando una procedura è compilata all'interno di un *modulo* e questo modulo è utilizzato dal programma chiamante, tutti i dettagli dell'interfaccia vengono messi a disposizione del compilatore. Quando il programma chiamante viene compilato, il compilatore può controllare automaticamente il *numero* degli argomenti coinvolti nella chiamata della procedura, il *tipo* di ogni argomento, lo *scopo* (`INTENT`) di ogni argomento e *se* gli argomenti sono *array*. In sintesi, il compilatore può rilevare molti degli errori tipici che si possono commettere quando si usano delle procedure. Pertanto, nel caso di procedura compilata all'interno di un *modulo* e utilizzata con l'istruzione `USE` si parla di *interfaccia esplicita*, in quanto il compilatore conosce *esplicitamente* tutti i dettagli su ogni argomento della procedura; tutte le volte che questa procedura viene utilizzata, il compilatore controlla l'interfaccia per verificare che ogni componente sia utilizzato in modo appropriato. Anche nel caso di *procedure interne* tutte queste informazioni sono note al compilatore per cui, ancora, si parla di interfaccia esplicita: da questo punto di vista questa situazione non è dissimile dalla chiamata di una *procedura intrinseca*. Al contrario, nel caso di procedure *esterne* non incluse in un modulo si parla di *interfaccia implicita*: il compilatore non ha informazioni su queste procedure quando compila l'unità di programma chiamante, quindi *suppone* che il programmatore abbia specificato correttamente il numero, il tipo, lo scopo e le altre caratteristiche degli argomenti delle procedure: in caso di errore il programma si arresta in modo imprevedibile.

Nel seguente esempio il programma chiamante invoca una procedura definita all'interno di un modulo tentando di passarle un valore reale anziché uno intero: il compilatore risponderà certamente con un messaggio di errore:

```
MODULE my_sub
CONTAINS
   SUBROUTINE bad_arg(k)
      IMPLICIT NONE
      INTEGER, INTENT(IN) :: k
         WRITE(*,*) "k = ", k
   END SUBROUTINE
END MODULE

PROGRAM bad_call
  USE my_sub
  IMPLICIT NONE
  REAL :: x=1.
  CALL bad_arg(x)  ! illegale: il compilatore dara' un messaggio di errore
END PROGRAM
```

Nel caso in cui la subroutine `bad_arg` fosse stata una procedura *esterna* il compilatore *non* avrebbe dato alcun messaggio di errore ma il programma si sarebbe arrestato durante il run.

5.7.1 *Interface Block*

Anche per le *procedure esterne*, tuttavia, il Fortran 90 mette a disposizione uno strumento, l'*interface block*, a mezzo del quale è possibile rendere esplicita l'interfaccia. La forma generale di un *interface block* è:

```
INTERFACE
  corpo dell'interfaccia
END INTERFACE
```

Si noti che, contrariamente alle istruzioni `END` delle altre unità di programma, nel Fortran 90 le istruzioni `END INTERFACE` non possono avere un nome. Questa limitazione è stata rimossa dal Fortran 95 per cui, con il nuovo standard, la sintassi dell'*interface block* è diventata:

```
INTERFACE [nome_interfaccia]
  corpo dell'interfaccia
END INTERFACE [nome_interfaccia]
```

Il *corpo dell'interfaccia* comprende la istruzione di `FUNCTION` (o `SUBROUTINE`), le istruzioni di dichiarazione di tipo dei parametri formali, e le istruzioni `END FUNCTION` (o `END SUBROUTINE`). In altre parole, esso è l'esatta copia del sottoprogramma privato della dichiarazione delle variabili locali, della sua sezione esecutiva e delle eventuali procedure interne.

Concettualmente l'*interface block* può essere visto come l'equivalente dei *prototipi di funzione* del linguaggio C.

Ad esempio, considerata la seguente procedura:

```
! N.B.: Tutte le righe evidenziate con l'asterisco
!       devono far parte dell'interface block
SUBROUTINE expsum(n,k,x,sum)          ! (*)
   USE dati_comuni, ONLY: long        ! (*)
   IMPLICIT NONE                      ! (*)
   INTEGER, INTENT(IN) :: n           ! (*)
   REAL(long), INTENT(IN)  :: k, x    ! (*)
   REAL(long), INTENT(OUT) :: sum     ! (*)
   REAL(long) :: cool_time
      sum = 0.0
      DO i = 1, n
         sum = sum + exp(-i*k*x)
      END DO
END SUBROUTINE expsum                 ! (*)
```

la sua interfaccia è resa esplicita a mezzo del seguente *interface block*:

```
INTERFACE
   SUBROUTINE expsum(n,k,x,sum)
      USE dati_comuni, ONLY: long
      IMPLICIT NONE
      INTEGER, INTENT(IN) :: n
      REAL(long), INTENT(IN)  :: k, x
      REAL(long), INTENT(OUT) :: sum
   END SUBROUTINE expsum
END INTERFACE
```

Chiaramente, l'*interface block* deve essere contenuto all'interno dell'unità di programma chiamante e può contenere anche più di una procedura.

Di seguito si riportano in maniera organica tutti i casi in cui un *interface block* è obbligatorio. Alcuni di questi casi saranno più chiari proseguendo nella lettura di queste note.

- Un *interface block* è richiesto per una *procedura esterna* o una *procedura di modulo*:
  - che "definisca" o "sovrapponga" un operatore aritmetico o l'operatore di assegnazione.
  - che usi un *nome generico*.
- Un *interface block* è richiesto per una *procedura esterna* quando:
  - è invocata con argomenti che siano *parole chiave* e/o *opzionali*.
  - è una *funzione array* o una *funzione puntatore*, oppure una *funzione stringa di caratteri lunghezza presunta*.

- abbia un parametro formale che sia un *array fittizio di forma presunta*, un *puntatore* oppure un *target*.
- faccia parte di una lista di argomenti (in questo caso l'*interface block* è raccomandato ma non obbligatorio).

Il programma che segue, concepito per la risoluzione di una equazione non lineare con il classico *metodo di Newton-Raphson*, rappresenta un esempio di utilizzo di funzioni esterne la cui interfaccia sia resa esplicita attraverso l'uso di un *interface block*. Si ricorda che il metodo di Newton-Raphson si basa sull'applicazione della formula iterativa:

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

la quale, a partire da un valore iniziale x_0, produce una successione di valori $x_1, x_2, \ldots$ che, in molti casi, si caratterizza per una rapida convergenza verso un'approssimazione della radice $\overline{x}$ della funzione $f(x)$. L'efficacia e la rapidità del metodo dipendono, comunque, anche dalla scelta del valore iniziale e dalle caratteristiche della funzione nell'intorno della radice.

```
PROGRAM main
   IMPLICIT NONE
   INTEGER, PARAMETER :: dp=KIND(1D0)
   REAL(KIND=dp) :: x, eps
   INTEGER :: nmax
! LEGENDA:
!  eps:  approssimazione richiesta
!  nmax: numero massimo di iterazioni
!  x:    (IN) valore di tentativo / (OUT) radice della funzione
   INTERFACE
      FUNCTION f(x)
         IMPLICIT NONE
         INTEGER, PARAMETER :: dp=KIND(1D0)
         REAL(KIND=dp), INTENT(IN) :: x
         REAL(KIND=dp) :: f
      END FUNCTION f
!
      FUNCTION df(x)
         IMPLICIT NONE
         INTEGER, PARAMETER :: dp=KIND(1D0)
         REAL(KIND=dp), INTENT(IN) :: x
         REAL(KIND=dp) :: df
      END FUNCTION df
!
      SUBROUTINE newton(f,df,x,nmax,eps)
```

```
            IMPLICIT NONE
            INTEGER, PARAMETER :: dp = KIND(1D0)
            REAL(KIND=dp), INTENT(INOUT) :: x
            REAL(KIND=dp), INTENT(IN) :: eps
            REAL(KIND=dp), EXTERNAL :: f, df
            INTEGER, INTENT(IN) :: nmax
         END SUBROUTINE newton
      END INTERFACE
      eps = 1.0E-6_dp
      nmax = 100
      x = 4.0_dp
      CALL newton(f,df,x,nmax,eps)
   END PROGRAM main

   SUBROUTINE newton(f,df,x,nmax,eps)
   ! Scopo: applica l'algoritmo di Newton-Raphson per la valutazione
   !        dello zero di una funzione reale di una variabile reale
      IMPLICIT NONE
      INTEGER, PARAMETER :: dp=KIND(1D0)
      REAL(KIND=dp), INTENT(INOUT) :: x
      REAL(KIND=dp), INTENT(IN) :: eps
      REAL(KIND=dp), external :: f, df
      INTEGER, INTENT(IN) :: nmax
      REAL(KIND=dp) :: fx
      INTEGER :: n=0
         WRITE(UNIT=*,FMT='(1X,"n",12X,"x",19X,"f(x)",/,46("="))')
         fx = f(x)
         WRITE(UNIT=*,FMT="(1X,I2,1X,F20.16,1X,F20.16)") n, x, fx
         DO n = 1,nmax
            x = x - fx/df(x)
            fx = f(x)
            WRITE(UNIT=*,FMT="(1X,I2,1X,F20.16,1X,F20.16)") n, x, fx
            IF(ABS(fx)<eps) EXIT
         END DO
         IF(n==nmax) THEN  ! Convergenza non raggiunta
            WRITE(UNIT=*,FMT=*) "Raggiunto il numero massimo di iterazioni "
         ELSE              ! Convergenza raggiunta
            WRITE(UNIT=*,FMT='(/,1X,A25,F12.7)') "Radice della funzione: ",x
            WRITE(UNIT=*,FMT='(1X,A25,F12.7)') "Valore della funzione: ",fx
         END IF
         RETURN
   END SUBROUTINE newton
```

```
FUNCTION f(x)
! Funzione di cui si deve valutare la radice
   IMPLICIT NONE
   INTEGER, PARAMETER :: dp=KIND(1D0)
   REAL(KIND=dp) , INTENT(IN) :: x
   REAL(KIND=dp) :: f
      f = ((x-2.0_dp)*x+1.0_dp)*x-3.0_dp
      RETURN
END FUNCTION f

FUNCTION df(x)
! Derivata della funzione argomento
   IMPLICIT NONE
   INTEGER, PARAMETER  :: dp=KIND(1D0)
   REAL(KIND=dp), INTENT(IN) :: x
   REAL(KIND=dp) :: df
      df = (3.0_dp*x - 4.0_dp)*x + 1.0_dp
      RETURN
END FUNCTION df
```

L'output del programma così sviluppato è:

```
 n            x                    f(x)
=============================================
  0   4.0000000000000000  32.9999999999999982
  1   3.0000000000000000   9.0000000000000000
  2   2.4375000000000000   2.0368652343750000
  3   2.2130327163151095   0.2563633850614173
  4   2.1755549387214880   0.0064633614888132
  5   2.1745601006664458   0.0000044790680502
  6   2.1745594102933125   0.0000000000021574

   Radice della funzione:    2.1745594
   Valore della funzione:    0.0000000
```

5.8 Argomenti delle Procedure

5.8.1 L'attributo INTENT

In Fortran è possibile (oltre ad essere buona regola di programmazione) specificare in che modo il parametro formale verrà adoperato nella procedura. In altre parole è possibile "marcare" un parametro formale come parametro *di ingresso*, *di uscita* o *di ingresso/uscita*. Ciò può essere fatto a mezzo dell'attributo INTENT, il quale può assumere una delle seguenti forme:

- `INTENT(IN)` - il parametro formale *deve* avere un valore quando la procedura viene chiamata, tuttavia questo valore *non* può essere aggiornato dalla procedura stessa.
- `INTENT(OUT)` - il parametro formale è *indefinito* all'atto della chiamata di procedura, ed è la procedura stessa che deve assegnargli un valore prima del ritorno all'unità chiamante.
- `INTENT(INOUT)` - il parametro formale *ha* un valore iniziale che sarà aggiornato durante l'esecuzione della procedura.

A chiarimento di quanto detto si può considerare il seguente esempio:

```
SUBROUTINE invert(a,inverse,count)
   REAL, INTENT(IN) :: a
   REAL, INTENT(OUT) :: inverse
   INTEGER, INTENT(INOUT) :: count
      inverse = 1./a
      count = count+1
END SUBROUTINE invert
```

La subroutine `invert` ha tre argomenti formali: `a` è una variabile di *input* per cui il suo valore è da ritenersi noto all'atto della chiamata di procedura ed inoltre non potrà essere modificato nel corso dell'esecuzione della procedura, pertanto essa ha attributo `INTENT(IN)`; la variabile `inverse` è di sola *uscita* per cui essa assumerà un valore soltanto a mezzo dell'esecuzione della procedura, pertanto ha attributo `INTENT(OUT)`; infine, `count` (che "conta" il numero di volte in cui la subroutine è stata chiamata) è una variabile di *input/output* in quanto ha già un valore all'atto della chiamata di procedura e questo valore viene incrementato nel corso dell'esecuzione della procedura stessa: il suo attributo deve essere, pertanto, `INTENT(INOUT)`.

5.8.2 Parametri formali riferiti per nome

Una caratteristica peculiare del Fortran 90 è che la corrispondenza fra *parametri formali* e *argomenti attuali* può avvenire non soltanto *per posizione* ma anche *per nome*. Questa caratteristica può essere efficacemente usata allo scopo di migliorare la leggibilità di una chiamata di procedura, in special modo quando il riferimento è verso una procedura *intrinseca*. Ad esempio, data la seguente procedura:

```
SUBROUTINE sub(a,b,c)
   INTEGER, INTENT(IN) :: a, b
   INTEGER, INTENT(INOUT) :: c
   ...
END SUBROUTINE sub
```

essa può essere invocata con una qualsiasi delle seguenti istruzioni, tutte perfettamente legali:

```
CALL sub(a=1,b=2,c=x)
CALL sub(1,c=x,b=2)
CALL sub(1,2,3)
```

I nomi dei parametri formali agiscono come delle *parole chiave* nelle istruzioni di chiamata di procedura. Pertanto, l'ordine degli argomenti può essere alterato purché tutte le corrispondenze *per nome* seguano quelle (ordinate) *per posizione*. Sono ad esempio, non valide le seguenti istruzioni di chiamata:

```
CALL sub(a=1,b=2,3)  ! illegale: c non è riferito ''per nome''
CALL sub(1,c=x,2)    ! illegale: c non è riportato per ultimo
```

Si noti che quando si usano argomenti in qualità di parole chiave l'interfaccia deve sempre essere esplicita nell'unità di programma chiamante. Inoltre, tutti gli argomenti con attributo `INTENT(INOUT)` che vengano riferiti per nome devono essere assegnati ad una *variabile* e *non* ad un *valore*. Nell'esempio precedente, infatti, la seguente istruzione di chiamata sarebbe stata illegale:

```
CALL sub(a=1,b=2,c=3)    ! illegale: c assegnata a un valore
```

mentre è perfettamente lecito il seguente frammento:

```
INTEGER :: x=3
...
CALL sub(a=1,b=2,c=x)
```

5.8.3 Argomenti opzionali

Talvolta, durante l'esecuzione di una procedura, non tutti gli argomenti sono usati. In questi casi è preferibile definire questi parametri formali con l'*attributo* `OPTIONAL`:

tipo, [*lista_di_attributi*,] OPTIONAL :: *parametri_formali_"opzionali"*

come nel seguente esempio:

```
SUBROUTINE sub(a,b,c,d)
   INTEGER, INTENT(IN) :: a, b
   INTEGER, INTENT(IN), OPTIONAL :: c, d
   ...
END SUBROUTINE sub
```

In questo esempio i parametri `a` e `b` sono *sempre* richiesti quando la procedura `sub` viene invocata, mentre i parametri `c` e `d` sono *opzionali* per cui le seguenti chiamate di procedura sono tutte sintatticamente corrette:

```
CALL sub(a,b)
CALL sub(a,b,c)
CALL sub(a,b,c,d)
```

Per quanto concerne la *lista degli attributi*, è bene osservare che l'attributo `OPTIONAL` è compatibile soltanto con gli attributi `DIMENSION`, `EXTERNAL`, `INTENT`, `POINTER`, e `TARGET`. Si noti, inoltre, che, a meno di non usare parametri formali come *parole chiave*, l'*ordine* con cui appaiono gli argomenti è essenziale sicché, con riferimento all'esempio precedente, è impossibile chiamare la subroutine `sub` con l'argomento `d` senza l'argomento `c`:

```
CALL sub(a,b,d)     ! illegale
```

In una istruzione di chiamata di procedura, all'interno di un lista di parametri attuali, gli argomenti opzionali devono apparire dietro a tutti gli argomenti associati per posizione. Inoltre la procedura invocata richiede una *interfaccia esplicita* nell'unità di programma chiamante.

E' possibile "interrogare" una procedura per sapere se un dato parametro opzionale è *presente* o meno a mezzo della funzione intrinseca `PRESENT`, come illustrato nell'esempio seguente:

```
PROGRAM appello
  INTEGER :: b
  CALL who(1,2)  ! stampa "a presente" e "b presente"
  CALL who(1)    ! stampa "a presente"
  CALL who(b=2)  ! stampa "b presente"
  CALL who()     ! non stampa niente
  CONTAINS
     SUBROUTINE who(a,b)
        INTEGER, OPTIONAL :: a, b
           IF(PRESENT(a)) PRINT*, "a presente"
           IF(PRESENT(b)) PRINT*, "b presente"
     END SUBROUTINE who
END PROGRAM appello
```

La funzione intrinseca `PRESENT`, dunque, riceve in ingresso l'identificatore di un argomento formale (e, dunque, si applica ad una variabile di tipo qualsiasi) e restituisce in uscita il valore logico `.TRUE.` se detto parametro fa parte della lista degli argomenti effettivi, il valore logico `.FALSE.` in caso contrario.

Si osservi, ora, la seguente subroutine:

```
SUBROUTINE sub(a,b,c)
   REAL, INTENT(IN) :: a, b
   REAL, INTENT(IN), OPTIONAL :: c
   REAL :: reciproco_c
   ...
   IF(PRESENT(c)) THEN
      reciproco_c = 1./c
   ELSE
      reciproco_c = 0.
   END IF
   ...
END SUBROUTINE sub
```

In questo esempio, nel caso in cui l'argomento opzionale `c` sia presente, la funzione `PRESENT` fornirà valore `.TRUE.` e potrà essere calcolato il reciproco di `c`. In caso contrario, l'operazione di assegnazione permetterà di evitare una probabile divisione per zero che provocherebbe l'inevitabile interruzione del programma.

Oltre all'*attributo* `OPTIONAL`, è possibile fare anche fare uso dell'*istruzione* `OPTIONAL`, la cui sintassi è:

```
tipo ::  parametri_formali
OPTIONAL [::]  parametri_formali_"opzionali"
```

(si noti l'uso facoltativo dei due punti). Un esempio di utilizzo dell'istruzione `OPTIONAL` è fornito dalla seguente subroutine:

```
SUBROUTINE somma(a,b,c,d,sum)
   REAL, INTENT(IN) :: a, b, c, d
   REAL, INTENT(OUT) :: sum
   OPTIONAL :: c
      IF (PRESENT(c)) THEN
         sum = a+b+c+d
      ELSE
         sum = a+b+d
      END IF
END SUBROUTINE
```

Nel caso in cui il parametro opzionale sia uno degli argomenti di una procedura esterna è necessario che l'interfaccia sia esplicita nel'unità chiamante. Così, ad esempio, data la funzione esterna:

```
FUNCTION series(x,lower,upper)
   IMPLICIT NONE
   INTEGER :: i,istart,upper
   REAL :: series,x
   INTEGER, OPTIONAL :: lower
      IF (PRESENT(lower)) THEN
         istart = lower
      ELSE
         istart = 1
      END IF
      series = 0.0
      DO i = istart, upper
         series = series + x**i
      END DO
END FUNCTION series
```

che valuta la sommatoria $\sum_{i=n}^{m} x^i$, l'unità chiamante dovrà, evidentemente, contenere il seguente *interface block*:

```
FUNCTION series(x,lower,upper)
   IMPLICIT NONE
   INTEGER :: i,istart,upper
   REAL :: series,x
   INTEGER, OPTIONAL :: lower
END FUNCTION series
```

Per una procedura così definita, istruzioni del tipo:

```
series(0.5,lower=1,upper=10)
series(0.5,upper=10)
```

sono validi esempi di istruzioni chiamanti.

5.8.4 Array come parametri formali

Come è noto, un argomento viene passato a una subroutine passando un puntatore che identifica la locazione di memoria dell'argomento. Se l'argomento è un *array*, il puntatore identifica il primo elemento dell'array. Tuttavia la procedura chiamata, per poter svolgere operazioni con l'array, necessita di conoscerne le dimensioni. Una tecnica potrebbe consistere nel passare i limiti di ogni dimensione dell'array sottoforma di argomenti nell'istruzione di chiamata, e di dichiarare il corrispondente array fittizio con le stesse dimensioni. L'array facente parte della lista di argomenti formali sarà, pertanto, un *array fittizio di forma esplicita* (*explicit-shape array*) in quanto ogni suo limite è specificato in forma esplicita. In tal caso la procedura conosce i limiti dell'array e il compilatore è in grado di identificare gli eventuali riferimenti alle locazioni di memoria che superano questi confini. Inoltre, è possibile in questo caso utilizzare *sezioni di array* o l'*intero array* per svolgere operazioni globali. Ad esempio, la seguente subroutine dichiara due array, `vet1` e `vet2`, *di forma esplicita*, di estensione `n`, e ne elabora `nval` elementi:

```
SUBROUTINE elabora(vet1,vet2,n,nval)
   IMPLICIT NONE
   INTEGER, INTENT(IN) n, nval
   REAL, DIMENSION(n), INTENT(IN) :: vet1
   REAL, DIMENSION(n), INTENT(OUT) :: vet2
      vet2 = 3.*vet1
      vet2(1:nval) = 1.
END SUBROUTINE elabora
```

Si noti che l'uso degli array fittizi di forma esplicita *non* richiede un'interfaccia esplicita. Questa tecnica è, tuttavia, poco pratica e trova applicazione solo occasionalmente.

Esiste un'altra tecnica, più flessibile ed elegante, per passare un array ad una procedura. Quando una procedura ha un'*interfaccia esplicita*, tutti i dettagli sul tipo, l'ordine, lo scopo ed il rango degli argomenti fittizi della procedura sono noti all'unità chiamante e, allo stesso tempo, la procedura conosce anche tutti i dettagli sugli argomenti effettivi dell'unità chiamante. Di conseguenza, la procedura può conoscere la *forma* e la *dimensione* degli array effettivi che le

vengono passati e può utilizzare queste informazioni per manipolare gli array. Dal momento che queste informazioni sono già note attraverso l'interfaccia esplicita, non c'è bisogno di dichiarare ogni argomento array come un array fittizio di forma esplicita con i limiti dell'array passati come argomenti effettivi. Piuttosto, gli array possono essere dichiarati come *array fittizi di forma presunta* (*assumed-shape array*). Un array fittizio di forma presunta è un tipo speciale di argomento fittizio che *presume* la forma di un argomento effettivo quando la procedura viene invocata. Gli array di forma presunta vengono dichiarati con un rango e un tipo specifici, ma con un simbolo di due punti (":") al posto dei limiti, per ciascuna dimensione. Ad esempio, nella seguente istruzione di dichiarazione l'array `mat` è un array fittizio di forma presunta:

```
SUBROUTINE sub(mat)
   REAL, INTENT(IN), DIMENSION(:,:) :: mat
   ...
END SUBROUTINE sub
```

Con gli array fittizi di forma presunta è possibile sia eseguire *operazioni globali* sia utilizzare *sezioni di array* o *funzioni intrinseche di array*. Se necessario, le dimensioni e le estensioni effettive di un array di forma presunta possono essere determinate a mezzo delle comuni funzioni intrinseche di interrogazione. Ciò che, invece, *non* può essere determinato sono i limiti inferiore e superiore di ogni dimensione, in quanto alla procedure viene passato soltanto la *forma* dell'array effettivo, non i suoi *limiti*. Se, per qualche ragione, si avesse bisogno di conoscere i *limiti effettivi* allora sarà necessario lavorare con gli array di forma esplicita.

Si sottolinea che ad una procedura che definisca un array fittizio di forma presunta è possibile passare anche una *sezione di array*, purché definita da una *tripletta di indici*, In altre parole, *non* è consentito che l'argomento effettivo corrispondente a un array fittizio di forma presunta sia una sezione di array definita con *indice vettoriale*.

Nel seguente esempio viene illustrato un possibile esempio di utilizzo di array fittizi di forma presunta:

```
PROGRAM test_array_presunto
  IMPLICIT NONE
  REAL, DIMENSION(50) :: x
  REAL, DIMENSION(50,50) :: y
  ...
  CALL sub(x,y)
  CALL sub(x(1:49:2),y(2:4,4:4))
  ...
  CONTAINS
     SUBROUTINE sub(a,b)
        REAL, INTENT(IN) :: a(:), b(:,:)
        ...
     END SUBROUTINE sub
END PROGRAM test_array_presunto
```

(si rammenti che le procedure interne rispettano automaticamente il vincolo dell'interfaccia esplicita).

5.8.5 Variabili stringa di caratteri come parametri formali

Quando una *stringa di caratteri* viene utilizzata come argomento fittizio di una procedura la sua lunghezza viene dichiarata con un asterisco. Poiché la memoria non viene allocata per gli argomenti fittizi, non è necessario conoscere la lunghezza di un argomento stringa quando la procedura viene compilata. Ecco un tipico esempio di argomento fittizio stringa di caratteri:

```
PROGRAM main
  CHARACTER(LEN=10) :: nome
  ...
  CALL sub(nome)
  ...
  CONTAINS
     SUBROUTINE sub(mia_stringa)
        CHARACTER(LEN=*), INTENT(IN) :: mia_stringa
        ...
     END SUBROUTINE sub
END PROGRAM main
```

All'atto della chiamata di procedura, la lunghezza dell'argomento fittizio stringa sarà pari alla lunghezza dell'argomento effettivo che verrà passato dall'unità chiamante. Per conoscere la lunghezza dell'argomento durante l'esecuzione della subroutine è possibile usare la funzione intrinseca `LEN`. Ad esempio, la seguente subroutine visualizza la lunghezza dell'argomento stringa che riceve in input dall'unità chiamante:

```
SUBROUTINE sub(mia_stringa)
   CHARACTER(LEN=*), INTENT(IN) :: mia_stringa
   WRITE(*,'(1X,A,I3)') "Lunghezza della variabile = ", LEN(mia_stringa)
   ...
END SUBROUTINE sub
```

Un parametro formale come `mia_stringa` viene detto *stringa fittizia di lunghezza presunta* (*assumed-length character*). Secondo un approccio molto meno flessibile sarebbe possibile anche dichiarare un argomento formale come una *stringa fittizia di lunghezza esplicita* (*explicit-length character*), come nel seguente esempio:

```
PROGRAM main
  CHARACTER(LEN=10) :: nome
  ...
  CALL sub(nome)
  ...
  CONTAINS
     SUBROUTINE sub(stringa_fissa)
        CHARACTER(LEN=15), INTENT(IN) :: stringa_fissa
        ...
     END SUBROUTINE sub
END PROGRAM main
```

ma di solito non c'è alcuna ragione per preferire questa tecnica così "rigida" alla più versatile modalità basata sull'uso delle stringhe di lunghezza presunta, tanto più che l'uso delle stringhe fittizie di lunghezza esplicita presenta l'ulteriore limitazione che la lunghezza della stringa (esplicitamente definita) deve essere necessariamente maggiore o uguale di quella della stringa effettiva.

5.8.6 Tipi di dati derivati come argomenti di procedure

Gli argomenti di una procedura possono essere variabili di un *tipo di dati derivato* purché la *visibilità* di questo tipo di dati sia non ambigua all'atto della chiamata di procedura. Ciò può avvenire in uno dei seguenti modi:

- La procedura è *interna* all'unità di programma chiamante in cui il tipo di dati è definito.
- Il tipo di dati derivato è definito in un modulo che sia accessibile alla procedura chiamata.

L'esempio seguente mostra un utilizzo di dati di tipo derivato come argomento di una procedura. L'esempio, inoltre, mostra un esempio di utilizzo comparato di procedure *interne* e *di modulo*:

```
MODULE miomodulo
! Sezione dichiarativa
   IMPLICIT NONE
   TYPE :: new
      CHARACTER(LEN=3) :: str
      INTEGER :: j, k
   END TYPE new
! Inizializzazione di x a mezzo di costruttore:
   TYPE(new) :: x=new("abc",1234,5678)
! Sezione "interna"
CONTAINS
   SUBROUTINE sub_mod_1()       ! sottoprogramma di modulo
! start subroutine sub_mod_1
      CALL sub_int()
      RETURN
   CONTAINS
      SUBROUTINE sub_int()      ! sottoprogramma interno
! start subroutine sub_int_1
         CALL sub_mod_2(x)
         RETURN
      END SUBROUTINE sub_int
   END SUBROUTINE sub_mod_1
   SUBROUTINE sub_mod_2(x)      ! sottoprogramma di modulo
      TYPE(new), INTENT (IN OUT) :: x
      INTEGER :: int1, int2
! start subroutine sub_mod_2
```

```
            WRITE(*,*) x%str
            int1 = x%j
            int2 = x%k
            x = new("def",11,22)
         RETURN
      END SUBROUTINE sub_mod_2
   END MODULE miomodulo
   !
   PROGRAM prova
      USE miomodulo
      CALL sub_mod_1( )
      WRITE(*,*) x
      STOP
   END PROGRAM prova
```

Si può banalmente verificare che l'output di questo programma è:

```
 abc
 def          11          22
```

Un esempio un pò più interessante potrebbe essere quello di definire un tipo di dati `grid2D` che raggruppi alcune informazioni (ad esempio estremi del dominio computazionale e numero di nodi lungo gli assi) relative ad una griglia bidimensionale per la discretizzazione di un problema risolto numericamente alle differenze finite:

```
TYPE grid2D
! N.B.: xmin, ymin, xmax e ymax definiscono un rettangolo
   REAL :: xmin, ymin
   REAL :: xmax, ymax
! nx ed ny sono le variabili di partizione
   INTEGER :: nx, ny
END TYPE grid2D
```

ed un set di funzioni che lavorino proprio su dati di tipo `grid2D` e che forniscano ad esempio la spaziatura della griglia lungo gli assi x ed y:

```
REAL FUNCTION dx(g)
   TYPE(grid2D) :: g
      dx = (g%xmax-g%xmin)/g%nx
      RETURN
END FUNCTION dx

REAL FUNCTION dy(g)
   TYPE(grid2D) :: g
      dy = (g%ymax-g%ymin)/g%ny
      RETURN
END FUNCTION dx
```

Un semplice esempio di applicazione delle procedure testé sviluppate potrebbe essere il seguente:

```
PROGRAM m
  TYPE(grid2D) :: g
  ...
  g%xmin = 0.0
  g%xmax = 10.0
  WRITE(*,*) "dx = " dx(g)
  ...
END PROGRAM m
```

5.8.7 Procedure come argomenti

E' possibile usare una procedura come un argomento attuale in un'istruzione di chiamata ad un'altra procedura. Una procedura può essere usata come un argomento di scambio solo se è una *procedura esterna* oppure una *procedura di modulo*: pertanto *non* è consentito usare come argomento una *procedura interna*. Un esempio assai comune è il caso in cui un argomento attuale sia il risultato di una funzione:

```
PROGRAM test_func
   INTERFACE
      FUNCTION func(x)
         REAL, INTENT(IN) :: x
      END FUNCTION func
   END INTERFACE
   ...
   CALL sub(a,b,func(3))
   ...
END PROGRAM test_func

REAL FUNCTION func(x)    ! N.B.: funzione ''esterna''
   REAL, INTENT(IN) :: x
   func = x**2+1.0
END FUNCTION func
```

Quando la subroutine `sub` viene invocata, ad essa vengono passati i *valori* dei parametri `a` e `b`, ed il risultato della funzione `func` (nel caso in esame pari a `10`). Pertanto, una procedura che sia usata come argomento verrà sempre eseguita *prima* di essere passata alla procedura chiamata. Come si è potuto evincere dall'esempio precedente, quando una procedura viene usata come argomento di un'altra procedura è necessario che abbia una *interfaccia esplicita* nell'unità chiamante. Naturalmente l'interfaccia è, come noto, da considerarsi esplicita *per default* quando si lavori con procedure di modulo.

Un'altra possibilità che il Fortran mette a disposizione per utilizzare una procedura come argomento è quella di definire la procedura argomento con l'*attributo* `EXTERNAL`. Ad esempio,

nel caso in cui il parametro formale sia una function, la dichiarazione del tipo della funzione dovrà includere l'attributo `EXTERNAL`, al modo seguente:

```
tipo, EXTERNAL ::  nome_function
```

Questa dichiarazione serve a specificare che il parametro formale è il nome di una funzione e non il nome di una variabile. Anche il corrispondente argomento attuale dovrà essere dichiarato, nell'unità di programma chiamante, con l'attributo `EXTERNAL` o, se è il nome di una *funzione intrinseca*, con l'attributo `INTRINSIC`:

```
tipo, EXTERNAL ::  nome_function
tipo, INTRINSIC ::  nome_function_intrinseca
```

Le seguenti dichiarazioni forniscono un esempio di quanto detto:

```
INTEGER, EXTERNAL :: func
REAL, INTRINSIC :: sin
```

Si noti che non tutte le funzioni intrinseche possono essere passate come parametri attuali in una chiamata di procedura. In particolare è illegale l'uso delle funzioni di conversione di tipo (`REAL`, `DBLE`, ...).

Il seguente codice di calcolo mostra un esempio di funzione di modulo usata come parametro attuale in una chiamata di procedura. Lo scopo del programma è quello di valutare l'integrale della suddetta funzione, `f(x)`, fra due estremi `a` e `b` con il *metodo del trapezio* secondo un approccio di tipo iterativo.

```
MODULE function_mod
   IMPLICIT NONE
CONTAINS
   FUNCTION f(x) RESULT(f_result)
      IMPLICIT NONE
         REAL, INTENT (IN) :: x
         REAL :: f_result
            f_result = EXP(-x**2)  ! Funzione integranda
   END FUNCTION f
END MODULE function_mod

MODULE integral_mod
   IMPLICIT NONE
CONTAINS
   RECURSIVE FUNCTION integral(f,a,b,epsilon) RESULT(integral_result)
      IMPLICIT NONE
      REAL, INTENT (IN) :: a, b, epsilon
      REAL :: integral_result, f
      EXTERNAL f
      REAL :: h, mid
```

```
      REAL :: one_trapezoid_area, two_trapezoid_area
      REAL :: left_area, right_area
         h = b-a
         mid = (a+b)/2.0
         one_trapezoid_area = h*(f(a)+f(b))/2.0
         two_trapezoid_area = h/2.0*(f(a)+f(mid))/2.0 +
                              h/2.0*(f(mid)+f(b))/2.0
         IF (ABS(one_trapezoid_area-two_trapezoid_area)  &
                <3.0*epsilon) THEN
            integral_result = two_trapezoid_area
         ELSE
            left_area = integral(f,a,mid,epsilon/2.0)
            right_area = integral(f,mid,b,epsilon/2.0)
            integral_result = left_area+right_area
         END IF
   END FUNCTION integral
END MODULE integral_mod

PROGRAM integrate
   USE function_mod
   USE integral_mod
   IMPLICIT NONE
      REAL :: x_min, x_max
      REAL :: answer, pi
         pi = 4.0*ATAN(1.0)
         x_min = -4.0
         x_max = 4.0
         answer = integral (f,x_min,x_max,0.01)
         PRINT "(A,F11.6)",  &
               "L'integrale vale approssimativamente: ",answer
         PRINT "(A,F11.6)",  &
               "La soluzione esatta e':               ",SQRT(pi)
END PROGRAM integrate
```

L'output del precedente programma è:

```
L'integrale vale approssimativamente:    1.777074
La soluzione esatta e':                  1.772454
```

Nel caso in cui la procedura da passare sia una *subroutine*, dal momento che al suo nome non è associato alcun *tipo*, l'approccio da seguire è lievemente diverso. In particolare, sia nell'unità chiamante che nella procedura invocata la subroutine dovrà essere dichiarata in una speciale *istruzione* EXTERNAL:

```
EXTERNAL nome_subroutine
```

Si noti come l'istruzione `EXTERNAL` *non* preveda i due punti ("::"). E' bene osservare che un *interface block non* può essere usato per una procedura specificata in una istruzione `EXTERNAL` (e viceversa). Si ricordi, infine, che *non* è permesso in Fortran usare *subroutine intrinseche* come parametri attuali in una chiamata di procedura.

Si riporta, a titolo di riepilogo, un ulteriore esempio che prevede l'impiego simultaneo dell'attributo `INTRINSIC` e dell'attributo e dell'istruzione `EXTERNAL`.

```
REAL FUNCTION reciproc(reala)
   IMPLICIT NONE
   REAL, INTENT(IN) :: reala
      reciproc = 1/reala
END FUNCTION reciproc

SUBROUTINE add_nums(real1,real2,result)
   IMPLICIT NONE
   REAL, INTENT(IN)  :: real1, real2
   REAL, INTENT(OUT) :: result
      result = real1+real2
END SUBROUTINE add_nums

SUBROUTINE do_math(arg1,arg2,math_fun,math_sub,result)
   IMPLICIT NONE
! Parametri formali
   REAL, INTENT(IN) :: arg1, arg2  ! variabili argomento
   REAL, EXTERNAL :: math_fun      ! funzione argomento
   EXTERNAL math_sub               ! subroutine argomento
   REAL, INTENT(OUT) :: result     ! variabile risultato
! Variabili locali
   REAL :: temp1, temp2
      temp1 = math_fun(arg1)
      temp2 = math_fun(arg2)
      CALL math_sub(temp1,temp2,result)
END SUBROUTINE

PROGRAM proc_pass
! Sezione dichiarativa
  IMPLICIT NONE
  REAL, EXTERNAL :: reciproc    ! "Actual FUNCTION"
  REAL, INTRINSIC :: SIN    ! "Actual INTRINSIC FUNCTION"
  EXTERNAL add_nums         ! "Actual SUBROUTINE"
! Dichiarazione di variabili
  REAL :: num1, num2, reciproc_ans, sin_ans
! Sezione esecutiva
  READ*, num1, num2
```

```
  CALL do_math(num1,num2,reciproc,add_nums,reciproc_ans)
! Ora la variabile reciproc_ans contiene la somma dei
! reciproci di num1 e di num2
  CALL do_math(num1,num2,SIN,add_nums,sin_ans)
! Ora la variabile reciproc_ans contiene la somma dei
! "seni" di num1 e di num2
  PRINT*, reciproc_ans, sin_ans
END PROGRAM proc_pass
```

5.9 Funzioni array

Una function può fornire non soltanto un valore scalare ma anche un puntatore, un array o una variabile di un tipo derivato. Per le funzioni che restituiscono un *array* le dimensioni del risultato possono essere determinate allo stesso modo con cui viene dichiarato un *array automatico*, come si avrà modo di capire nel prossimo capitolo.

La seguenti funzioni rappresentano semplici esempio di *array valued function*:

```
FUNCTION prod(a,b)
  IMPLICIT NONE
  REAL, DIMENSION(:), INTENT(IN) :: a, b
  REAL DIMENSION(SIZE(a)) :: prod
  prod = a * b
END FUNCTION prod

FUNCTION Partial_sum(p)
   IMPLICIT NONE
   REAL     :: p(:)                  ! Assumed-shape dummy array
   REAL     :: Partial_sum(size(p)) !
   INTEGER ::  k
      Partial_sum = (/(sum(p(1:k),k=1,size(p))/)
! Si sarebbe potuto ottenere lo stesso risultato
! anche con il seguente ciclo for:
! DO k=1,size(P)
!    Partial_sum(k) = sum(P(1:k))
! END DO
! tuttavia il DO-loop specifica un insieme di operazioni
! sequenziali invece di operazioni parallele
      RETURN
END FUNCTION Partial_sum
```

Il seguente programma mostra, in più, anche un esempio di programma principale che utilizza di *funzione array*:

```
PROGRAM test_array_func
  IMPLICIT NONE
  INTEGER, PARAMETER :: m=6
  INTEGER, DIMENSION (m,m) :: im1, im2
  ... ! assegnazione di m
  im2 = array_func(im1,1)
  ...
  CONTAINS
     FUNCTION array_func(x,cost)
        INTEGER, INTENT(IN) :: x(:,:)
        INTEGER, INTENT(IN) :: cost
        INTEGER :: array_func(SIZE(x,1),SIZE(x,2))
           array_func(:,:) = cost*x(:,:)
     END FUNCTION array_func
END PROGRAM test_array_func
```

Certamente in questi esempi si sarebbe potuto dichiarare il risultato anche con dimensioni fisse (ossia come un *explicit shape array*) ma questo approccio sarebbe stato molto meno flessibile.

Si noti che per default le funzioni sono sempre pensate produrre un risultato scalare per cui in tutti gli altri casi deve essere fornita una interfaccia esplicita nell'unità di programma chiamante. Questo naturalmente è vero non soltanto per le funzioni che restituiscono un array ma anche per tutte quelle funzioni che restituiscono puntatori oppure oggetti di tipo derivato. Mentre per le procedure interne o di modulo non sussiste alcun problema, per le procedure esterne deve essere previsto un opportuno *interface block*. Di seguito viene riportato lo stesso esempio di prima, questa volta però con l'ausilio di una funzione esterna:

```
PROGRAM test_array_func
  IMPLICIT NONE
  INTERFACE   ! obbligatorio
     FUNCTION array_func(ima,scal)
        INTEGER, INTENT(IN) :: ima(:,:)
        INTEGER, INTENT(IN) :: scal
        INTEGER, DIMENSION(SIZE(ima,1),SIZE(ima,2)) :: array_func
     END FUNCTION array_func
  END INTERFACE
  INTEGER, PARAMETER      :: m = 6
  INTEGER, DIMENSION(m,m) :: im1, im2
  ...
  im2 = array_func(im1,1)
  ...
END PROGRAM test_array_func

FUNCTION array_func(x,cost)
   IMPLICIT NONE
```

```
   INTEGER, INTENT(IN) :: x(:,:)
   INTEGER, INTENT(IN) :: cost
   INTEGER, DIMENSION(SIZE(x,1),SIZE(x,2)) :: array_func
      array_func(:,:) = x(:,:)*cost
END FUNCTION array_func
```

5.10 Funzioni stringa di caratteri

In molti casi può essere utile definire una function che restituisca una *stringa di caratteri* di una data lunghezza. Detta lunghezza può, indifferentemente, essere fissa oppure dipendere da uno dei parametri formali. Per chiarire quanto detto, si guardi il seguente esempio in cui la function `reverse` acquisisce una stringa in input e la restituisce nella sua forma "invertita":

```
FUNCTION reverse(parola)
   IMPLICIT NONE
   CHARACTER(LEN=*), INTENT(IN) :: parola
   CHARACTER(LEN=LEN(parola)) :: reverse
   INTEGER :: l, i
      l = LEN(parola)
      DO i=1,l
         reverse(l-1+1:l-i+1) = parola(i:i)
      END DO
END FUNCTION reverse
```

Si noti che la stringa `parola` non può essere usata fino a che non venga dichiarata: è per questa ragione che il tipo della funzione non può apparire come prefisso nella dichiarazione di funzione. In questo caso la lunghezza del risultato della funzione è determinata automaticamente pari a quella del parametro formale `parola`.

Un esempio lievemente più complesso è il seguente:

```
PROGRAM stringhe
   IMPLICIT NONE
   INTEGER, PARAMETER :: l=7
   CHARACTER(LEN=l) :: s1 = "Antonio"
! start
   WRITE(*,*) func(2*l,s1)
   STOP
CONTAINS
   FUNCTION func(n,b)
      INTEGER, PARAMETER :: k=3
      CHARACTER(LEN=*), INTENT(IN) :: b  ! stringa di lunghezza fittizia
      INTEGER, INTENT(IN) :: n
      CHARACTER(LEN=2*LEN(b)) :: func ! stringa di lung. automatica
      CHARACTER(LEN=LEN(b)) :: y      ! stringa locale di lung. automatica
```

```
      CHARACTER(LEN=k) :: x = "Aul"   ! stringa locale di lunghezza fissa
      CHARACTER(LEN=*), PARAMETER :: word = "stringhetta"
! start function func
         func = b//b      ! func="AntonioRaucci"
         y = word(k+2:)   ! y="nghetta"
         y(:k) = x        ! y="Raucci"
         func(n/2+1:) = y ! func="AntonioRaucci"
         RETURN
   END FUNCTION func
END PROGRAM stringhe
```

5.11 Funzioni di tipo derivato

Una function può restituire anche il valore di un *tipo di dati derivato*. Ad esempio, costruito il tipo `complesso`, al modo seguente:

```
TYPE complesso
   REAL :: p_real
   REAL :: p_imm
END TYPE complesso
```

la seguente funzione implementa l'algoritmo di moltiplicazione di due valori di tipo `complesso`:

```
TYPE(complesso) FUNCTION prod_comp(a,b)
   TYPE(complesso), INTENT(IN) a, b
      prod_comp%p_real = a%p_real*b%p_real - a%p_imm*b%p_imm
      prod_comp%p_imm = a%p_real*b%p_imm + a%p_imm*b%p_real
END FUNCTION prod_comp
```

Naturalmente, nell'esempio precedente il tipo derivato `complesso` è stato definito per motivi puramente "didattici", visto che in Fortran è già disponibile il tipo predefinito `COMPLEX`.

Un esempio un pò più interessante è fornito dal seguente modulo in cui vengono definiti sia un tipo di dati (`money`), sia una coppia di funzioni che operano su di esso:

```
MODULE moneytype

   IMPLICIT NONE
   TYPE money
      INTEGER :: euro, cent
   END TYPE money

CONTAINS

   FUNCTION addmoney(a,b)
```

```
      IMPLICIT NONE
      TYPE (money) :: addmoney
      TYPE (money), INTENT(IN) :: a, b
      INTEGER :: carry, tempcent
         tempcent = a%cent + b%cent
         carry = 0
         IF (tempcent>100) THEN
            tempcent = tempcent - 100
            carry = 1
         END IF
         addmoney%euro = a%euro + b%euro + carry
         addmoney%cent = tempcent
   END FUNCTION addmoney

   FUNCTION subtractmoney(a,b)
      IMPLICIT NONE
      TYPE (money) :: subtractmoney
      TYPE (money), INTENT(IN) :: a, b
      INTEGER :: euro, tempcent, carry
         tempcent = a%cent - b%cent
         euro = a%euro - b%euro
         IF ((tempcent<0).AND.(euro>0)) THEN
            tempcent = 100 + tempcent
            euro = euro - 1
         ELSE IF ((tempcent>0).AND.(euro<0)) THEN
            tempcent = tempcent - 100
            euro = euro + 1
         END IF
         subtractmoney%cent = tempcent
         subtractmoney%euro = euro
   END FUNCTION subtractmoney

END MODULE moneytype
```

Chiaramente il tipo di dati `money` rappresenta nient'altro che una somma di denaro (espressa in `euro` e `cent`) con la quale si possono effettuare le comuni operazioni di somma e sottrazione previste da una qualsiasi gestione di un budget. Un programma scritto *ad hoc* per testare l'efficacia di queste funzioni di tipo derivato potrebbe essere il seguente:

```
PROGRAM prova_moneytype
  USE moneytype
  IMPLICIT NONE
  TYPE(money) :: costo1, costo2, spesa, anticipo, resto
  costo1=money(22,15)     ! 22 euro e 15 cent
```

```
  costo2=money(15,25)     ! 15 euro e 25 cent
  anticipo=money(50,0)    ! 50 euro
  spesa=addmoney(costo1,costo2)
  resto=subtractmoney(anticipo,spesa)
  WRITE(*,100) "Costo del primo articolo:   ", costo1
  WRITE(*,110) "Costo del secondo articolo: ", costo2
  WRITE(*,120)
  WRITE(*,130) "Spesa totale:               ", spesa
  WRITE(*,140) "Contante versato:           ", anticipo
  WRITE(*,150)
  WRITE(*,160) "Resto ricevuto:             ", resto
100 FORMAT(1X,A28,1X,I3,1X,"Euro",1X,I3,1X,"Cent",1X,"+",/)
110 FORMAT(1X,A28,1X,I3,1X,"Euro",1X,I3,1X,"Cent",1X,"=",/)
120 FORMAT(30X,20("-"),/)
130 FORMAT(1X,A26,1X,"-",1X,I3,1X,"Euro",1X,I3,1X,"Cent",1X,"+",/)
140 FORMAT(1X,A28,1X,I3,1X,"Euro",1X,I3,1X,"Cent",1X,"=",/)
150 FORMAT(30X,20("-"),/)
160 FORMAT(1X,A28,1X,I3,1X,"Euro",1X,I3,1X,"Cent")

END PROGRAM prova_moneytype
```

Il risultato prodotto da questo programma è, chiaramente, il seguente:

```
Costo del primo articolo:     22 Euro  15 Cent +

Costo del secondo articolo:   15 Euro  25 Cent =

                             --------------------

Spesa totale:              -  37 Euro  40 Cent +

Contante versato:             50 Euro   0 Cent =

                             --------------------

Resto ricevuto:               12 Euro  60 Cent
```

5.12 Procedure intrinseche

Il Fortran 90 standard comprende 113 *procedure intrinseche* progettate allo scopo di risolvere in maniera efficiente problemi specifici. Esse possono essere raggruppate come:

- *Procedure di elemento*, a loro volta ripartite in:

  - *Procedure matematiche*, ad esempio: `SIN` e `LOG`.
  - *Procedure numeriche*, ad esempio: `SUM` e `CEILING`.
  - *Procedure per il trattamento di caratteri*, ad esempio: `INDEX` e `TRIM`.
  - *Procedure di manipolazione di bit*, ad esempio: `IAND` e `IOR`.

- *Procedure di interrogazione*, ad esempio: `ALLOCATED` e `SIZE`. Tipicamente una "interrogazione" riguarda:
  - Lo *stato* di un oggetto dinamico.
  - La *forma*, le *dimensioni* ed i *limiti* di un array.
  - I *parametri di kind* di un oggetto.
  - Il modello numerico usato per la *rappresentazione dei tipi* e dei *parametri di kind*.
  - La presenza di un argomento *opzionale* in una chiamata di procedura.

- *Procedure di trasformazione*, ad esempio: `REAL` e `TRANSPOSE`. Queste possono essere, a loro volta:
  - *Procedure di ripetizione*, ad esempio per la "ripetizione" di stringhe di caratteri.
  - *Procedure di riduzione matematica*, ad esempio per ottenere un array di rango minore a partire da un dato array.
  - *Procedure di manipolazione di array*, ad esempio per le operazioni di *shift*, come `RESHAPE` e `PACK`.
  - *Procedure di coercizione di tipo*, ad esempio `TRANSFER` che copia un oggetto, *bit per bit*, in uno di tipo differente.

- *Procedure "varie"* (subroutine *non* di elemento), come `SYSTEM_CLOCK` e `DATE_AND_TIME`.

Le procedure intrinseche si differenziano a seconda degli argomenti sui quali operano. Alcune procedure possono lavorare solo con variabili scalari, altre soltanto con array. Tutte le procedure intrinseche che si applicano ad argomenti di tipo `REAL` in *singola precisione* accettano anche argomenti in *doppia precisione*.

Delle procedure matematiche, numeriche e di manipolazione di stringhe si è già parlato al capitolo 1, così come di alcune funzioni di interrogazione. Delle altre funzioni di interrogazione e delle procedure di manipolazione di array si parlerà nei prossimi capitoli. Restano da analizzare le *subroutine intrinseche* e le procedure operanti sui bit e sarà questo lo scopo dei prossimi paragrafi.

5.12.1 Subroutine intrinseche

Fanno parte di questo gruppo delle utility per la generazione di numeri pseudo-random e per la interrogazione dell'orologio di sistema.

`DATE_AND_TIME([DATE],[TIME],[ZONE],[VALUES])`

Questa subroutine restituisce la data e il tempo correnti così come forniti dall'orologio interno del sistema. I suoi argomenti, tutti opzionali e tutti con attributo `INTENT(OUT)`, sono:

- `DATE`: stringa di (almeno) otto caratteri; gli otto caratteri più a sinistra sono posti pari a `CCYYMMDD` con il significato di *century/year/month/day*.
- `TIME`: stringa di (almeno) dieci caratteri; i dieci caratteri più a sinistra sono posti pari a `HHMMSS.SSS` con il significato di *hour/minute/second/millisecond*.
- `ZONE`: stringa di (almeno) cinque caratteri; i cinque caratteri più a sinistra sono posti pari a `+HHMM`, dove `HH` ed `MM` rappresentano le ore ed i minuti di differenza rispetto al tempo medio di Greenwich (anche detto UTC, *Coordinate Universal Time*).
- `VALUES`: array di tipo `INTEGER` di lunghezza (minima) pari ad otto. Gli elementi di `VALUES` hanno il seguente significato:

  `value(1)` anno corrente
  `value(2)` mese corrente
  `value(3)` giorno corrente
  `value(4)` differenza, in minuti, rispetto all'UTC (`0-59`)
  `value(5)` ora (`0-23`)
  `value(6)` minuti (`0-59`)
  `value(7)` secondi (`0-59`)
  `value(8)` millisecondi (`0-999`)

Se qualcuna di queste informazioni non è disponibile, il corrispondente valore sarà posto pari a `-HUGE(0)`.

Un esempio di utilizzo della routine `DATE_AND_TIME` è il seguente:

```
CHARACTER(LEN=8)  :: d
CHARACTER(LEN=10) :: t
CHARACTER(LEN=5)  :: z
...
CALL DATE_AND_TIME(DATE=d,TIME=t,ZONE=z)
PRINT*, "Data corrente: ",d(7:8),"/",d(5:6),"/",d(1:4)
PRINT*, "Ora corrente:  ",t(1:2),":",t(3:4),":",t(5:10)
PRINT*
PRINT*, "Differenza rispetto all'UTC:"
PRINT*, "Ore: ",z(2:3)," Minuti: ",z(3:4)
```

che, eseguito il giorno 8 dicembre 2001 alle ore 12:48, fornisce il seguente risultato:

```
Data corrente: 08/12/2001
Ora corrente:  12:48:12.607
```

```
Differenza rispetto all'UTC:
Ore: 01 Minuti: 10
```

`SYSTEM_CLOCK([COUNT],[COUNT_RATE],[COUNT_MAX])`

Subroutine che restituisce il tempo del sistema. Gli argomenti, aventi tutti attributo `INTENT(OUT`, sono tutti opzionali sebbene almeno uno di essi debba essere necessariamente specificato.

- `COUNT` è uno scalare di tipo `INTEGER` che restituisce un valore calcolato in base al valore corrente del clock di sistema. Questo valore viene incrementato di un'unità ad ogni ciclo del clock fino al raggiungimento del valore `COUNT_MAX`, dopodiché viene resettato a zero ed il conteggio riprende daccapo. Nel caso in cui non esistesse un orologio di sistema, il valore restituito da `COUNT` sarebbe `-HUGE(0)`.
- `COUNT_RATE` è uno scalare di tipo `INTEGER` che fornisce il numero di cicli per secondo del clock del processore. Nel caso in cui non esistesse un orologio di sistema, il valore restituito da `COUNT_RATE` sarebbe `0`.
- `COUNT_MAX` è uno scalare di tipo `INTEGER` che fornisce il massimo valore raggiungibile da `COUNT`. Nel caso in cui non esistesse un orologio di sistema, il valore restituito da `COUNT_MAX` sarebbe `0`.

Un esempio di utilizzo della procedura intrinseca `SYSTEM_CLOCK` è fornito dal seguente frammento di codice:

```
INTEGER :: ic, crate, cmax
CALL SYSTEM_CLOCK(COUNT=ic,COUNT_RATE=crate,COUNT_MAX=cmax)
PRINT*, "Ciclo corrente:   ",ic
PRINT*, "Cicli al secondo: ",crate
PRINT*, "Ciclo massimo:    ",cmax
```

un cui possibile output potrebbe essere questo:

```
Ciclo corrente:          46364
Cicli al secondo:            1
Ciclo massimo:           86399
```

`CPU_TIME(TIME)`

Si tratta di una subroutine intrinseca introdotta dal Fortran 95 che fornisce il tempo (in secondi) speso dal processore sul programma corrente, con una approssimazione dipendente dal processore. Il parametro formale `TIME` è una variabile scalare di tipo `REAL` con attributo `INTENT(OUT)`. Se la procedura non è in grado di restituire un valore significativo, alla variabile `TIME` viene assegnato un valore negativo dipendente dalla specifica implementazione.

Il seguente frammento di programma può aiutare a comprendere il funzionamento della procedura `CPU_TIME`:

```
REAL time_begin, time_end
...
CALL CPU_TIME(time_begin)
...
CALL CPU_TIME(time_end)
PRINT*, "Il tempo impiegato dalle operazioni e' stato: ", &
                    time_begin - time_end, " secondi"
```

Lo scopo principale per cui è stata introdotta questa procedura è quello di permettere di confrontare algoritmi differenti sul medesimo computer oppure di scoprire quale ramo di programma è più costoso in termini di sforzo di calcolo.

Si noti che su computer con diverse CPU, il parametro `TIME` può essere implementato come un array i cui elementi contengono i tempi di ciascun processore.

`RANDOM_NUMBER(HARVEST)`

Questa subroutine consente di generare un valore reale pseudo-random nell'intervallo [0-1]. a partire da uno *starting point* rappresentato da un vettore di interi. Il parametro formale `HARVEST` ha attributo `INTENT(OUT)` e può essere un valore scalare oppure un array di tipo `REAL`

Il valore (o la sequenza) random viene prodotta a partire da un punto di partenza (*seed*) che, se non viene imposto tramite la subroutine `RANDOM_SEED`, viene posto pari ad un valore dipendente dal processore.

Il seguente frammento di programma produce un valore ed un array di 5×5 elementi pseudo-random:

```
REAL :: x, y(5,5)
! Assegna ad x un un valore pseudo-random
CALL RANDOM_NUMBER(HARVEST=x)
! Produce un array di 5x5 valori pseudo-random
CALL RANDOM_NUMBER(y)
```

`RANDOM_SEED([SIZE],[PUT],[GET])`

Questa subroutine serve a "resettare" (o ad avere informazioni circa) il *generatore di numeri random*. I suoi parametri formali hanno il seguente significato:

- `SIZE`: è uno scalare di tipo `INTEGER` e fornisce il numero di interi che il processore utilizza come *starting point*. Ha attributo `INTENT(OUT)`.
- `PUT`: è un array di tipo `INTEGER` attraverso cui l'utente può fornire uno *starting point* al generatore di numeri random. Ha attributo `INTENT(IN)`.
- `GET`: è un array di tipo `INTEGER` che contiene il valore corrente dello *starting point*. Ha attributo `INTENT(OUT)`.

Il seguente frammento di codice fornisce un esempio di utilizzo delle subroutine di generazione di numeri casuali:

```
REAL :: x
...
CALL RANDOM_SEED()     ! inizializzazione
CALL RANDOM_NUMBER(x)  ! generazione
```

Quello che segue è un ulteriore esempio che mostra l'utilizzo degli altri parametri formali di RANDOM_SEED:

```
CALL RANDOM_SEED                ! Inizializzazione
CALL RANDOM SEED(SIZE=k)        ! Legge quanti numeri il processore
                                !   utilizza come valori iniziali
CALL RANDOM_SEED(PUT=seed(1:k)) ! Inserisce valori di partenza scelti
                                !   dall'utente
CALL RANDOM_SEED(GET=old(1:k))  ! Legge i valori iniziali ''attuali''
```

5.12.2 Procedure di manipolazione dei bit

Il Fortran 90/95 dispone di un insieme di undici procedure per la manipolazione e di una per la interrogazione dei bit presenti negli interi. Tali procedure sono basate su un modello secondo cui un intero si compone di s bit w_k (con $k = 0, 1, \ldots, s-1$), in una sequenza che procede da destra a sinistra, basata sul valore non negativo $\sum_{k=0}^{s-1} w_k \times 2^k$.

Questo modello è valido solo in relazione a queste procedure intrinseche e coincide con il già descritto modello per gli interi (cfr. capitolo 1) quando r è una potenza intera di 2 e $w_{s-1} = 0$, ma quando $w_{s-1} = 1$ i modelli non corrispondono ed il valore espresso come un intero può variare da processore a processore.

BIT_SIZE(I)

E' una funzione di interrogazione che fornisce il numero di bit nel modello assunto per gli interi aventi stesso parametro di kind di I.

Il prossimo frammento di programma mostra come sia possibile interrogare un processore per conoscere il numero di bit necessario a rappresentare un intero di assegnato parametro di kind:

```
INTEGER :: i=1
INTEGER(KIND=SELECTED_INT_KIND(2)) :: i1
INTEGER(KIND=SELECTED_INT_KIND(4)) :: i2
INTEGER(KIND=SELECTED_INT_KIND(9)) :: i3
! ...
i1=INT(i,KIND(2))
i2=INT(i,KIND(4))
i3=INT(i,KIND(9))
```

```
! ...
WRITE(*,10) "N.ro di kind = ",KIND(i1)," ---> N.ro di bit = ",BIT_SIZE(i1)
WRITE(*,10) "N.ro di kind = ",KIND(i2)," ---> N.ro di bit = ",BIT_SIZE(i2)
WRITE(*,10) "N.ro di kind = ",KIND(i3)," ---> N.ro di bit = ",BIT_SIZE(i3)
10 FORMAT(T1,A15,I2,A19,I3)
```

Eseguito su un PC con processore Pentium il frammento precedente produce il seguente output:

```
N.ro di kind =  1 ---> N.ro di bit =  8
N.ro di kind =  2 ---> N.ro di bit = 16
N.ro di kind =  4 ---> N.ro di bit = 32
```

`BTEST(I,POS)`

Fornisce il valore logico `.TRUE.` se il bit avente posizione `POS` nella rappresentazione di `I` ha valore `1`, altrimenti fornisce il valore `.FALSE.`. Entrambi gli argomenti sono di tipo `INTEGER`, in particolare deve risultare $0 \leq$`POS`$\leq$`BIT_SIZE(I)`.

Una interessante applicazione della funzione intrinseca `BTEST` è riportata di seguito. Il seguente programma, infatti, fa uso della funzione `BTEST` per "costruire" la rappresentazione binaria di un valore intero di modo che essa possa essere riprodotta a stampa evitando, in tal modo, di ricorrere al descrittore di formato `B`.

```
PROGRAM Integer_Representation
!
! Scopo: stampare un valore intero secondo la rappresentazione
!        binaria, senza fare uso del descrittore di formato "B"
!
  INTEGER :: num  ! intero da stampare
  INTEGER :: j    ! variabile contatore
  INTEGER, PARAMETER :: size=32
  CHARACTER(LEN=size) :: num_in_bits  ! rappresentazione binaria di num
!
  PRINT*, " Introdurre un intero: "
  READ(*,*) num
  num_in_bits = " "
  DO j=0,size-1
     IF (BTEST(num,j)) THEN
        num_in_bits(size-j:size-j) = "1"
     ELSE
        num_in_bits(size-j:size-j) = "0"
     END IF
  END DO
  PRINT*, "La rappresentazione binaria di ", num, "e': "
  PRINT*, num_in_bits
END PROGRAM  Integer_representation
```

Del programma si offre anche un esempio di impiego:

```
 Introdurre un intero:
13
 La rappresentazione binaria di  13 e':
 00000000000000000000000000001101
```

`IAND(I,J)`

Restituisce l'*and logico* di tutti i bit di `I` con i corrispondenti bit di `J`, in accordo con la seguente *tabella di verità*:

`I`	1	1	0	0
`J`	1	0	1	0
`IAND(I,J)`	1	0	0	0

Gli argomenti `I` e `J` devono avere medesimo parametro di kind, che corrisponde al parametro di kind del risultato.

`IBCLR(I,POS)`

Produce un intero avente stesso parametro di kind di `I` ed il cui valore viene valutato a partire da quello di `I` ponendone pari a zero il bit `POS`. Naturalmente il valore intero `POS` deve soddisfare alla condizione `0`$\leq$`POS`$\leq$`BIT_SIZE(I)`.

`IBITS(I,POS,LEN)`

Restituisce un intero avente stesso parametro di kind di `I` e valori tutti nulli tranne i `LEN` valori a partire dal bit `POS` che coincidono con i corrispondenti bit di `I`. Gli argomenti `POS` e `LEN` sono interi non negativi soddisfacenti alla condizione `POS`+`LEN`$\leq$`BIT_SIZE(I)`.

`IBSET(I,POS)`

Produce un intero avente stesso parametro di kind di `I` ed il cui valore viene valutato a partire da quello di `I` ponendone pari ad uno il bit `POS`. Naturalmente il valore intero `POS` deve soddisfare alla condizione `0`$\leq$`POS`$\leq$`BIT_SIZE(I)`.

`IEOR(I,J)`

Restituisce l'*or esclusivo* di tutti i bit di `I` con i corrispondenti bit di `J`, in accordo con la seguente *tabella di verità*:

`I`	1	1	0	0
`J`	1	0	1	0
`IEOR(I,J)`	0	1	1	0

Gli argomenti `I` e `J` devono avere medesimo parametro di kind, che corrisponde al parametro di kind del risultato.

`IOR(I,J)`

Restituisce l'*or inclusivo* di tutti i bit di `I` con i corrispondenti bit di `J`, in accordo con la seguente *tabella di verità*:

`I`	1	1	0	0
`J`	1	0	1	0
`IOR(I,J)`	1	1	1	0

Gli argomenti `I` e `J` devono avere medesimo parametro di kind, che corrisponde al parametro di kind del risultato.

`ISHIFT(I,SHIFT)`

Restituisce un intero con il medesimo parametro di kind di `I` e valore ottenuto da quello di `I` spostandone i bit di `SHIFT` posizioni a sinistra, se `SHIFT` è positivo, a destra se `SHIFT` è negativo. I bit "sgomberati" sono posti pari a zero. L'argomento intero `SHIFT` deve soddisfare alla condizione $|$`SHIFT`$|\leq$`BIT_SIZE(I)`.

`ISHIFTC(I,SHIFT[,SIZE])`

Fornisce un intero avente il parametro di kind di `I` e valore ottenuto applicando ai `SIZE` bit più a destra di `I` (o a tutti i bit di `I` se `SIZE` è assente) uno "shift circolare" a destra (se `SHIFT` è positivo) o a sinistra (se `SHIFT` è negativo). Il valore dell'argomento intero `SHIFT` non deve superare, in valore assoluto, il valore di `SIZE` (o di `BIT_SIZE(I)` se `SIZE` è assente).

`NOT(I)`

Restituisce il *complemento logico* di tutti i bit di `I`, in accordo con la seguente tavola di verità:

`I`	0	1
`NOT(I)`	1	0

`MVBITS(FROM,FROMPOS,LEN,TO,TOPOS)`

Si tratta dell'unica subroutine del gruppo. Essa copia la sequenza di bit di `FROM` di lunghezza `LEN` e che parte dalla posizione `FROMPOS` nel target `TO` a partire dalla posizione `TOPOS`. I bit rimanenti restano inalterati. Tutti i parametri sono di tipo `INTEGER` ed hanno attributo `INTENT(IN)` ad eccezione di `TO` che ha attributo `INTENT(INOUT)`. L'argomento `TO`, inoltre, deve avere lo stesso parametro di kind di `FROM`. Si noti che una stessa variabile può essere specificata contemporaneamente come `FROM` e come `TO`. I parametri di questa procedura devono soddisfare alle condizioni:

`FROMPOS`$+$`LEN`$\leq$`BIT_SIZE(FROM)`
`LEN`$\geq$`0`
`FROMPOS`$\geq$`0`
`TOPOS`$+$`LEN`$\leq$`BIT_SIZE(TO)`
`TOPOS`$\geq$`0`

Come esempio di applicazione della funzione "di scorrimento" `ISHFT`, si guardi il seguente programma:

```
PROGRAM prova_bit
  IMPLICIT NONE
  INTEGER :: n=84, m
  CHARACTER(LEN=29) :: formato1='(1X,A31,I3,1X,A3,1X,B12.12)'
  CHARACTER(LEN=47) :: formato2='(1X,A9,1X,I3,1X,A31,1X,B12.12, &
                                  /,1X,A33,1X,I3)'
  CHARACTER(LEN=47) :: formato3='(1X,A9,1X,I3,1X,A29,1X,B12.12, &
                                  /,1X,A33,1X,I3)'
! Start
  WRITE(*,formato1) "La rappresentazione binaria di ",n,"e': ", n
  WRITE(*,formato1) "La rappresentazione binaria di ",n/2,"e':",n/2
  WRITE(*,formato1) "La rappresentazione binaria di ",n/4,"e':",n/4
  WRITE(*,*)
  m=ISHFT(n,1)
  WRITE(*,formato2) "Shiftando",n,"a sinistra di 1 bit si ottiene:",m, &
                    "che corrisponde al numero intero:",m
  m=ISHFT(n,2)
  WRITE(*,formato2) "Shiftando",n,"a sinistra di 2 bit si ottiene:",m, &
                    "che corrisponde al numero intero:",m
  WRITE(*,'(/,1X,A,/)') "Viceversa:"
  m=ISHFT(n,-1)
  WRITE(*,formato3) "Shiftando",n,"a destra di 1 bit si ottiene:",m, &
                    "che corrisponde al numero intero:",m
  m=ISHFT(n,-2)
  WRITE(*,formato3) "Shiftando",n,"a destra di 2 bit si ottiene:",m, &
                    "che corrisponde al numero intero:",m
  STOP
END PROGRAM prova_bit
```

il cui output è:

```
 La rappresentazione binaria di  84 e': 000001010100
 La rappresentazione binaria di  42 e': 000000101010
 La rappresentazione binaria di  21 e': 000000010101

 Shiftando  84 a sinistra di 1 bit si ottiene: 000010101000
 che corrisponde al numero intero: 168
 Shiftando  84 a sinistra di 2 bit si ottiene: 000101010000
 che corrisponde al numero intero: 336

 Viceversa:
```

```
Shiftando  84 a destra di 1 bit si ottiene: 000000101010
che corrisponde al numero intero:  42
Shiftando  84 a destra di 2 bit si ottiene: 000000010101
che corrisponde al numero intero:  21
```

Osservando con attenzione l'output prodotto da questo programma ci si accorge immediatamente che spostare verso sinistra di una posizione i bit di un numero equivale a *moltiplicare* questo numero per due (purché, è doveroso aggiungere, i bit `1` non oltrepassino la posizione limite e si faccia uso di interi senza segno). Analogamente, lo scorrimento verso destra di una posizione ha l'effetto di *dividere* quel numero per due. Un'applicazione molto interessante di questa proprietà potrebbe essere, pertanto, quella di utilizzare la funzione `ISHFT` in luogo dei comuni operatori algebrici `*` e `/` nel caso in cui sia necessario eseguire *in maniera estremamente rapida* delle semplici operazioni di divisione e moltiplicazione per potenze di due, purché, naturalmente, i dati corrispondano alle limitazioni indicate.

5.13 Effetti collaterali nelle funzioni

Attraverso la lista degli argomenti una function riceve i puntatori alle locazioni di memoria dei suoi argomenti e può, deliberatamente o accidentalmente, modificare il contenuto di queste locazioni. Dunque, una function può modificare i valori dei suoi argomenti; ciò accade se uno o più dei suoi argomenti compaiono alla sinistra di un'istruzione di assegnazione. Una funzione che modifichi il valore dei suoi argomenti è detta avere *effetti collaterali* (*side effect*). Per definizione, una funzione deve produrre un *singolo valore di output* che sia *univocamente determinato* a partire dai valori dei suoi argomenti di input, e non dovrebbe *mai* avere effetti collaterali. Se il programmatore necessita di produrre più di un output dalla sua procedura, allora dovrà scrivere una subroutine e non una function. Per assicurare che gli argomenti di una funzione non vengano accidentalmente modificati essi dovrebbero sempre essere dichiarati con l'attributo `INTENT(IN)`.

Al fine di comprendere quali effetti subdoli possono avere gli effetti collaterali di una function, si osservi il seguente esempio. Si consideri la seguente istruzione di assegnazione:

```
somma = func1(a,b,c)+func2(a,b,c)
```

in cui le function `func1` e `func2` sono le seguenti:

```
INTEGER FUNCTION func1(a,b,c)
   IMPLICIT NONE
   ...
      a = a*a
      func1 = a/b
END FUNCTION func1

INTEGER FUNCTION func2(a,b,c)
```

```
    IMPLICIT NONE
    ...
       a = a*2
       func2 = a/c
END FUNCTION func2
```

Si noti come *entrambe* le funzioni modifichino il valore del parametro `a`: questo significa che il valore di `somma` è totalmente dipendente dall'ordine (indefinito) in cui vengono eseguite le due funzioni. Posto, ad esempio, `a=4`, `b=2` e `c=4`, si possono presentare i due seguenti casi:

- Se `func1` è eseguita per prima, allora *somma* sarà pari a `16`. Infatti:
  - in `func1` `a` è inizialmente uguale a `4`;
  - all'uscita da `func1` `a` vale `16`;
  - `func1` è uguale a `16/2=8`;
  - in `func2` `a` è inizialmente pari a `16`;
  - all'uscita da `func2` `a` vale `32`;
  - `func2` è uguale a `32/4=8`;
  - pertanto `somma` è pari a `8+8=16`.
- Se `func2` è eseguita per prima, allora *somma* sarà pari a `34`. Infatti:
  - in `func2` `a` è inizialmente pari a `4`;
  - all'uscita da `func2` `a` vale `8`;
  - `func2` è uguale a `8/4=2`;
  - in `func1` `a` è inizialmente pari a `8`;
  - all'uscita da `func1` `a` vale `64`;
  - `func1` è uguale a `64/2=32`;
  - pertanto `somma` è pari a `2+32=34`.

Al fine di risolvere questo problema alla radice, il Fortran 95 ha introdotto una nuova categoria di procedure, le *pure procedure*, caratterizzate dal fatto di essere totalmente esenti da effetti collaterali. Le *pure function* si definiscono aggiungendo il *prefisso* `PURE` all'istruzione di dichiarazione della funzione. Esse si caratterizzano per il fatto che tutti gli argomenti *devono* avere l'attributo `INTENT(IN)`, nessuna variabile locale può avere attributo `SAVE` né sono permesse inizializzazioni delle variabili in istruzioni di dichiarazione. Ogni procedura chiamata da una *pure function* deve essere, a sua volta, una *pure procedure*. Inoltre, esse non possono contenere operazioni di I/O *esterne* né è permesso l'uso dell'istruzione `STOP`.

Un esempio di *pure function* è il seguente:

```
PURE FUNCTION lunghezza(x,y)
   IMPLICIT NONE
   REAL, INTENT(IN) :: x, y
   REAL :: lunghezza
      lunghezza = SQRT(x**2+y**2)
END FUNCTION lunghezza
```

Analogamente, le *pure subroutine* si definiscono aggiungendo il prefisso `PURE` all'istruzione di dichiarazione di subroutine. I loro vincoli sono esattamente uguali a quelli delle *pure function* tranne per il fatto che esse *possono* modificare i valori degli argomenti dichiarati con l'attributo `INTENT(OUT)` o `INTENT(INOUT)`.

Si noti che le funzioni `PURE` sono le uniche funzioni, a parte quelle intrinseche, che possono essere invocate all'interno di un costrutto `FORALL`. Così, ad esempio, il seguente costrutto:

```
FORALL(i=1:n,j=1:m)
   x(i,j) = my_func(y(i,j))
END FORALL
```

è valido soltanto se la funzione `my_func` viene definita con l'attributo `PURE`, come ad esempio:

```
PURE FUNCTION my_func(var1) RESULT(var2)
   REAL(KIND(1.D0)), INTENT(IN)  :: var1
   REAL(KIND(1.D0))              :: var2
   ...
END FUNCTION my_func
```

In tutti gli altri casi l'istruzione di chiamata di funzione produce un errore di compilazione.

Una particolare classe di procedure `PURE`, sono le *elemental procedure*, le quali si definiscono sostituendo al prefisso `PURE` il prefisso `ELEMENTAL` nell'istruzione di dichiarazione di procedura.

Le *elemental function* non sono altro che *pure function* aventi come parametri formali soltanto variabili scalari (quindi *non* sono ammessi come parametri formali né puntatori né procedure) e che forniscono un risultato scalare. Si noti che il prefisso `RECURSIVE` è incompatibile con il prefisso `ELEMENTAL`.

Le *elemental subroutine* sono definite allo stesso modo. L'unica differenza rispetto alle corrispondenti function è il fatto che *è permesso* modificare i valori dei parametri formali specificati con gli attributi `INTENT(OUT)` o `INTENT(INOUT)`.

Il vantaggio sostanziale offerto dalle procedure `ELEMENTAL` rispetto alle `PURE` consiste in una più efficace *parallelizzazione* delle operazioni. A parte le differenze testé citate, le *elemental procedure* si comportano esattamente come le comuni procedure, come dimostrato dal seguente esempio in cui si confrontano i valori della funzione esponenziale e^x ottenuti con la funzione intrinseca `EXP` e con i primi venti termini dello sviluppo in serie di Taylor.

```
PROGRAM prova_user_exp
!
  IMPLICIT NONE
```

```
  INTEGER, PARAMETER :: max_test = 8
  REAL,DIMENSION(max_test) :: x_test=(/-10.,-5.,-1.,0.,1.,5.,10.,15./)
  REAL,DIMENSION(max_test) :: series_exp, intrinsic_exp, rel_err
  INTEGER :: i
  INTERFACE
     REAL ELEMENTAL FUNCTION user_exp(x) RESULT(exp_x)
        IMPLICIT NONE
        REAL, INTENT(IN) :: x
     END FUNCTION user_exp
  END INTERFACE
!
  series_exp = user_exp(x_test)
  intrinsic_exp = EXP(x_test)
  rel_err = ABS((series_exp - intrinsic_exp)/intrinsic_exp)
  WRITE(*,'(T9,A,T17,A,T30,A,T43,A,/,T6,52("="))') &
        "x", "serie", "intrinseca", "errore relativo"
  DO i =1,max_test
     WRITE(*,'(F10.1,3ES15.6)') x_test(i), series_exp(i), &
        intrinsic_exp(i), rel_err(i)
  END DO
END PROGRAM prova_user_exp
!
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
!
REAL ELEMENTAL FUNCTION user_exp(x) RESULT(exp_x)
!
!Scopo: Stimare il valore di EXP(x) usando i primi 30 termini
! dello sviluppo in serie di the Taylor.
!
   IMPLICIT NONE
   REAL, INTENT(IN) :: x
   INTEGER, PARAMETER :: n_terms = 20
   INTEGER :: n
   REAL :: factor
!
      exp_x = 1
      factor = 1
      DO n =1,n_terms -1
         factor = factor*x/n
         exp_x = exp_x + factor
      END DO
END FUNCTION user_exp
```

5.14 Clausola RESULT per le funzioni

Le funzioni Fortran possono essere definite con una clausola `RESULT` la quale specifica, tra due parentesi, il nome della variabile che immagazzinerà il risultato fornito dalla funzione. A titolo di esempio, si consideri la seguente funzione:

```
FUNCTION addizione(a,b,c) RESULT(somma)
   IMPLICIT NONE
   REAL, INTENT(IN) :: a, b, c
   REAL :: somma
      somma = a+b+c
END FUNCTION addizione
```

Si noti come la dichiarazione del tipo della funzione sia insito nella dichiarazione del tipo della variabile "risultato" (nel caso in esame, la funzione ha lo stesso tipo della variabile `somma` ossia è una funzione *reale*). La clausola `RESULT` è, tipicamente, opzionale; il suo uso risulta, invece, obbligatorio in presenza di una funzione *direttamente ricorsiva* (alle procedure ricorsive sarà dedicato il prossimo paragrafo). Prima, però, si vuole riportare un altro esempio di utilizzo della clausola `RESULT` che sia, al contempo, un ulteriore esempio di funzioni che restituiscono in output una *stringa di caratteri*. La prossima funzione, infatti, restituisce il giorno della settimana (ossia una stringa del tipo `lunedi`, `martedi`, ...) che corrisponde ad una certa data:

```
FUNCTION day_of_week(year,mounth,day) RESULT(weekday)
! Calcola il giorno della settimana nota una data
! La data di input non deve essere anteriore al mese di
! Ottobre del 1752 (infatti furono "cancellati" 11 giorni
! al mese di Settembre del 1752)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: year, mounth, day
   CHARACTER (LEN=9)   :: weekday
! Variabili locali
   INTEGER          :: yr, mnth, hundreds, day_ptr
   INTEGER, PARAMETER, DIMENSION(12) :: max_days =                  &
                          (/31,29,31,30,31,30,31,31,30,31,30,31/)
   CHARACTER(LEN=9), PARAMETER :: day_name(0:6) = (/'Domenica ',   &
                                        'Lunedi   ','Martedi  ',  &
                                        'Mercoledi','Giovedi  ',  &
                                        'Venerdi  ','Sabato   ' /)
! Check sul mese
      weekday = CHAR(7) // 'Illegale'
      mnth=mounth
      IF (mnth<1 .OR. mnth>12) RETURN
! Numera i mesi a partire da Marzo; Gennaio e Febbraio sono
! trattati come i mesi 11 e 12 dell'anno precedente
```

```
      mnth = mounth-2
      IF (mnth <= 0) THEN
         mnth = mnth+12
         yr = year-1
      ELSE
         yr = year
      END IF
! Check sul giorno del mese
! N.B. Sono "consentiti" 29 giorni a Febbraio anche negli anni
! non bisestili
      IF (day < 1 .OR. day > max_days(mnth)) RETURN
      hundreds = yr/100
      yr = yr - 100*hundreds
! I giorni sono numerati a partire dalla Domenica (0) fino al
! Sabato (6)
      day_ptr = MOD(day+(26*mnth-2)/10 + 5*hundreds +  &
                      yr + (yr/4) + (hundreds/4), 7)
      weekday = day_name(day_ptr)
      RETURN
END FUNCTION day_of_week

PROGRAM which_day
  IMPLICIT NONE
  INTERFACE
     FUNCTION day_of_week(year,mounth,day) RESULT(weekday)
        IMPLICIT NONE
        INTEGER, INTENT(IN) :: year, mounth, day
        CHARACTER(LEN=9) :: weekday
     END FUNCTION day_of_week
  END INTERFACE
  INTEGER :: year, mounth, day
!
  WRITE(*,*) "Inserisci la data come gg, mm & aaa: "
  READ(*,*) day,mounth,year
  WRITE(*,"(1X,A,A)") "Giorno: ", day_of_week(year,mounth,day)
  STOP
END PROGRAM which_day
```

Un possibile esempio di utilizzo di questo programma potrebbe essere il seguente:

```
C:\MYPROG>which_day
Inserisci la data come gg, mm & aaa:
12 05 1972
 Giorno: Venerdi
```

5.15 Procedure ricorsive

Il Fortran 90 consente a una procedura di invocare sé stessa, direttamente o indirettamente: una tale procedura è detta *ricorsiva* e *deve* essere definita con l'attributo `RECURSIVE`. In particolare, per le *funzioni* è necessario specificare la clausola `RESULT` poiché il nome di una funzione ricorsiva *non* può essere usato "direttamente" come una normale variabile. Il seguente esempio, che illustra un metodo molto elegante per il calcolo del fattoriale di un intero, servirà a chiarire l'uso delle funzioni ricorsive:

```
RECURSIVE FUNCTION fattoriale(n) RESULT(fatt)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: n
   INTEGER :: fatt
      IF(n==1) THEN
         fatt = 1
      ELSE
         fatt = n*fattoriale(n-1)
      END IF
END FUNCTION fattoriale
```

Lo stesso esempio può essere prodotto anche a mezzo di una subroutine:

```
RECURSIVE SUBROUTINE fattoriale(n,n_fatt)
   IMPLICIT NONE
   INTEGER, INTENT(IN)  :: n
   INTEGER, INTENT(OUT) :: n_fatt
   INTEGER :: tmp
      IF(n>=1) THEN
         CALL fattoriale(n-1,tmp)
         n_fatt = n*tmp
      ELSE
         n_fatt = 1
      END IF
END SUBROUTINE fattoriale
```

Come ulteriore esempio, ci si propone di sviluppare un programma che risolva il problema della *torre di Hanoi*. Il gioco consiste nello spostare, uno per volta, una serie di dischi, inizialmente impilati su un piolo per grandezza decrescente, dal piolo di origine ad un altro piolo, utilizzando un terzo piolo di appoggio, muovendo un disco alla volta e senza mai collocare un disco più grande sopra uno più piccolo. Il problema in esame ben si presta ad essere risolto con un procedimento ricorsivo come si può facilmente capire seguendo questo ragionamento. Se si gioca con un solo disco la soluzione è banale. D'altra parte, se si sa risolvere il gioco con n dischi lo si sa risolvere anche con $n+1$ dischi, basta allo scopo utilizzare l'algoritmo per n dischi spostando gli n dischi superiori dal piolo iniziale a quello intermedio, spostando, quindi, il disco più grande dal piolo iniziale a quello finale ed infine riapplicando l'algoritmo valido per

n dischi per spostare i suddetti dischi (ora sul piolo intermedio) sovrapponendoli al disco più grande appena disposto sul piolo finale. Il seguente programma è nulla di più che la semplice traduzione Fortran di questo procedimento:

```
PROGRAM test_hanoi
! Scopo: Risolvere il problema della torre di Hanoi con un assegnato
!        numero di dischi
! MEMO:
!          +        +        +              +        +        +
!          +        +        +              +        +        +
!        _|+|_      +        +              +      _|+|_      +
!      _|__+__|_    +        +              +    _|__+__|_    +
!    _|____+____|_  +        +              +  _|____+____|_  +
!  _|______+______|_+________+______   _____+_|______+______|_+______
!
!
!          +        +        +              +        +        +
!          +        +        +              +        +        +
!          +        +        +              +     ___+___     +
!      ____+____    +        +              +    |___+___|    +
!    _|____+____|_  +      _|+|_            +  ___|__+__|___  +
!  _|______+______|_+_____|__+__|___   _____+_|______+______|_+______
!
!                  OK                              ERRATO
  IMPLICIT NONE
  INTEGER :: n    ! numero di dischi

  WRITE(*,'(1X,A)',ADVANCE='NO') "Con quanti dischi vuoi giocare? "
  READ(*,*) n
  PRINT*, "Soluzione:"
  CALL hanoi(n,1,3)

CONTAINS
   RECURSIVE SUBROUTINE hanoi(n,starting,goal)
      IMPLICIT NONE
      INTEGER, INTENT (IN) :: n,starting,goal
! la variabile "all" e' la somma dei valori dei tre pioli 1+2+3
! sicche' il posto libero puo' essere determinato sottraendo
! i valori delle variabili "starting" e "goal" da questa somma.
       INTEGER :: free    ! posti liberi
       INTEGER, PARAMETER :: all = 6

        IF (n > 0) THEN
           free = all-starting-goal
```

```
            CALL hanoi(n-1,starting,free)
            PRINT 100, "Muovi il disco", n, "dal piolo", &
                     starting, "al piolo", goal
            CALL hanoi(n-1,free,goal)
          END IF
100 FORMAT(1X,A15,I3,1X,A10,1X,I1,1X,A9,1X,I1)
  END SUBROUTINE hanoi
END PROGRAM test_hanoi
```

Si riporta, per completezza, un esempio di run del precedente programma:

```
C:\MYPROG>hanoi
 Con quanti dischi vuoi giocare? 4
 Soluzione:
  Muovi il disco  1  dal piolo 1  al piolo 2
  Muovi il disco  2  dal piolo 1  al piolo 3
  Muovi il disco  1  dal piolo 2  al piolo 3
  Muovi il disco  3  dal piolo 1  al piolo 2
  Muovi il disco  1  dal piolo 3  al piolo 1
  Muovi il disco  2  dal piolo 3  al piolo 2
  Muovi il disco  1  dal piolo 1  al piolo 2
  Muovi il disco  4  dal piolo 1  al piolo 3
  Muovi il disco  1  dal piolo 2  al piolo 3
  Muovi il disco  2  dal piolo 2  al piolo 1
  Muovi il disco  1  dal piolo 3  al piolo 1
  Muovi il disco  3  dal piolo 2  al piolo 3
  Muovi il disco  1  dal piolo 1  al piolo 2
  Muovi il disco  2  dal piolo 1  al piolo 3
  Muovi il disco  1  dal piolo 2  al piolo 3
```

Come ultimo esempio si vuole riportare un programma che implementa un algoritmo, quello di ordinamento *a doppio indice* (anche detto *quick sort*), che si presta ad essere espresso in forma ricorsiva in maniera molto naturale. L'algoritmo in esame, che si caratterizza per una eccezionale efficienza da un punto di vista di complessità di calcolo, può così essere schematizzato:

1. Si preleva un elemento dalla lista `a`, e lo si pone nella sua posizione corretta ossia si spostano alla sua sinistra tutti i valori minori di questo elemento e si spostano alla sua destra tutti gli elementi maggiori.

2. Si perviene così a due sottoliste (quella di sinistra e quella di destra) ancora intrinsecamente disordinate ma comunque tali che ciascun loro elemento non dovrà lasciare, nelle successive operazioni di ordinamento, la propria sottolista.

3. Applicando ricorsivamente questo procedimento alle due sottoliste si perviene ad una successione di liste che, alla fine, degenerano ciascuna in un solo elemento. A questo punto la lista di partenza è completamente ordinata.

```
RECURSIVE SUBROUTINE quick_sort(a)
   IMPLICIT NONE
   INTEGER, DIMENSION(:), INTENT(inout) :: a
   INTEGER :: i,n
      n = SIZE(a)
      IF (n>1) THEN
         CALL partition(a,i)
         CALL quick_sort(a(:i-1))
         CALL quick_sort(a(i+1:))
      END IF
CONTAINS
   SUBROUTINE partition(a,j)
      INTEGER, DIMENSION(:), INTENT(inout) :: a
      INTEGER,INTENT(out) :: j
      INTEGER :: i,temp
      i = 1
      j = SIZE(a)
      DO
         DO
            IF (i>j) EXIT
            IF (a(i)>a(1)) EXIT
            i = i+1
         END DO
         DO
            IF ((j<i) .OR. (a(j)<=a(1))) EXIT
            j = j-1
         END DO
         IF (i>=j) EXIT
         temp = a(i)
         a(i) = a(j)
         a(j) = temp
      END DO
      temp = a(j)
      a(j) = a(1)
      a(1) = temp
   END SUBROUTINE partition
END SUBROUTINE quick_sort

PROGRAM prova_quick_sort
  IMPLICIT NONE
  INTEGER,PARAMETER :: n=10
  INTEGER,DIMENSION(n) :: a=(/11, 6, 2, 3, 13, 0, 24, -2, 9, 4/)
  INTERFACE
```

```
      RECURSIVE SUBROUTINE quick_sort(a)
         IMPLICIT NONE
         INTEGER, DIMENSION(:), INTENT(inout) :: a
      END SUBROUTINE quick_sort
   END INTERFACE
   PRINT "('Lista di partenza:',/,<n>(I2,1X),/)", a
   CALL quick_sort(a)
   PRINT "('Lista ordinata:',/,<n>(I2,1X))", a
END PROGRAM prova_quick_sort
```

L'output del programmino di test fornisce questo risultato:

```
Lista di partenza:
11  6  2  3 13  0 24 -2  9  4

Lista ordinata:
-2  0  2  3  4  6  9 11 13 24
```

a riprova della bontà della routine.

In tutti gli esempi sviluppati in questo paragrafo una procedura ricorsiva era tale perché invocava direttamente sé stessa (si parla in tal caso di *ricorsione diretta*). Tuttavia, una procedura può essere invocata anche per *ricorsione indiretta*, cioè può chiamare sé stessa attraverso l'invocazione di un'altra procedura che la richiami. Per illustrare questo concetto può essere utile riferirsi ad un esempio concreto. Si supponga si voler eseguire l'integrazione di una funzione bidimensionale $f(x, y)$ su di un dominio rettangolare avendo, però, a disposizione soltanto una procedura di integrazione monodimensionale del tipo seguente:

```
RECURSIVE FUNCTION integrate(f,bounds)
! Scopo: Integrare f da bounds(1) a bounds(2)
   IMPLICIT NONE
   REAL :: integrate
   INTERFACE
      FUNCTION f(x)
         REAL :: f
         REAL, INTENT(IN) :: x
      END FUNCTION f
   END INTERFACE
   REAL, DIMENSION(2),INTENT(IN) :: bounds
   ...
END FUNCTION integrate
```

Ebbene, in questo caso si potrebbe scrivere una funzione di modulo che riceva il valore di x come argomento ed il valore di y dal modulo stesso per host association:

```
MODULE func
```

```
   IMPLICIT NONE
   REAL :: yval
   REAL, DIMENSION(2) :: xbounds, ybounds
CONTAINS
   FUNCTION f(xval)
      REAL :: f
      REAL, INTENT(IN) :: xval
         r = ...  ! Espressione che coinvolge xval e yval
   END FUNCTION f

END MODULE func
```

e, quindi, integrare su x per un particolare valore di y al modo seguente:

```
FUNCTION fy(y)
   USE func
   IMPLICIT NONE
   REAL :: fy
   REAL, INTENT(IN) :: y
      yval = y
      r = integrate(f,xbounds)
END FUNCTION fy
```

L'integrazione sull'intero rettangolo potrebbe, così, avvenire a mezzo dell'unica chiamata di procedura:

```
volume = integrate(fy,ybounds)
```

5.16 Procedure generiche

Accade spesso che le operazioni eseguite da una procedura su un particolare tipo di dati possano essere eseguite in un modo formalmente analogo anche su tipi di dati differenti. Ad esempio, una procedura scritta per ordinare in senso crescente gli elementi di un array *reale* sarà formalmente identica ad una procedura che, invece, sia stata scritta per ordinare gli elementi di un array *intero*: l'unica differenza fra le due procedure sarà il *tipo* dei parametri formali. Per comodità, il Fortran 90/95 permette a due o più procedure di essere *chiamate* con lo stesso *nome generico*: quale delle due (o più) procedure verrà effettivamente invocata in una istruzione di chiamata dipenderà esclusivamente dal tipo (e/o dal rango) degli argomenti attuali. Questa proprietà è nota come *polimorfismo*.

Una *procedura generica* è definita usando un *generic interface block* che specifichi il nome comune per tutte le procedure definite nel blocco. La forma generale di questo costrutto è la seguente:

```
INTERFACE nome_generico
  corpo dell'interfaccia
END INTERFACE
```

Tutte le procedure specificate nel blocco *interface* devono poter essere invocate in maniera non ambigua e di conseguenza ogni *generic interface block* può contenere solo funzioni o solo subroutine. A titolo di esempio, si supponga di volere scrivere una subroutine che permetta di "invertire" due numeri che possono essere, a seconda dei casi, entrambi reali o entrambi complessi oppure entrambi interi. Ciò può essere fatto scrivendo le tre subroutine seguenti:

```
SUBROUTINE swapint
   IMPLICIT NONE
   INTEGER, INTENT(INOUT) :: a, b
   INTEGER :: temp
      temp = a
      a = b
      b = temp
END SUBROUTINE swapint

SUBROUTINE swapreal
   IMPLICIT NONE
   REAL, INTENT(INOUT) :: a, b
   REAL :: temp
      temp = a
      a = b
      b = temp
END SUBROUTINE swapreal

SUBROUTINE swapcomplex
   IMPLICIT NONE
   COMPLEX, INTENT(INOUT) :: a, b
   COMPLEX :: temp
      temp = a
      a = b
      b = temp
END SUBROUTINE swapcomplex
```

Dato, allora, il seguente *interface block*:

```
INTERFACE swap
   SUBROUTINE swapint(a,b)
      INTEGER, INTENT(INOUT) :: a, b
   END SUBROUTINE swapint

   SUBROUTINE swapreal(a,b)
      REAL, INTENT(INOUT) :: a, b
   END SUBROUTINE swapreal
```

```
    SUBROUTINE swapcomplex(a,b)
       REAL, INTENT(INOUT) :: a, b
    END SUBROUTINE swapcomplex
END INTERFACE
```

per invertire due numeri `x` ed `y` sarà sufficiente invocare la procedura generica `swap` mediante l'istruzione di chiamata `swap(x,y)` indipendentemente dal tipo (intero, reale o complesso) dei parametri attuali.

Tutto quanto detto finora vale esclusivamente per funzioni *esterne* compilate separatamente e *non* aventi una interfaccia esplicita. Un discorso a parte vale nel caso in cui le procedure "individuali" fanno parte di un *modulo* e quindi hanno già una interfaccia esplicita. Sebbene sia illegale inserire una procedura di modulo in un *interface block*, è, invece, perfettamente lecito inserire una procedura di modulo in un *generic interface block* attraverso la speciale istruzione `MODULE PROCEDURE`, la cui sintassi è:

```
MODULE PROCEDURE module_procedure1 [, module_procedure2, ...]
```

in cui *`module_procedure`* è una procedura di modulo resa disponibile attraverso una associazione di `USE`. A titolo di esempio, se le procedure `swapint`, `swapreal` e `swapcomplex` dell'esempio precedente non fossero state delle procedure esterne bensì delle procedure di modulo, allora il *generic interface block* sarebbe dovuto essere il seguente:

```
INTERFACE swap
    MODULE PROCEDURE swapint
    MODULE PROCEDURE swapreal
    MODULE PROCEDURE swapcomplex
END INTERFACE
```

5.17 *Overloading*

Il riferimento ad una procedura, facente parte di un certo insieme, attraverso un nome generico è noto come *overloading*. Quale fra le diverse procedure sarà invocata dipende dagli argomenti scambiati. Allo stesso modo che con i nomi di procedura è possibile "generalizzare" i comuni operatori aritmetici (`+`, `-`, `*`, `/`, `**`) al fine di definire un set di operatori che possano essere applicati, ad esempio, a variabili di tipi di dati derivati.

Il Fortran 90/95 consente sia l'*overloading* degli operatori che l'*overloading* dell'assegnazione e in entrambi i casi è necessario un *interface block*. Moduli sono spesso usati per fornire un accesso *globale* ai *nuovi* operatori.

5.17.1 *Overloading* degli operatori

E' possibile estendere il "significato" di un operatore *intrinseco* per poterlo applicare a tipi di dati supplementari. Questo richiede un *interface block* nella forma:

```
INTERFACE OPERATOR (operatore_intrinseco)
  corpo dell'interfaccia
END INTERFACE
```

dove *operatore_intrinseco* è l'operatore cui applicare l'*overloading* mentre con l'espressione *corpo dell'interfaccia* si è indicato una *function* con uno o due argomenti aventi attributo `INTENT(IN)`.

Ad esempio, l'operatore di addizione, "+", potrebbe essere esteso alle variabili di tipo carattere al fine di concatenare due stringhe ignorando ogni eventuale spazio bianco di coda, e questa estensione potrebbe essere efficacemente inserita in un modulo:

```
MODULE overloading_addizione
   IMPLICIT NONE
   ...
   INTERFACE OPERATOR (+)
      MODULE PROCEDURE concat
   END INTERFACE
   ...
CONTAINS
   FUNCTION concat(cha,chb)
      IMPLICIT NONE
      CHARACTER (LEN=*), INTENT(IN) :: cha, chb
      CHARACTER (LEN=(LEN_TRIM(cha) + LEN_TRIM(chb))) :: concat
         concat = TRIM(cha)//TRIM(chb)
   END FUNCTION concat
   ...
END MODULE overloading_addizione
```

In una qualsiasi unità di programma che usi il modulo `overloading_addizione`, l'espressione `cha+chb` è perfettamente interpretata come una operazione di concatenamento di stringhe.

Un altro esempio, sempre relativo alle variabili di tipo stringa, potrebbe essere quello di definire una operazione di divisione ("/") fra una stringa e una variabile carattere con il significato di fornire il numero di volte che detto carattere compare all'interno della stringa. Ciò può essere ottenuto al modo seguente:

```
MODULE  overloading_divisione
   INTERFACE OPERATOR (/)
      MODULE PROCEDURE num
   END INTERFACE
CONTAINS
   INTEGER FUNCTION num(s,c)
      CHARACTER(LEN=*), INTENT(IN) :: s
      CHARACTER, INTENT(IN) :: c
         num = 0
         DO i=1,LEN(s)
```

```
            IF(s(i:i)== c) num = num+1
         END DO
   END FUNCTION num
END MODULE overloading_divisione
```

Normalmente il comune operatore di divisione non è definito per variabili carattere o stringhe ma il modulo `overloading_divisione` contiene una function che definisce una simile operazione, ed un'interfaccia ad essa. Così, ad esempio, la divisione fra la stringa `'hello world'` ed un carattere potrà essere effettuata al seguente modo:

```
USE overloading_divisione
...
i = "hello world"/"l"     ! i = 3
i = "hello world"/"o"     ! i = 2
i = "hello world"/"z"     ! i = 0
```

Per rendere ancora più chiara la versatilità e la generalità dell'*overloading degli operatori*, si farà ora un esempio un pò più complesso. Si immagini di lavorare con un tipo di dati derivato, `intervallo`, i cui elementi, `lower` e `upper`, rappresentino gli estremi inferiore e superiore di un intervallo numerico, e si supponga di voler creare una "aritmetica" degli intervalli definita in modo tale che, dati due "intervalli", un comune operatore aritmetico binario applicato alla coppia restituisca una nuova variabile `intervallo` i cui componenti (gli estremi dell'intervallo risultante) siano ottenuti applicando l'operatore aritmetico "intrinseco" alle coppie formate dai componenti omologhi. Il modulo seguente, evidentemente, è quanto occorre allo scopo:

```
MODULE aritmetica_intervalli
   TYPE intervallo
      REAL lower, upper
   END TYPE intervallo
   INTERFACE OPERATOR (+)
      MODULE PROCEDURE addizione_intervalli
   END INTERFACE
   INTERFACE OPERATOR (-)
      MODULE PROCEDURE sottrazione_intervalli
   END INTERFACE
   INTERFACE OPERATOR (*)
      MODULE PROCEDURE moltiplicazione_intervalli
   END INTERFACE
   INTERFACE OPERATOR (/)
      MODULE PROCEDURE divisione_intervalli
   END INTERFACE
   CONTAINS
      FUNCTION addizione_intervalli(a,b)
         TYPE(intervallo), INTENT(IN) :: a, b
         TYPE(intervallo) :: addizione_intervalli
```

```
            addizione_intervalli%lower = a%lower + b%lower
            addizione_intervalli%upper = a%upper + b%upper
      END FUNCTION addizione_intervalli
      FUNCTION sottrazione_intervalli(a,b)
         TYPE(intervallo), INTENT(IN) :: a, b
         TYPE (intervallo) :: sottrazione_intervalli
            sottrazione_intervalli%lower = a%lower - b%upper
            sottrazione_intervalli%upper = a%upper - b%lower
      END FUNCTION sottrazione_intervalli
      FUNCTION moltiplicazione_intervalli(a,b)
         TYPE(intervallo), INTENT(IN) :: a, b
         TYPE (intervallo) :: moltiplicazione_intervalli
            moltiplicazione_intervalli%lower = a%lower * b%lower
            moltiplicazione_intervalli%upper = a%upper * b%upper
      END FUNCTION moltiplicazione_intervalli
      FUNCTION divisione_intervalli(a,b)
         TYPE(intervallo), INTENT(IN) :: a, b
         TYPE(intervallo) :: divisione_intervalli
            divisione_intervalli%lower = a%lower / b%upper
            divisione_intervalli%upper = a%upper / b%lower
      END FUNCTION divisione_intervalli
END MODULE aritmetica_intervalli
```

Il seguente programma fornisce un esempio di applicazione del modulo `aritmetica_intervalli`.

```
PROGRAM abc
  USE aritmetica_intervalli
  IMPLICIT NONE
  TYPE (intervallo) :: a, b, c, d, e, f
  a%lower = 6.9
  a%upper = 7.1
  b%lower = 10.9
  b%upper = 11.1
  WRITE (*,*) a,b
  c = a+b
  d = a-b
  e = a*b
  f = a/b
  WRITE(*,*) c,d
  WRITE(*,*) e,f
END
```

Chiaramente l'output di questo programma di test è il seguente:

```
 6.9000001     7.0999999    10.8999996    11.1000004
```

```
17.7999992    18.2000008    -4.2000003    -3.7999997
75.2099991    78.8100052     0.6216216     0.6513762
```

E' importante osservare, in conclusione, che la *precedenza* di un operatore predefinito *non* viene alterata dall'operazione di *overloading*, per cui, ad esempio, gli operatori "/" e "*" estesi hanno un livello di precedenza superiore rispetto agli operatori "+" e "-", e così via. Inoltre, è da precisare che, allorquando un operatore intrinseco viene sottoposto ad overloading, il numero degli argomenti deve essere consistente con la forma intrinseca (per cui, ad esempio, non sarà possibile definire un operatore * di tipo unario).

Il modulo seguente rappresenta un interessante esempio di applicazione delle possibilità di overloading offerte dal Fortran. Lo scopo di questo esempio è quello di definire un tipo `matrix` tale che gli operatori aritmetici di somma, prodotto, divisione etc. lavorino "a la Matlab". In un certo senso, quindi, si vuole "riscrivere" il significato degli operatori aritmetici * e / in modo tale che questi non lavorino più *elemento*×*elemento*; in particolare si vuole utilizzare l'operatore * per eseguire prodotti righe×colonne e renderlo, pertanto, applicabile anche ad operandi di forma diversa, e l'operatore / per eseguire procedure di inversione di matrici.

```
MODULE operatori

  TYPE matrix
      REAL :: elem
   END TYPE matrix

   INTERFACE OPERATOR(+)
      MODULE PROCEDURE matrix_add, vector_add
   END INTERFACE

   INTERFACE OPERATOR(-)
      MODULE PROCEDURE matrix_sub, vector_sub
   END INTERFACE

   INTERFACE OPERATOR(*)
      MODULE PROCEDURE matrix_mul, vector_mul, matrix_vector_mul
   END INTERFACE

   INTERFACE OPERATOR(/)
      MODULE PROCEDURE matrix_div, matrix_vector_div
   END INTERFACE

CONTAINS
!
! Operatore "somma": X = Y + Z
!
   FUNCTION matrix_add(Y,Z) RESULT(X)
```

```
! Esegue la somma di due array 2D di tipo "matrix"
      TYPE(matrix), INTENT(IN), DIMENSION(:,:) :: Y
      TYPE(matrix), INTENT(IN), DIMENSION(SIZE(Y,1),SIZE(Y,2)) :: Z
      TYPE(matrix), DIMENSION(SIZE(Y,1),SIZE(Y,2)) :: X
         X(:,:)%elem = Y(:,:)%elem + Z(:,:)%elem
   END FUNCTION matrix_add

   FUNCTION vector_add(Y,Z) RESULT(X)
! Esegue la somma di due array 1D di tipo "matrix"
      TYPE(matrix), INTENT(IN), DIMENSION(:) :: Y
      TYPE(matrix), INTENT(IN), DIMENSION(SIZE(Y,1)) :: Z
      TYPE(matrix), DIMENSION(SIZE(Y,1)) :: X
         X(:)%elem = Y(:)%elem + Z(:)%elem
   END FUNCTION vector_add
!
! Operatore "differenza": X = Y - Z
!
   FUNCTION matrix_sub(Y,Z) RESULT(X)
! Esegue la differenza fra due array 2D di tipo "matrix"
      TYPE(matrix), INTENT(IN), DIMENSION(:,:) :: Y
      TYPE(matrix), INTENT(IN), DIMENSION(SIZE(Y,1),SIZE(Y,2)) :: Z
      TYPE(matrix), DIMENSION(SIZE(Y,1),SIZE(Y,2)) :: X
         X(:,:)%elem = Y(:,:)%elem - Z(:,:)%elem
   END FUNCTION matrix_sub

   FUNCTION vector_sub(Y,Z) RESULT(X)
! Esegue la differenza fra due array 1D di tipo "matrix"
      TYPE(matrix), INTENT(IN), DIMENSION(:) :: Y
      TYPE(matrix), INTENT(IN), DIMENSION(SIZE(Y,1)) :: Z
      TYPE(matrix), DIMENSION(SIZE(Y,1)) :: X
         X(:)%elem = Y(:)%elem - Z(:)%elem
   END FUNCTION vector_sub
!
! Operatore "moltiplicazione": X = Y * Z
!
   FUNCTION matrix_mul(Y,Z) RESULT(X)
! Esegue il prodotto righe x colonne fra due array 2D di tipo "matrix"
      TYPE(matrix), INTENT(IN), DIMENSION(:,:) :: Y
      TYPE(matrix), INTENT(IN), DIMENSION(:,:) :: Z
      TYPE(matrix), DIMENSION(SIZE(Y,1),SIZE(Z,2)) :: X
         X(:,:)%elem = MATMUL(Y(:,:)%elem, Z(:,:)%elem)
   END FUNCTION matrix_mul
```

```
   FUNCTION vector_mul(Y,Z) RESULT(X)
! Esegue il prodotto elemento x elemento di due array 1D di tipo "matrix"
      TYPE(matrix), INTENT(IN), DIMENSION(:) :: Y
      TYPE(matrix), INTENT(IN), DIMENSION(SIZE(Y,1)) :: Z
      REAL :: X
         X = DOT_PRODUCT(Y(:)%elem, Z(:)%elem)
   END FUNCTION vector_mul

   FUNCTION matrix_vector_mul(Y,Z) RESULT(X)
! Esegue il prodotto righe x colonne fra un array 2D
! ed un array 1D di tipo "matrix"
      TYPE(matrix), INTENT(IN), DIMENSION(:,:) :: Y
      TYPE(matrix), INTENT(IN), DIMENSION(SIZE(Y,2)) :: Z
      TYPE(matrix), DIMENSION(SIZE(Y,1)) :: X
         X(:)%elem = MATMUL(Y(:,:)%elem, Z(:)%elem)
   END FUNCTION matrix_vector_mul
!
! Operatore "divisione": X = Y / Z = INV(Z) * Y
!
   FUNCTION matrix_div(Y,Z) RESULT(X)
! Esegue il "rapporto" fra due array 2D di tipo "matrix"
      TYPE(matrix), INTENT(IN), DIMENSION(:,:) :: Y
      TYPE(matrix), INTENT(IN), DIMENSION(:,:) :: Z
      TYPE(matrix), DIMENSION(SIZE(Y,1),SIZE(Y,2)) :: X
      REAL, DIMENSION(SIZE(Z,1),SIZE(Z,2)) :: W
      INTEGER :: i, j, k, n
        ! effettua una copia di riserva degli argomenti
         W(:,:) = Z(:,:)%elem
         X(:,:)%elem = Y(:,:)%elem
        ! eliminazione di Gauss sulla matrice argomento (W|X)
        n = SIZE(Z,2)
        DO k = 1,n-1
           DO i=k+1,n
              W(i,k) = W(i,k)/W(k,k)
              X(i,:)%elem = X(i,:)%elem - W(i,k) * X(k,:)%elem
           END DO
           DO j=k+1,n
              DO i=k+1,n
                 W(i,j) = W(i,j) - W(i,k) * W(k,j)
               END DO
           END DO
        END DO
        ! "back substitution" su X
```

```
            DO k = n,1,-1
               X(k,:)%elem = X(k,:)%elem / W(k,k)
               DO i=1,k-1
                  X(i,:)%elem = X(i,:)%elem - W(i,k) * X(k,:)%elem
               END DO
            END DO
      END FUNCTION matrix_div

      FUNCTION matrix_vector_div(Y,Z) RESULT(X)
   ! "Divide" un array 1D per un array 2D di tipo "matrix"
         TYPE(matrix), INTENT(IN), DIMENSION(:) :: Y
         TYPE(matrix), INTENT(IN), DIMENSION(:,:) :: Z
         TYPE(matrix), DIMENSION(SIZE(Y,1)) :: X
         REAL, DIMENSION(SIZE(Z,1),SIZE(Z,2)) :: W
         INTEGER :: i, j, k, n
           ! effettua una copia di riserva degli argomenti
           W(:,:) = Z(:,:)%elem
           X(:)%elem = Y(:)%elem
           ! eliminazione di Gauss sulla matrice argomento (W|X)
           n = SIZE(Z,2)
           DO k = 1,n-1
              DO i=k+1,n
                 W(i,k) = W(i,k)/W(k,k)
                 X(i)%elem = X(i)%elem - W(i,k) * X(k)%elem
              END DO
              DO j=k+1,n
                 DO i=k+1,n
                    W(i,j) = W(i,j) - W(i,k) * W(k,j)
                 END DO
              END DO
           END DO
          ! "back substitution" su X
           DO k = n,1,-1
              X(k)%elem = X(k)%elem / W(k,k)
              DO i=1,k-1
                 X(i)%elem = X(i)%elem - W(i,k) * X(k)%elem
              END DO
            END DO
      END FUNCTION matrix_vector_div

   END MODULE operatori
```

Il programma che segue mostra un semplice utilizzo di questo modulo e delle operazioni da esso

definite. In particolare, definite due matrici 3×3 `A` e `B` ed un vettore `c` lo scopo del programma è quello di eseguire le operazioni:

$A \times B$
$A \times c$
A^{-1}
$A/c = A^{-1} \times c$

```
PROGRAM prova_operatori
!
  USE operatori
  IMPLICIT NONE
  TYPE(matrix),DIMENSION(3,3) :: A, B, Unity
  TYPE(matrix),DIMENSION(3)   :: c
  TYPE(matrix),DIMENSION(3,3) :: AperB, Inv_A
  TYPE(matrix),DIMENSION(3)   :: Aperc, cdivA
  INTEGER :: i, j, dim
  REAL :: h
! Inizializzazione delle matrici
  DO i=1,3
     DO j=1,3
        Unity%elem=0.0
        CALL RANDOM_NUMBER(HARVEST=h)
        A(i,j)%elem = h+0.5
        CALL RANDOM_NUMBER(HARVEST=h)
        B(i,j)%elem = h-0.5
     END DO
     CALL RANDOM_NUMBER(HARVEST=h)
     c(i)%elem = h
  END DO
  FORALL (i=1:3)
     A(i,i)%elem = 2.0*SUM(ABS(A(i,:)%elem))  ! assicura la dominanza
                                              !  diagonale forte
     Unity(i,i)%elem = 1.0  ! Unity e' la matrice identica 3x3
  END FORALL

  AperB = A*B      ! ris: A*B righe x colonne
  Aperc = A*c      ! ris. A*c righe x colonne
  Inv_A = Unity/A  ! ris: INV(A)*Unity = INV(A)
  cdivA = c/A      ! ris: INV(A)*c

  dim=10
  PRINT 100, "Matrice A:"
  DO i=1,3
```

```
      WRITE(*,200) (A(i,j),j=1,3)
   END DO

   PRINT 100, "Matrice B:"
   DO i=1,3
      WRITE(*,200) (B(i,j),j=1,3)
     END DO

   PRINT 100, "Vettore c:"
   WRITE(*,*) (c(i),i=1,3)

   dim=21
   PRINT 100, "Matrice Prodotto A*B:"
   DO i=1,3
      WRITE(*,200) (AperB(i,j),j=1,3)
   END DO

   PRINT 100, "Vettore Prodotto A*c:"
   WRITE(*,200) (Aperc(i),i=1,3)

   PRINT 100, "Matrice Rapporto c/A:"
   WRITE(*,200) (cdivA(i),i=1,3)

   dim=13
   PRINT 100, "Inversa di A:"
   DO i=1,3
      WRITE(*,200) (Inv_A(i,j),j=1,3)
   END DO

100 FORMAT(/,1X,A<dim>,/1X,<dim>("="))
200 FORMAT(3(1X,F7.4))

END PROGRAM prova_operatori
```

Compilato con l'Intel Fortran Compiler (Ver. 4.5) e mandato in esecuzione su un PC con processore Intel Pentium III, il programma precedente fornisce il seguente output:

```
 Matrice A:
 ==========
  6.5269  0.6481  1.3421
  1.2688  6.3646  1.3090
  0.5735  1.3217  6.0345

 Matrice B:
```

```
==========
 0.3233  0.1567  0.1286
 0.1811 -0.2140  0.2437
 0.2361 -0.4603 -0.4364

Vettore c:
==========
 0.3305  0.7259  0.3565

Matrice Prodotto A*B:
====================
 2.5441  0.2664  0.4115
 1.8716 -1.7658  1.1426
 1.8496 -2.9707 -2.2376

Vettore Prodotto A*c:
====================
 3.1059  5.5060  3.3004

Matrice Rapporto c/A:
====================
 0.0337  0.1004  0.0339

Inversa di A:
=============
 0.1579 -0.0092 -0.0331
-0.0297  0.1663 -0.0295
-0.0085 -0.0355  0.1753
```

L'unico scotto che bisogna pagare in questo caso è quello di lavorare con array di tipo `matrix` piuttosto che con array di tipo `REAL` il che rende le operazioni di assegnazione un pò più pesanti dal punto di vista formale. Nulla vieta, tuttavia, di estendere anche il significato dell'operatore di assegnazione al fine di poter lavorare con array di tipo `matrix` esattamente come si lavorerebbe con comuni array reali. Come ciò possa essere fatto sarà chiarito nel prosieguo.

5.17.2 Definizione di nuovi operatori

Così come è possibile estendere il significato di operatori già *esistenti*, in Fortran è anche possibile definire *nuovi* operatori e ciò è particolarmente utile quando si usano tipi di dati *non predefiniti*. Tali operatori *definiti dall'utente* (*user defined*) devono avere un punto (".") all'inizio e alla fine del nome ed il loro effetto può essere definito attraverso una *function* avente uno o due argomenti *non* opzionali con l'attributo `INTENT(IN)`. Ad esempio, negli esempi

proposti al paragrafo precedente si sarebbero potuti definire un operatore ".PLUS." ed un operatore ".DIVIDE.", invece di estendere il significato degli operatori "+" e "/".

Il seguente esempio mostra la definizione di un operatore binario .DIST. che calcola la distanza tra due "punti" essendo questi ultimi variabili di un tipo di dati derivato. L'operatore in questione viene definito all'interno di un modulo sicché potrà essere usato da unità di programma diverse.

```
MODULE distance_mod
   ...
   TYPE point
      REAL :: x, y   ! ascissa e ordinata di point
   END TYPE point
   ...
   INTERFACE OPERATOR (.DIST.)
      MODULE PROCEDURE calcdist
   END INTERFACE
   ...
   CONTAINS
      ...
      FUNCTION calcdist (px,py)
         REAL :: calcdist
         TYPE (point), INTENT(IN) :: px, py
            calcdist = SQRT((px%x-py%x)**2 + (px%y-py%y)**2)
      END FUNCTION calcdist
      ...
END MODULE distance_mod
```

Un programma chiamante includerà, ad esempio, le istruzioni:

```
USE distance_mod
TYPE(point) :: px, py
...
distance = px.DIST.py
```

Si noti che l'operatore .DIST. appena definito è applicabile *esclusivamente* a variabili di tipo punto; sarebbe, pertanto, illegale un qualsiasi tentativo di applicare .DIST. a variabili di tipo diverso.

Un interessante esempio di operatore unario, invece, può essere il seguente. L'area di un triangolo note le coordinate dei vertici $P_1(x_1, y_1)$, $P_2(x_2, y_2)$ e $P_3(x_3, y_3)$ è, come noto, valutabile secondo la relazione:

$$Area = \frac{1}{2} |\det A|$$

essendo A la matrice delle coordinate dei vertici così assemblata:

$$A = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix}$$

Per il calcolo del determinante di A si può ricorrere alla nota *regola di Sarrus* che, si ricorda, è applicabile proprio alle matrici 3×3:

$$\det A = x_1y_2 + y_1x_3 + x_2y_3 - y_2x_3 - y_3x_1 - y_1x_2$$

Quindi, se si definiscono un tipo `punto` come:

```
TYPE punto
   REAL :: x  ! ascissa
   REAL :: y  ! ordinata
END TYPE punto
```

ed un tipo `triangolo` come:

```
TYPE triangolo
   TYPE(punto) :: P1  ! primo vertice
   TYPE(punto) :: P2  ! secondo vertice
   TYPE(punto) :: P3  ! terzo vertice
END TYPE triangolo
```

è possibile definire in maniera assolutamente semplice un operatore `.AREA.` che, applicato ad un oggetto di tipo `triangolo` ne restituisca l'area valutata secondo il procedimento anzidetto. Quanto asserito è efficacemente descritto dal seguente modulo:

```
MODULE defin_area
! Sezione dichiarativa
   IMPLICIT NONE
   TYPE punto
      REAL :: x  ! ascissa
      REAL :: y  ! ordinata
   END TYPE punto
   TYPE triangolo
      TYPE(punto) :: P1  ! primo vertice
      TYPE(punto) :: P2  ! secondo vertice
      TYPE(punto) :: P3  ! terzo vertice
   END TYPE
   TYPE(triangolo) :: mytr
   INTERFACE OPERATOR(.AREA.)
      MODULE PROCEDURE area
   END INTERFACE

   CONTAINS
      FUNCTION area(mytr)
! *** Sezione dichiarativa ***
         IMPLICIT NONE
```

```
! Parametri formali e tipo della funzione
        TYPE(triangolo),INTENT(IN) :: mytr
        REAL :: area  ! area del triangolo
! Variabili locali
        REAL,DIMENSION(3) :: vet1, vet2 ! vettori che ospiteranno le
                                        ! coordinate dei vertici
        REAL :: det   ! determinante della matrice delle coordinate
! *** Sezione esecutiva ***
! Assemblaggio dei vettori delle coordinate
           vet1=(/mytr%P1%x, mytr%P2%x, mytr%P3%x/)
           vet2=(/mytr%P1%y, mytr%P2%y, mytr%P3%y/)
! Valutazione dell'area del triangolo
          det= DOT_PRODUCT(vet1,CSHIFT(vet2,SHIFT=-1)) - &
               DOT_PRODUCT(vet2,CSHIFT(vet1,SHIFT=-1))
          area = 0.5*ABS(det)
      END FUNCTION area
END MODULE defin_area
```

Un esempio di applicazione del nuovo operatore è descritto nel seguente programma di test:

```
PROGRAM prova_triangolo
  USE defin_area
  IMPLICIT NONE
  TYPE(triangolo) mytr
  REAL :: x1, x2, y1, y2, x3, y3
! Sezione esecutiva
! Fase di input
  WRITE(*,*) "Coordinate del primo vertice: "
  WRITE(*,'(1X,A)',ADVANCE='NO') " ascissa:  "; READ(*,*) x1
  WRITE(*,'(1X,A)',ADVANCE='NO') " ordinata: "; READ(*,*) y1
  WRITE(*,*) "Coordinate del secondo vertice: "
  WRITE(*,'(1X,A)',ADVANCE='NO') " ascissa:  "; READ(*,*) x2
  WRITE(*,'(1X,A)',ADVANCE='NO') " ordinata: "; READ(*,*) y2
  WRITE(*,*) "Coordinate del terzo vertice: "
  WRITE(*,'(1X,A)',ADVANCE='NO') " ascissa:  "; READ(*,*) x3
  WRITE(*,'(1X,A)',ADVANCE='NO') " ordinata: "; READ(*,*) y3
  mytr = triangolo(punto(x1,y1),punto(x2,y2),punto(x3,y3))
! Applicazione dell'operatore .AREA. e output del risultato
  WRITE(*,'(1X,A)',ADVANCE='NO') "L'area del triangolo e': ",
  WRITE(*,'(1X,F7.3)') .AREA.mytr
  STOP
END PROGRAM prova_triangolo
```

Può giovare considerare un semplice esempio di utilizzo di questo programma:

```
C:\MYPROG>prova_triangolo
 Coordinate del primo vertice:
  ascissa:  5
  ordinata: 1
 Coordinate del secondo vertice:
  ascissa:  3
  ordinata: 3
 Coordinate del terzo vertice:
  ascissa:  0
  ordinata: -2
 L'area del triangolo e':    8.000
```

Naturalmente, sfruttando la definizione dei tipi di dati `punto` e `triangolo` e dell'operatore binario `.DIST.` è possibile definire in maniera assolutamente immediata anche un nuovo operatore unario `.PERIMETRO.` che applicato ad un oggetto `triangolo` restituisca la somma delle distanze fra i vertici.

E' da precisare che anche per gli operatori definiti dall'utente esistono *regole di precedenza*. A tal proposito, è bene tenere a mente che gli operatori *unari* definiti dall'utente hanno la precedenza *più alta* fra *tutti* gli operatori, mentre gli operatori *binari* definiti dall'utente hanno la precedenza *più bassa*. Un operatore definito dall'utente che abbia lo *stesso nome* di un operatore intrinseco conserverà la precedenza di quest'ultimo. A titolo di esempio, se `.OPUN.` e `.OPBIN.` sono due operatori *non* intrinseci, di cui il primo unario ed il secondo binario, allora la seguente espressione:

```
.OPUN.e**j/a.OPBIN.b+c.AND.d
```

risulta perfettamente equivalente a:

```
(((.OPUN.e)**j)/a).OPBIN.((b+c).AND.d)
```

5.17.3 *Overloading* dell'operatore di assegnazione

Quando si usano tipi di dati derivati può essere necessario estendere il significato dell'*operatore di assegnazione* ("="). Anche in questo caso è necessario un *interface block*, questa volta, però, relativo ad una *subroutine*.

Ad esempio, si supponga di lavorare con due variabili, `ax` e `px`, dichiarate al modo seguente:

```
REAL ::  ax
TYPE (punto) ::  px
```

e che nel programma sia necessaria la seguente assegnazione:

```
ax = px
```

cioè sia necessario assegnare un valore di tipo *reale* ad una variabile di tipo `punto`. Tale operazione è, normalmente, non valida a meno di non essere, in qualche modo, esplicitamente

definita. Ad esempio, si supponga che la variabile `ax` debba assumerne il valore del più grande fra le componenti `x` ed `y` di `px`. Questa assegnazione deve essere definita attraverso una subroutine con due argomenti non opzionali, il primo avente l'attributo `INTENT(OUT)` o `INTENT(INOUT)`, il secondo l'attributo `INTENT(IN)`, ed inoltre deve essere creato un *interface assignment block*.

L'*interface block* necessario per l'estensione dell'operatore di assegnazione ha la seguente forma:

```
INTERFACE ASSIGNMENT (=)
  corpo_dell'interfaccia
END INTERFACE
```

La definizione del nuovo operatore di assegnazione può essere efficacemente inserita in un modulo, come di seguito illustrato:

```
MODULE overloading_assegnazione
   IMPLICIT NONE
   TYPE point
      REAL :: x, y
   END TYPE point
   ...
   INTERFACE ASSIGNMENT(=)
      MODULE PROCEDURE assignnew
   END INTERFACE
CONTAINS
   SUBROUTINE assignnew(ax,px)
      REAL, INTENT(OUT) :: ax
      TYPE(point), INTENT(IN) :: px
         ax = MAX(px%x,px%y)
   END SUBROUTINE assignnew
   ...
END MODULE overloading_assegnazione
```

Un programma che invochi questo modulo attraverso l'associazione di `USE` può eseguire, così come richiesto, l'assegnazione (ora definita) tra una variabile di tipo `punto` e una variabile di tipo reale:

```
USE overloading_assegnazione
REAL :: ax
TYPE (point) :: px
...
ax = px
```

Ritornando per un attimo al modulo `operatori` visto a proposito dell'overloading degli operatori ed al tipo `matrix` in esso definito, un altro esempio di overloading dell'assegnazione potrebbe avere lo scopo di consentire di assegnare un array di tipo `RAEL` ad un array di tipo

`matrix` al fine di poter lavorare, all'esterno del modulo, con comuni array reali ma di passarli poi come array di tipo `matrix` alle procedure di modulo definenti nuovi operatori. Ciò può essere fatto aggiungendo al modulo `operatori` l'interfaccia seguente:

```
INTERFACE ASSIGNMENT(=)
   MODULE PROCEDURE matrix_from_real, matrix_from_matrix, &
                    vector_from_real, vector_from_vector
END INTERFACE
```

e le seguenti procedure:

```
!
! Operatori di assegnazione: X = Y
!
SUBROUTINE matrix_from_real(X,Y)
! Copia un array 2D di tipo REAL in un array 2D di tipo matrix
   REAL, INTENT(IN), DIMENSION(:,:) :: Y
   TYPE(matrix), INTENT(OUT), DIMENSION(SIZE(Y,1),SIZE(Y,2)) :: X
      X(:,:)%elem = Y(:,:)
END SUBROUTINE matrix_from_real

SUBROUTINE matrix_from_matrix(X, Y)
! Copia un array 2D di tipo matrix
   TYPE(matrix), INTENT(IN), DIMENSION(:,:) :: Y
   TYPE(matrix), INTENT(OUT), DIMENSION(SIZE(Y,1),SIZE(Y,2)) :: X
      X(:,:)%elem = Y(:,:)%elem
END SUBROUTINE matrix_from_matrix

SUBROUTINE vector_from_real(X, Y)
! Copia un array 1D di tipo REAL in un array 1D di tipo matrix
   REAL, INTENT(IN), DIMENSION(:) :: Y
   TYPE(matrix), INTENT(OUT), DIMENSION(SIZE(Y,1)) :: X
      X(:)%elem = Y(:)
END SUBROUTINE vector_from_real

SUBROUTINE vector_from_vector(X, Y)
! Copia un array 1D di tipo matrix
   TYPE(matrix), INTENT(IN), DIMENSION(:) :: Y
   TYPE(matrix), INTENT(OUT), DIMENSION(SIZE(Y,1)) :: X
      X(:)%elem = Y(:)%elem
END SUBROUTINE vector_from_vector
```

5.17.4 *Overloading* delle procedure intrinseche

Quando viene introdotto un nuovo tipo di dati può essere utile estendere l'applicabilità di alcune *procedure intrinseche* a variabili del nuovo tipo. In questo caso si definisce una *generic interface* con lo *stesso nome* della procedura intrinseca già esistente. Una procedura intrinseca potrà essere così "estesa" e sarà questa procedura ad essere richiamata (col nome usuale) quando la variabile argomento sia del nuovo tipo di dati.

Ad esempio, si supponga di voler estendere il significato della funzione intrinseca `LEN_TRIM` affinché fornisca il numero di lettere che compongono il nome di `proprietario` quando applicata ad un oggetto di tipo `casa`. Il seguente modulo illustra come ciò possa essere fatto:

```
MODULE mio_modulo
   IMPLICIT NONE
   TYPE casa
      CHARACTER(LEN=20) :: proprietario
      INTEGER :: residenti
      REAL :: valore
   END TYPE casa
   INTERFACE LEN_trim
      MODULE PROCEDURE lettere
   END INTERFACE
   CONTAINS
      FUNCTION lettere(c)
         TYPE(casa), INTENT(IN) :: c
         INTEGER :: lettere
            lettere = LEN_TRIM(c%proprietario)
      END FUNCTION lettere
END MODULE mio_modulo
```

5.18 Visibilità

La *visibilità* (*scope*) di una entità con nome (variabile o procedura) o di un'etichetta è, per definizione, quella parte di un programma all'interno della quale quel nome o quell'etichetta sono unici. In altre parole, la visibilità di un'entità con nome o di un etichetta è l'insieme di tutte le *unità di visibilità* "non sovrapposte" in cui quel nome o quella etichetta possono essere usati senza ambiguità.

Una *unità di visibilità* è uno qualsiasi dei seguenti ambienti:

- Una definizione di un tipo di dati derivato.
- Un *interface block*, esclusi ogni definizione di tipo di dati derivati o *interface block* al suo interno.
- Un'unità di programma o procedura interna, esclusi ogni definizione di tipo di dati derivati, procedure o *interface block* al suo interno.

Tutte le variabili, i tipi di dati, le etichette, i nomi di procedura, etc. all'interno della stessa unità di visibilità devono avere nomi differenti. Le entità, invece, aventi lo stesso nome ma che si trovino in differenti unità di visibilità, sono sempre entità separate l'una dall'altra.

5.18.1 Visibilità di una etichetta

Ogni programma e ogni procedura, sia essa interna o esterna, ha un *proprio* set di etichette (in sostanza, quelle delle istruzioni `FORMAT`). La visibilità di un'etichetta è l'*intero* programma principale o l'*intera* procedura, ad eccezione delle eventuali procedure interne. Pertanto è possibile per la stessa etichetta apparire in *differenti* unità di uno stesso programma senza alcuna ambiguità.

5.18.2 Visibilità di un nome

Del tutto in generale si può dire che i nomi sono detti accessibili o per *host association* o per *use association*:

- *Host association* - La visibilità di un nome dichiarato in un'unità di programma si estende dalla "testa" dell'unità di programma fino alla istruzione `END`.
- *Use association* - La visibilità di un nome dichiarato in un modulo si estende a qualsiasi unità di programma che usi quel modulo.

Si noti che nessuno dei due tipi di associazione si estende alle eventuali procedure esterne invocate, ed inoltre essi non includono le procedure interne in cui il nome venga ridichiarato. Il significato delle precedenti definizioni è meglio chiarito se si discutono, caso per caso, tutte le possibilità connesse alla visibilità di un nome in una unità di programma.

- La visibilità di un nome (ad esempio di una variabile) dichiarato in una unità di programma si estende dall'inizio dell'unità di programma fino all'istruzione `END`.
- La visibilità di un nome dichiarato in un programma principale o in una procedura esterna si estende a tutte le procedure interne a meno che non venga ridefinito nella procedura stessa.
- La visibilità di un nome dichiarato in una procedura interna è limitata alla procedura stessa.
- La visibilità di un nome dichiarato in un modulo si estende a tutte le unità di programma che usino quel modulo, fatta eccezione per quei casi in cui una procedura interna ridichiara quel nome.
- I nomi delle unità di programma sono globali e pertanto devono essere unici. Il nome di un'unità di programma deve, inoltre, essere differente da quello di tutte le entità locali a quella unità.

- Il nome di una procedura interna si estende a tutta l'unità di programma ospite. Inoltre, tutte le procedure interne ad una stessa unità di programma devono avere nomi differenti.

Il seguente frammento fornisce un esempio dei diversi livelli delle unità di visibilità:

```
MODULE scope1              ! scope 1
   ...                     ! scope 1
CONTAINS                   ! scope 1
   SUBROUTINE scope2()     ! scope 2
      TYPE scope3          ! scope 3
         ...               ! scope 3
      END TYPE scope3      ! scope 3
      INTERFACE            ! scope 3
         ...               ! scope 4
      END INTERFACE        ! scope 3
      REAL :: a, b         ! scope 3
10       ...               ! scope 3
   CONTAINS                ! scope 2
      FUNCTION scope5()    ! scope 5
         REAL :: b         ! scope 5
            b = a+1        ! scope 5
            ...            ! scope 5
      END FUNCTION         ! scope 5
   END SUBROUTINE          ! scope 2
END MODULE                 ! scope 1
```

A titolo di esempio si riporta il seguente programma che illustra la caratteristica di visibilita' di una struttura in una procedura interna:

```
PROGRAM convert
! Questo tipo e' visibile "anche" dalle procedure interne
   TYPE :: polar
      REAL :: r, theta
   END TYPE polar
!
   WRITE(*,*) complex_to_polar(CMPLX(-SQRT(2.0),-SQRT(2.0)))
   STOP
CONTAINS
   FUNCTION complex_to_polar(z) RESULT(s)
! Queste variabili sono visibili "solo" dalla procedura interna
      COMPLEX, INTENT(IN) :: z
      TYPE(polar) :: s
         s%r = ABS(z)
         s%theta = ATAN2(AIMAG(z),REAL(z))
         RETURN
```

```
    END FUNCTION complex_to_polar
END PROGRAM convert
```

Si vuole concludere il capitolo con esempio di programma abbastanza "completo" e che riassuma buona parte dei concetti descritti nelle pagine precedenti. Il codice che segue è, in un certo senso, una riscrittura del programma già proposto in precedenza per il calcolo della radice di una funzione con il metodo di Newton-Raphson. In questo caso, oltre a modificare l'algoritmo per rendere ricorsiva l'esecuzione della routine, si prevede anche la possibilità che la derivata della funzione argomento non sia disponibile. In questo caso, non potendosi applicare direttamente la formula iterativa

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

si dovrà ricorrere al cosiddetto *metodo delle secanti* il quale discretizza la derivata $f'(x)$ secondo la relazione:

$$f'(c) = \frac{f(b) - f(a)}{b - a}$$

in cui a e b rappresentano gli estremi di un intervallo per il quale passa la generica retta secante la funzione, mentre il termine c rappresenta l'intersezione della retta secante passante per i punti $(a, f(a))$ e $(b, f(b))$ con l'asse delle ascisse. Così come per il metodo di Newton-Raphson, la successione dei valori assunti da c secondo la formula iterativa:

$$c = a - f(a) \times \left(\frac{b - a}{f(b) - f(a)} \right)$$

converge, entro una certa ammissibile tolleranza, alla radice della funzione nell'intervallo specificato.

```
MODULE RootFinders
! Massimo errore consentito nell'individuazione della radice
   REAL,PARAMETER :: default_tolerance = EPSILON(1.0)
! Restringe la visibilita' delle funzioni di modulo...
   PRIVATE secant, newton
! ... per cui secant e newton potranno essere invocate
! soltanto all'interno del modulo
CONTAINS
! Usa il metodo delle secanti per trovare una radice di f se df,
! la derivata di f, non e' disponibile, altrimenti, usa il metodo
! di Newton. Gli estremi "a" e "b" sono usati come intervallo di
! partenza used dal metodo delle secanti. La media aritmetica di
! "a" e "b" è usata come punto iniziale dal metodo di Newton
   FUNCTION findRoot(a,b,f,df,tolerance)
! *** Sezione dichiarativa ***
      IMPLICIT NONE
! Tipo della funzione
      REAL :: findRoot
```

```
! Parametri formali
      REAL,INTENT(IN) :: a, b
      REAL,OPTIONAL,INTENT(IN) :: tolerance
! Interface block per le funzioni esterne argomento
      INTERFACE
         FUNCTION f(x)
            IMPLICIT NONE
            REAL :: f
            REAL,INTENT(IN) :: x
         END FUNCTION f
         FUNCTION df(x)
            IMPLICIT NONE
            REAL :: df
            REAL,INTENT(IN) :: x
         END FUNCTION df
      END INTERFACE
      OPTIONAL df
! Variabili locali
      REAL :: tol
! *** Sezione esecutiva ***
! Inizializzazione di tol
         IF (PRESENT(tolerance)) THEN
            tol = tolerance
         ELSE
            tol = default_tolerance
         END IF
! Scelta del metodo da impiegare
         IF (PRESENT(df)) THEN   ! Usa il metodo di Newton
            findRoot = newton((a+b)/2.,f,df,tol)
         ELSE                    ! Usa il metodo delle secanti
            findRoot = secant(a,b,f,tol)
         END IF
   END FUNCTION findRoot

   RECURSIVE FUNCTION secant(a,b,f,tol) RESULT(root)
! *** Sezione dichiarativa ***
      IMPLICIT NONE
! Tipo della funzione
      REAL :: root
! Parametri formali
      REAL, INTENT(IN) :: a, b, tol
! Interface block per la funzione esterna argomento
      INTERFACE
```

```
          FUNCTION f(x)
             IMPLICIT NONE
             REAL :: f
             REAL,INTENT(IN) :: x
          END FUNCTION f
       END INTERFACE
! Variabili locali
       REAL :: c   ! l'intercetta sull'asse x della linea secante
       REAL :: fa, fb, fc   ! i valori f(a), f(b) e f(c), rispettivamente
!
! *** Sezione esecutiva ***
! Inizializzazione di fa e fb
          fa = f(a); fb = f(b)
! Calcolo di c, l'intercetta sull'asse x della linea secante
! passante per i due punti (a,f(a)) e (b,f(b))
          c = a - fa*((b-a)/(fb-fa))
! Calcolo del valore della funzione in questo punto
          fc = f(c)
! Check su f(c). Il punto c e' radice se f(c) e' minore della tolleranza.
! NB: Qualora l'errore di round-off eccedesse la tolleranza, l'esame di
! questa condizione determinerebbe l'insorgere di un loop infinito
          IF ((ABS(fc)<=tol) .OR. ((ABS(c-b)<=tol))) THEN   ! Radice trovata
             root = c
          ELSE   ! Continua le iterazioni
! Check per verificare che la funzione non restituisca un risultato di
! valore assoluto crescente ad ogni sua chiamata ricorsiva
             IF (ABS(fa)<ABS(fb)) THEN   ! Usa a e c
                root = secant(a,c,f,tol)
             ELSE   ! Usa b e c
                root = secant(b,c,f,tol)
             END IF
          END IF
    END FUNCTION secant

    RECURSIVE FUNCTION newton(guess,f,df,tol) RESULT(root)
! *** Sezione dichiarativa ***
       IMPLICIT NONE
! Tipo della funzione
       REAL :: root
! Parametri formali
       REAL,INTENT(IN) :: guess, tol
! Interface block per la funzione esterna argomento
       INTERFACE
```

```
            FUNCTION f(x)
               IMPLICIT NONE
               REAL :: f
               REAL,INTENT(IN) :: x
            END FUNCTION f
            FUNCTION df(x)
               IMPLICIT NONE
               REAL df
               REAL,INTENT(IN) :: x
            END FUNCTION df
         END INTERFACE
! Variabili locali
         REAL :: fGuess, dfGuess   ! f(guess) e df(guess) rispettivamente
         REAL :: newGuess
! *** Sezione esecutiva ***
! Calcola df(guess) e f(guess)
            fGuess = f(guess); dfGuess = df(guess)
! Check su f(c). Il punto c e' radice se f(c) e' minore della tolleranza.
! NB: Qualora l'errore di round-off eccedesse la tolleranza, l'esame di
! questa condizione determinerebbe l'insorgere di un loop infinito
            IF (ABS(fGuess)<=tol) THEN   ! Radice trovata
               root = guess
            ELSE   ! Continua le iterazioni
               newGuess = guess-fGuess/dfGuess
               root = newton(newGuess,f,df,tol)
            END IF
      END FUNCTION newton
END MODULE RootFinders
```

Si fa notare la grande versatilità della funzione di modulo `findRoot` dovuta alla presenza di due argomenti opzionali. Così, ad esempio, se essa fosse invocata con l'istruzione:

```
x = findRoot(a,b,g,dg)
```

userebbe il valore di default per la tolleranza e chiamerebbe la routine `newton` vista la presenza del termine `dg` che rappresenta la derivata di `g`. Per fornire, invece, esplicitamente la tolleranza si può ricorrere ad una istruzione di chiamata del tipo:

```
x = findRoot(a,b,g,dg,1.0E-10)
```

Quando la derivata della funzione argomento non è disponibile, un'istruzione di chiamata del tipo:

```
x = findRoot(a,b,g)
```

userebbe il valore di default per la tolleranza e chiamerebbe la routine `secant` in quanto non viene passata alcuna derivata. Infine, un'invocazione di funzione del tipo seguente:

```
x = findRoot(a,b,g,tolerance=1.0E-10)
```

serve ancora ad invocare la funzione `secant`, ma con un valore della tolleranza ben preciso. Si noti che, in quest'ultimo caso, se non si fosse usata la parola chiave `tolerance` si sarebbe presentata una condizione ambigua di errore in quanto il valore `1.0E-10` sarebbe stato associato al termine `df` (che nella lista dei parametri formali di `findRoot` occupa la quarta posizione) mentre per la tolleranza si sarebbe erroneamente utilizzato il valore di default.

Di seguito si riporta un programma per testare il modulo `RootFinders`, una funzione di prova di cui valutare la radice e la sua derivata:

```
PROGRAM Test_Root
! Scopo: verificare l'efficacia delle funzioni definite
! nel modulo RootFinders
  USE RootFinders
  IMPLICIT NONE
  REAL a, b
  REAL, PARAMETER :: tol = 1.0E-6
  INTERFACE
     FUNCTION f(x)
        IMPLICIT NONE
        REAL :: f
        REAL,INTENT(IN) :: x
     END FUNCTION f
     FUNCTION df(x)
        IMPLICIT NONE
        REAL :: df
        REAL, INTENT(IN) :: x
     END FUNCTION df
  END INTERFACE
!
  PRINT*, "Inserisci gli estremi sinistro e destro &
           &dell'intervallo di tentativo: "
  READ(*,*) a, b
  PRINT*, "Metodo di Newton - La radice di f vale:     ", findRoot(a,b,f,df)
  PRINT*, "Metodo delle secanti - La radice di f vale: ", findRoot(a,b,f)
END PROGRAM Test_Root

FUNCTION f(x)
! Funzione esterna di cui valutare la radice
   IMPLICIT NONE
   REAL :: f
   REAL,INTENT(IN) :: x
      f = x+EXP(x)
END FUNCTION f
```

```
FUNCTION df(x)
! Derivata della funzione esterna
   IMPLICIT NONE
   REAL :: df
   REAL,INTENT(IN) :: x
      df = 1.0+exp(x)
END FUNCTION df
```

Per completezza si riporta anche un esempio di impiego del programma:

```
C:\MYPROG>test_root
 Inserisci gli estremi sinistro e destro dell'intervallo:
0.5  1.0
 Metodo di Newton - La radice di f vale:      -0.5671433
 Metodo delle secanti - La radice di f vale:  -0.5671433
```

Capitolo 6

Array Processing

6.1 Le diverse classi di array

Esistono tre possibili tipologie di array che differiscono tra loro a seconda del modo in cui essi sono "associati" allo spazio in memoria ad essi relativo.

- *Array statici*: la loro ampiezza è fissata in fase di dichiarazione e *non* può in alcun modo essere modificata durante l'esecuzione del programma. Questa forma di array è la meno flessibile nella maggior parte delle applicazioni in quanto per adattare un programma a nuove esigenze in termini di spazio in memoria è necessario intervenire sul codice sorgente, modificare le dimensioni degli array e ricompilare il programma. L'unico modo per ovviare a questo inconveniente è quello di sovradimensionare gli array, soluzione questa non certo ottimale in quanto a gestione della memoria.

  Nel Fortran 90/95 tali array sono detti *array di forma esplicita*

- *Array semi-dinamici*: l'ampiezza di tali array, che possono essere dichiarati *esclusivamente* in una procedura, è determinata in fase di esecuzione quando il comando del programma "entra" nella suddetta procedura. In tal modo è possibile creare array di dimensioni tali da adattare il programma alle esigenze attuali.

  Nel Fortran 90/95 tali array sono chiamati *array fittizi di forma presunta* (se sono *parametri formali*), o *array automatici* (se, invece, rappresentano *variabili locali*).

- *Array dinamici*: l'ampiezza e, con essa, l'impegno in termini di memoria associato a tali array *possono* essere alterati durante l'esecuzione del programma. Questa è la forma di array più flessibile e potente ma soffre di due inconvenienti: la lentezza in fase di esecuzione associata al loro uso e la mancanza di un controllo che individui, in fase di compilazione, eventuali errori dovuti a *limiti fuori range*.

  Nel Fortran 90/95 tali array sono detti *array allocabili*.

Nei prossimi paragrafi queste diverse forme di array saranno analizzate in dettaglio.

6.2 Array di forma esplicita

Un *array di forma esplicita* (*explicit-shape array*) è un array che si caratterizza per avere estensioni *esplicitamente* specificate per ciascuna delle sue dimensioni già all'atto della istruzione di dichiarazione di tipo. In aggiunta, esso può, opzionalmente, avere limiti inferiori dichiarati per alcune o tutte le sue dimensioni.

Quando un array di forma esplicita viene dichiarato, ciascuna dimensione viene specificata come:

```
[limite_inferiore]:limite_superiore
```

in cui il *limite_inferiore* vale uno per default.

Ad esempio, le seguenti istruzioni:

```
INTEGER, DIMENSION(10,10,10) ::  m
INTEGER, DIMENSION(-3:6,4:13,0:9) ::  k
```

dichiarano due array, `m` e `k`, entrambi di forma esplicita con un rango pari a `3`, un'ampiezza `1000`, e forma `(/10,10,10/)`. L'array `m` usa il limite inferiore di default (`1`) per ciascuna dimensione, per cui nella sua dichiarazione compaiono soltanto i limiti superiori. Per ciascuna dimensione di `m`, invece, sono specificati entrambi i limiti.

Un array di forma esplicita può essere dichiarato sia in un programma principale che in una procedura.

I limiti inferiore e superiore di una dimensione di un'array di forma esplicita che sia dichiarato all'interno di una procedura possono essere specificati anche a mezzo di variabili o espressioni le quali verranno valutate ogni qualvolta il controllo del programma entri nella procedura. Successive assegnazioni di queste variabili *non* avranno, tuttavia, alcun effetto sui limiti degli array. Ad esempio, data la seguente subroutine:

```
SUBROUTINE esempio(n,r1,r2)
   IMPLICIT NONE
   DIMENSION(n,5) :: a
   DIMENSION(10*n) :: b
      ...
      n = r1+r2
      ...
END SUBROUTINE esempio
```

quando essa viene invocata, gli array `a` e `b` verranno dimensionati in relazione al valore attuale del parametro formale `n`. La successiva istruzione di assegnazione che modifica il valore di `n` non avrà alcun effetto sulle dimensioni degli array `a` e `b`.

Un *explicit-shape array* che venga dimensionato con variabili o espressioni *deve* necessariamente essere un argomento formale di una procedura oppure il risultato di una funzione oppure deve essere un *array automatico*. Un array automatico è un particolare array di forma esplicita dichiarato in una procedura come variabile *locale* e con limiti *non* costanti. Gli array `a` e `b`

dell'esempio precedente sono array automatici. A questa classe di array sarà dedicato uno dei prossimi paragrafi.

Un aspetto importante da osservare è che, quando degli array di forma esplicita vengono "passati" in istruzioni di chiamata di procedura, *non* è necessario che i *limiti* degli array corrispondenti nelle liste dei parametri attuali e formali coincidano perfettamente, essendo necessario unicamente una coincidenza delle *estensioni* lungo ciascuna dimensione. Ad esempio, considerata la seguente subroutine:

```
SUBROUTINE explicit(a,b,m,n)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: m, n
   REAL, DIMENSION(m,n*n+1), INTENT(INOUT) :: a
   REAL, DIMENSION(-n:n,m,INT(m/n)), INTENT(OUT) :: b
   ...
END SUBROUTINE explicit
```

il seguente frammento di procedura chiamante è perfettamente valido:

```
REAL, DIMENSION(15,50) :: p
REAL, DIMENSION(15,15,2) :: q
...
CALL explicit(p,q,15,7)
```

sebbene soltanto il primo array, `p`, coincida in estensione e limiti (`1:15` e `1:50`) con il parametro formale `a`, mentre per quanto riguarda il secondo array, `q`, oltre alle estensioni coincidono soltanto i limiti delle ultime due dimensioni.

Una delle maggiori difficoltà associate all'uso di array di forma esplicita in chiamate di procedura è rappresentato dalla necessità di fornire informazioni necessarie alla valutazione dei limiti degli array; tipicamente ciò viene fatto, come visto, incrementando la lista dei parametri formali. Una soluzione più efficace ed elegante è quella di utilizzare array di *forma presunta*, come sarà spiegato nel prossimo paragrafo.

6.3 Array fittizi di forma presunta

Un *array fittizio di forma presunta* (*assumed shape array*) è un particolare array la cui *forma non* è nota in fase di compilazione, bensì gli viene attribuita a mezzo di una istruzione di chiamata di procedura.

Quando un array di forma presunta viene dichiarato, ciascuna dimensione viene specificata come:

`[`*`limite_inferiore`*`]:`

in cui il *`limite_inferiore`* vale uno per default.

Gli *assumed shape array* rendono possibile il trasferimento di array tra diverse unità di programma senza la necessità di doverne passare, nella lista degli argomenti, anche le dimensioni.

Tuttavia, se un tale array è l'argomento di una *procedura esterna* allora è necessario fornire un *interface block* nell'unità di programma chiamante.

Ad esempio, considerata la seguente procedura esterna definente gli array fittizi di forma presunta `arr1`, `arr2`, `arr3`:

```
SUBROUTINE sub(arr1,arr2,arr3)
   IMPLICIT NONE
   REAL, DIMENSION(:,:), INTENT(IN) :: arr1, arr2
   REAL, DIMENSION(0:,2:), INTENT(OUT) :: arr3
   ...
END SUBROUTINE sub
```

una unità di programma chiamante sintatticamente valida potrebbe essere la seguente:

```
PROGRAM prova_sub
  REAL, DIMENSION (0:9,10) :: arr1    ! forma (/10,10/)
  INTERFACE
     SUBROUTINE sub(arr1,arr2,arr3)
        IMPLICIT NONE
        REAL, DIMENSION(:,:), INTENT(IN) :: arr1, arr2
        REAL, DIMENSION(0:,2:), INTENT(OUT) :: arr3
     END SUBROUTINE sub
  END INTERFACE
  ...
  CALL SUB (arr1,arr2(1:5,3:7),arr3(0:4,2:6))
  ...
END PROGRAM prova_sub
```

6.4 Array automatici

Un *array automatico* (*automatic array*) è uno speciale tipo di *explicit-shape array* che può essere dichiarato *soltanto* in una *procedura* come *oggetto locale* ma *non* come parametro formale ed inoltre deve essere caratterizzato dall'avere *almeno uno* dei limiti di un indice *non* costante. Lo spazio in memoria per gli elementi di un array automatico è allocato dinamicamente quando l'esecuzione del programma entra nella procedura, ed è rimosso automaticamente all'atto dell'uscita dalla procedura. Durante l'esecuzione della procedura nella quale esso è dichiarato, un array automatico può essere usato allo stesso modo di un qualsiasi altro array, in particolare può essere passato come parametro attuale in una ulteriore chiamata di procedura.

Si guardi il seguente esempio:

```
SUBROUTINE mia_sub(x,y,n)
   IMPLICIT NONE
! Parametri Formali
   INTEGER, INTENT(IN) :: n
```

```
   REAL, DIMENSION(n), INTENT(INOUT) :: x     ! explicit-shape
   REAL, DIMENSION(:), INTENT(INOUT) :: y     ! assumed-shape
! Variabili Locali
   REAL, DIMENSION(SIZE(y,1)) :: a            ! automatico
   REAL, DIMENSION(n,n) :: b                  ! automatico
   REAL, DIMENSION(10) :: c                   ! explicit-shape
      ...
END SUBROUTINE mia_sub
```

In questa procedura, gli array `a` e `b` *non* sono parametri formali, tuttavia i limiti dei loro indici *non* sono costanti (in particolare, il limite superiore di `a` dipende dalla *forma* del parametro formale `y`, mentre entrambi i limiti di `b` sono pari al parametro formale `n`). Dunque, le variabili (*locali*) `a` e `b` sono array automatici, mentre l'array `c`, avendo dimensioni costanti, non lo è.

Può giovare riportare un secondo esempio:

```
PROGRAM stringhe
  CHARACTER(LEN=20), PARAMETER :: a="Questo e' un esempio"
  PRINT*, double(a)
CONTAINS
   FUNCTION double(a)
      CHARACTER(LEN=*), INTENT(IN) :: a
      CHARACTER(LEN=2*LEN(a))      :: double
         double=a//a
   END FUNCTION double(a)
END PROGRAM stringhe
```

Gli array automatici sono utili quando in una procedura è richiesto, in forma *temporanea*, spazio in memoria per un array la cui forma non è nota a priori. Per tale ragione gli array automatici sono spesso chiamati *array di lavoro*. In tali situazioni l'uso degli array automatici può notevolmente semplificare la sequenza di chiamata di una procedura eliminando la necessità di passare, come argomenti, array di forma talvolta "complicata". Inoltre, gli array automatici *possono* essere passati come argomenti ad altre procedure che vengano invocate dalla procedura all'interno della quale detti array sono dichiarati.

E' doveroso sottolineare che *non* è possibile inizializzare gli array automatici nelle istruzioni di dichiarazione di tipo, ma è possibile farlo nelle istruzioni di assegnazione all'inizio della procedura in cui gli array vengono creati. Inoltre, dal momento che un array automatico è, per definizione, un oggetto temporaneo, *non* è possibile specificare per esso l'attributo `SAVE`. Tuttavia, gli array automatici *possono* essere passati come argomenti ad altre procedure che vengano invocate dalla procedura all'interno della quale detti array sono dichiarati.

Un ulteriore problema associato agli array automatici è che non esiste alcun modo per sapere se un tale array sia stato creato con successo o meno. Infatti, se non c'è sufficiente memoria per allocare un array automatico, il programma abortirà in maniera piuttosto subdola. In caso di dubbio circa la capacità del processore di creare un array di dimensioni considerevoli, è bene optare per la dichiarazione dei più complessi array allocabili i quali, al contrario, prevedono un meccanismo di *check* riguardo la riuscita dell'operazione di allocazione.

6.5 Array allocabili

Nel paragrafo precedente si è visto come gli array automatici possano fornire una soluzione al problema della creazione di array *temporanei* le cui dimensioni possano essere determinate soltanto *durante* l'esecuzione del programma. Una soluzione più "completa" è fornita, però, dagli *allocatable array*. Questi ultimi sono più flessibili degli array automatici poiché le operazioni di *allocazione* e di *deallocazione* dei loro elementi avvengono sotto il completo controllo del programmatore (per gli array automatici, invece, lo spazio in memoria è sempre allocato all'atto della chiamata di procedura, ed è sempre rilasciato all'uscita dalla procedura).

L'uso degli array allocabili consiste, generalmente, di tre fasi:

1. L'array allocabile è specificato in una *istruzione di dichiarazione di tipo*.

2. Lo spazio in memoria è dinamicamente allocato per i suoi elementi in una *istruzione di allocazione* separata, dopodiché l'array può essere normalmente usato.

3. Quando l'array non è più necessario, lo spazio per i suoi elementi viene rilasciato attraverso una speciale *istruzione di deallocazione*.

Una volta che per esso sia stato allocato dello spazio in memoria, un array allocabile può essere utilizzato come un comune array: i suoi elementi possono essere assegnati e riferiti, o anche passati come *parametri attuali* in istruzioni di chiamata di procedura. Inoltre un array allocabile *non* può essere dichiarato come *parametro formale* di una procedura.

Un *allocatable array* è dichiarato in una istruzione di dichiarazione di tipo che includa un *attributo* `ALLOCATABLE`. Tuttavia, dal momento che le sue dimensioni *non* sono note all'atto della dichiarazione, il limite di ciascuno dei suoi indici va sostituito con il simbolo dei due punti:

```
tipo, ALLOCATABLE, DIMENSION(:  [, :, ...])  ::  nome_array
```

(il rango, chiaramente, è noto dal numero di ricorrenze del simbolo dei due punti).

Si noti come l'istruzione di dichiarazione di tipo per un array allocabile sia molto simile a quella di un *array fittizio di forma presunta*. Tuttavia, l'analogia è soltanto formale poiché mentre un *assumed-shape array deve* essere un *parametro formale* di una procedura e *non* può essere dichiarato con l'attributo `ALLOCATABLE`, l'esatto contrario vale per un *allocatable array*.

Così come per altri attributi, anche l'attributo `ALLOCATABLE` può essere sostituito da una *istruzione* `ALLOCATABLE` separata:

```
tipo ::  nome_array
ALLOCATABLE ::  nome_array(:  [, :, ...])
```

E' sempre preferibile, tuttavia, dichiarare un array allocabile con l'*attributo* `ALLOCATABLE` piuttosto che a mezzo dell'*istruzione* `ALLOCATE` per ragioni di chiarezza e concisione: è sempre opportuno fare in modo, quando possibile, che *tutte* le informazioni circa una *stessa* entità siano contenute in un'*unica* istruzione.

Gli array allocabili sono spesso chiamati *deferred-shape array* poiché, quando essi vengono dichiarati, la loro estensione lungo ciascuna dimensione non viene specificata, bensì *rimandata* (*deferred*) ad una fase successiva.

Una cosa importante da osservare è che, contrariamente a quanto avviene con tutte le altre forme di array, la dichiarazione di un array allocabile *non* alloca spazio alcuno per l'array sicché *non* è possibile usare detto array fino a che esso non venga allocato. Difatti, un array allocabile è detto avere uno *stato di associazione* per cui, fintanto che non venga allocato spazio in memoria per i suoi elementi, il suo stato di allocazione è detto *correntemente non allocato* o, più semplicemente, *non allocato*.

Un array *non allocato* può essere allocato a mezzo dell'istruzione `ALLOCATE` che ha la funzione di riservare spazio in memoria per i suoi elementi. La sintassi dell'istruzione di allocazione è la seguente:

`ALLOCATE(`*`lista_di_specificazioni_di_array`* `[, STAT=`*`variabile_di_stato`*`])`

in cui ogni *`specificazione_di_array`* consiste nel *nome* dell'array allocabile seguito dai *limiti* degli indici, per ciascuna dimensione, racchiusi tra parentesi. La clausola opzionale:

`STAT=`*`variabile_di_stato`*

fa sì che il processore sia in grado di fornire indicazioni riguardo il successo o meno dell'operazione di allocazione, allo stesso modo in cui la clausola `IOSTAT=`*`variabile_di_stato`* informa circa la riuscita o meno di una operazione di I/O su file. Se l'allocazione dell'array ha avuto successo *`variabile_di_stato`* assumerà valore nullo, in caso contrario essa assumerà un valore positivo dipendente dal processore. Si noti che, pur essendo opzionale, l'uso della clausola `STAT=` è sempre consigliabile poiché, in sua assenza, un eventuale fallimento nell'allocazione di un array comporterebbe un inevitabile arresto del programma. A titolo di esempio, si guardi la seguente istruzione:

```
ALLOCATE(arr1(20),arr2(10:30,-10:10),arr3(20,30:50,10),STAT=error)
```

Essa alloca tre vettori di cui uno (`arr1`) monodimensionale di `20` elementi, un altro (`arr2`) bidimensionale di `21`×`21` elementi, l'ultimo (`arr3`) tridimensionale di `20`×`21`×`10` elementi. Si noti che in una stessa istruzione `ALLOCATE` possono essere allocati array non solo di diversa forma e dimensioni ma anche di tipo differente, come mostrato dal seguente frammento di codice:

```
INTEGER, ALLOCATABLE :: matrix(:,:)
REAL, ALLOCATABLE    :: vector(:)
...
ALLOCATE(matrix(3,5),vector(-2:N+2))
```

Il programma che segue mostra un esempio di impiego di array allocabili. Lo scopo del programma è quello di verificare se un valore reale ed un vettore siano o meno un autovalore ed il corrispondente autovettore di una matrice quadrata assegnata. Si ricorda che, detta A una

matrice quadrata un valore scalare λ è detto *autovalore* di A se esiste un vettore non nullo u tale che

$$Au = \lambda u$$

Tale vettore u è chiamato *autovettore* di A associato a λ.

```
PROGRAM eigen
  IMPLICIT NONE
! Programma per testare gli autovalori e gli autovettori
! di una matrice A di tipo intero
  REAL, ALLOCATABLE, DIMENSION (:,:) :: A
  REAL, ALLOCATABLE, DIMENSION (:) :: u
  REAL, ALLOCATABLE, DIMENSION (:) :: zero
  REAL :: lamda
  REAL, PARAMETER :: eps=0.001
  INTEGER :: i, n, stat
  LOGICAL :: correct=.TRUE.,answer=.TRUE.
!
  PRINT*, "Dimensione di A: "
  READ(*,*) n
  ALLOCATE(A(1:n,1:n),u(1:n),zero(1:n),STATUS=stat)
  IF (stat/=0)
     PRINT*, "Allocazione degli array fallita!"
     PRINT*, "Codice dell'errore: ", stat
     STOP
  END IF
  PRINT*, "Introdurre A per righe: "
  DO i=1,n
     READ(*,*) A(i,1:n)
  END DO
  PRINT*, "Introdurre un autovalore di A: "
  READ(*,*) lamda
  PRINT*, "Introdurre il corrispondente autovettore: "
  READ(*,*) u(1:n)
  PRINT*, "A: "
  DO i=1,n
     PRINT*, A(i,1:n)
  END DO
  FORALL (i=1:n)
     A(i,i) = A(i,i) - lamda
  END FORALL
  zero = MATMUL(A,u)
  i = 1
  DO WHILE (correct.AND.i<=n)
```

```
      IF(ABS(zero(i))>eps) THEN
         correct = .FALSE.
      END IF
      i = i+1
   END DO
   IF(correct) THEN
      PRINT*, " lamda = ", lamda
      PRINT*, " u = ", u
      PRINT*, " sono un autovalore ed il relativo autovettore di A"
   ELSE
      PRINT*, " lamda = ", lamda
      PRINT*, " u = " , u
      PRINT*, " non sono un autovalore ed il relativo autovettore di A"
   END IF
END PROGRAM eigen
```

Può essere utile presentare un esempio di impiego del programma al fine di testarne la correttezza:

```
 Dimensione di A:
3
 Introdurre A per righe:
1.0  2.0  3.0
4.0  5.0  6.0
7.0  8.0  9.0
 Introdurre un autovalore di A:
16.1168
 Introdurre il corrispondente autovettore:
0.23197 .525320 .81867
 A:
   1.000000       2.000000       3.000000
   4.000000       5.000000       6.000000
   7.000000       8.000000       9.000000
  lamda =    16.11680
  u =   0.2319700      0.5253200      0.8186700
  sono un autovalore ed il relativo autovettore di A
```

Il programma seguente mostra, invce, un esempio di definizione e di utilizzo di un array allocabile di un tipo di dati derivato:

```
! Demo strutture
PROGRAM strutture
   IMPLICIT NONE
   INTEGER, PARAMETER :: several=40, length=5
   TYPE :: station  ! Definizione di tipo di dati derivato
```

```
      INTEGER :: station_number
      CHARACTER(LEN=several) :: name
      REAL, DIMENSION(length) :: date
   END TYPE station
   TYPE(station), ALLOCATABLE, DIMENSION(:) :: traverse
   TYPE(station) :: bench_mark
   INTEGER :: loop, n_stations
! start
   READ(*,*) n_stations
   ALLOCATE(traverse(n_stations))
   DO loop = 1, n_stations
      READ(*,*) traverse(loop) ! Legge tutti i componenti di una struttura
   END DO
   bench_mark = traverse(1)    ! Assegnazione di un'intera struttura
   WRITE(*,*) bench_mark       ! Stampa tutti i componenti di una struttura
   STOP
END PROGRAM strutture
```

Un array *correntemente allocato* può essere *deallocato* mediante l'istruzione `DEALLOCATE` che ha lo scopo di *liberare* la memoria precedentemente riservata all'array rendendola disponibile per altre applicazioni. La sintassi dell'istruzione di deallocazione è la seguente:

`DEALLOCATE(`*`lista_di_array`* `[, STAT=`*`variabile_di_stato`*`])`

dove il significato e l'uso della clausola `STAT=`*`variabile_di_stato`* sono identici a quelli validi per l'istruzione di allocazione. E' importante notare che una volta che un array sia stato deallocato i valori in esso immagazzinati sono perduti per sempre.

L'uso combinato delle istruzioni `ALLOCATE` e `DEALLOCATE` consente di lavorare con un array allocabile di dimensioni *continuamente variabili*, come ben descritto dal seguente esempio:

```
REAL, ALLOCATABLE, DIMENSION(:) :: array_variabile
INTEGER :: i, n, alloc_stat, dealloc_stat
...
READ(*,*) n
DO i=1,n
   ALLOCATE(array_variabile(-i:i),STAT=alloc_stat)
   IF(alloc_stat/=0) THEN
      PRINT*, "Spazio insufficiente per allocare l'array quando i = ", i
      STOP
   END IF
   ...     ! Istruzioni che coinvolgono array_variabile
   DEALLOCATE(array_variabile,STAT=dealloc_stat)
   IF(dealloc_stat/=0) THEN
      PRINT*, "Errore in fase di deallocazione"
      STOP
```

```
   END IF
END DO
```

Il precedente frammento di programma alloca il vettore `array_variabile` con estremi `-1` e `+1`, quindi esegue una serie di istruzioni che coinvolgono questo array al termine delle quali l'array viene deallocato. A questo punto il vettore `array_variabile` viene allocato nuovamente ma con estremi `-2` e `+2` e così via. Il suddetto procedimento viene ripetuto `n` volte. E' evidente che, in questo esempio, se il vettore `array_allocabile` non venisse deallocato al termine di ogni iterazione, alla successiva istruzione di allocazione avrebbe luogo un errore in fase di esecuzione.

E' possibile verificare se un array allocabile sia o meno *correntemente allocato* attraverso la funzione intrinseca `ALLOCATED`. Questa funzione, che sarà trattata in maggiore dettaglio nella prossima sezione, ha come argomento di input il nome di un array allocabile e restituisce in output il valore `.TRUE.` se l'array è correntemente allocato, `.FALSE.` in caso contrario. L'uso di questa procedura di interrogazione previene la possibilità di commettere l'errore di tentare lavorare con array non allocati o di allocare array già allocati. Le seguenti istruzioni forniscono due possibili esempi di utilizzo della funzione `ALLOCATED`:

```
REAL, ALLOCATABLE, DIMENSION(:,:) :: a
INTEGER :: stato
...
IF (ALLOCATED(a)) DEALLOCATE(a)
...
IF (.NOT.ALLOCATED(a)) ALLOCATE(a(5,20),STAT=stato)
```

E' bene osservare che l'esecuzione dell'istruzione `DEALLOCATE` *non* è l'unico modo in cui un array allocabile può perdere il suo stato *allocato*. Infatti, l'uscita da una procedura nella quale l'array è correntemente allocato rende l'array stesso *indefinito*: tale array *non* potrà più essere usato, ridefinito, allocato o deallocato. Permettere ad un array di avere uno stato indefinito comporta una riduzione della memoria disponibile per il programma (*memory leak*). Si noti che le *memory leak* sono cumulative per cui un uso ripetuto di una procedura che provochi un simile problema aumenterà sempre più le dimensioni della regione di memoria inutilizzabile. D'altra parte soltanto la deallocazione degli array allocabili è responsabilità del programmatore in quanto tutte le variabili statiche (compresi gli array automatici) vengono rimossi automaticamente all'uscita dalla procedura a cui esse sono locali (a meno, ovviamente, di non averne specificato l'attributo `SAVE` in fase di dichiarazione). Questo "preoccupazione" è stata eliminata dal Fortran 95 il cui standard prevede una *deallocazione automatica*, all'uscita da una procedura, di tutti gli array allocabili locali *non* salvati. E', in ogni caso, sempre cattiva abitudine (oltre che una errore concettuale in Fortran 90) permettere allo stato di array allocabile di diventare indefinito. Quando un *allocatable array* non serve più è sempre consigliabile deallocarlo prima di uscire dalla procedura nella quale è stato allocato. Nel caso in cui, invece, fosse necessario salvarlo fra due chiamate successive, esso andrà dichiarato con l'attributo `SAVE`, come nel seguente esempio:

```
CHARACTER(LEN=50), ALLOCATABLE, DIMENSION(:), SAVE ::  nome
```

In tal caso, l'array rimarrà allocato all'uscita dalla procedura ed i suoi elementi conserveranno il loro valore. Se, invece, all'uscita dalla procedura l'array si trovasse già in uno stato deallocato, esso manterrebbe questo stato.

Si noti che la possibilità di salvare un array allocabile fra due chiamate di procedura è uno dei maggiori vantaggi che presenta l'array allocabile rispetto a quello automatico il quale cessa sempre di esistere all'uscita dalla procedura nel quale è dichiarato. Inoltre, il maggiore controllo offerto dagli array allocabili rispetto a quelli automatici consente di gestire la memoria in maniera molto più dinamica ed efficiente, come mostrato dalla seguente procedura:

```
SUBROUTINE space(n)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: n
   REAL, ALLOCATABLE, DIMENSION(:,:) :: a, b
   ...
      ALLOCATE(a(2*n,6*n))   ! allocato spazio per a
      ...                    ! istruzioni che coinvolgono a
      DEALLOCATE(a)          ! liberato lo spazio occupato da a
      ALLOCATE(b(3*n,4*n))   ! allocato spazio per b
      ...                    ! istruzioni che coinvolgono b
      DEALLOCATE(b)          ! liberato lo spazio occupato da b
END SUBROUTINE space
```

La subroutine `space` usa $12n^2$ elementi per `a` durante l'esecuzione della prima parte della procedura, successivamente libera lo spazio associato a questi elementi, usa ancora $12n^2$ elementi per `b` durante la seconda parte, infine libera anche questa memoria. Pertanto, durante l'esecuzione della procedura, il *massimo* spazio di memoria usato dagli array `a` e `b` è quello richiesto per immagazzinare $12n^2$ valori reali. Se, invece, la subroutine `space` venisse riscritta in modo da usare *array automatici* al posto di quelli allocabili:

```
SUBROUTINE space(n)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: n
   REAL, DIMENSION(2*n,6*n) :: a
   REAL, DIMENSION(3*n,4*n) :: b
      ...      ! istruzioni che coinvolgono a
      ...      ! istruzioni che coinvolgono b
END SUBROUTINE space
```

per tutta la durata dell'esecuzione della procedura sarebbero allocati $12n^2$ elementi per `a` e $12n^2$ elementi per `b`, per un totale (costante) di $24n^2$ elementi creati quando il controllo del programma entra nella procedura e rilasciati soltanto all'uscita dalla procedura stessa. In casi come quello in esame, l'importanza di lavorare con array allocabili è, chiaramente, tanto più avvertita quanto maggiore è il valore di `n`. In ogni caso, la decisione di usare array allocabili o array automatici dipende essenzialmente dalle particolari condizioni. Infatti, nel caso in cui siano richiesti dei grossi array o se essi devono essere conservati fra due chiamate di procedura

consecutive allora sarà necessario lavorare con array allocabili. Se, invece, sono necessari array di modeste dimensioni e non è richiesto che i loro valori vengano conservati, è da preferirsi la maggiore semplicità associata all'uso degli array automatici.

Si noti che esistono tre limitazioni connesse all'uso degli array allocabili:

- Gli array allocabili *non* possono essere *argomenti formali* di una procedura.
- Il risultato di una funzione *non* può essere un array allocabile.
- Gli array allocabili *non* possono essere usati nella definizione di un *tipo di dati derivato*.

A causa della prima di tali restrizioni, un array allocabile deve essere allocato e deallocato nella stessa unità di programma nella quale è stato dichiarato. Questo limite può, tuttavia, essere aggirato dichiarando l'array allocabile in un *modulo*: ciascuna unità di programma che usi quel modulo avrà la possibilità di usare, allocare e deallocare l'array. Se, invece, fosse necessario avere un array dinamico come componente di un tipo di dati derivato, allora si dovrà necessariamente fare uso dei *puntatori*.

E' da precisare, inoltre, che un array allocabile che non risulti correntemente allocato non può essere utilizzato come parametro attuale in una chiamata di procedura, potendo al più essere usato come argomento di alcune funzioni intrinseche, come `ALLOCATED` (che, come detto, fornisce lo stato di allocazione di un array allocabile), `DIGITS` (che dà informazioni circa il *tipo*), `KIND` e `LEN` (che forniscono informazioni circa i *parametri di kind*).

Il seguente esempio mostra, a scopo di riepilogo, un uso combinato di array *allocabili*, *automatici* e *di forma presunta*:

```
PROGRAM array
  IMPLICIT NONE
  REAL, ALLOCATABLE, DIMENSION(:,:) :: a
  REAL :: res
  INTEGER :: n, impos
  INTERFACE
     SUBROUTINE sub(a,res)
        REAL, DIMENSION(:,:), INTENT(IN) :: a
        REAL, DIMENSION(SIZE(a,1),SIZE(a,2)) :: work
     END SUBROUTINE sub
  END INTERFACE
  ...
  READ (*,*) n
  ALLOCATE (a(n,n),STAT=impos)    ! allocatable array
  IF (impos=0) THEN
     CALL sub(a,res)
     ...
  END IF
  ...
```

```
  CONTAINS
     SUBROUTINE sub(a,res)
        IMPLICIT NONE
! Parametri formali
        REAL, INTENT(OUT) :: res
        REAL, DIMENSION(:,:), INTENT(IN) :: a          ! assumed shape array
! Variabili locali
        REAL, DIMENSION (SIZE(a,1),SIZE(a,2)) :: work  ! automatic array
           ...
           res = a(...)
           ...
     END SUBROUTINE sub
END PROGRAM array
```

Un ulteriore esempio di uso combinato delle diverse tipologie di array è fornito dal seguente programma che valuta il determinante di una matrice quadrata A. Detta matrice viene definita come allocabile nel programma principale, viene quindi allocata in base alla dimensione n il cui valore è di input. Una volta che la lettura della matrice abbia avuto termine, il main invoca la funzione ricorsiva `det` che valuta il determinante secondo la *regola di Laplace* applicata alla prima riga ossia come la somma dei prodotti degli elementi della suddetta riga per i rispettivi complementi algebrici:

$$\det A = \sum_{j=1}^{n} (-1)^{(1+j)} a_{1j} \det A_{1j}$$

dove, si ricorda, A_{1j} è la sotto-matrice ottenuta da A cancellando la prima riga e la j_ma colonna. Il determinante $\det A_{1j}$ è detto *minore complementare dell'elemento* a_{1j}, il prodotto $(-1)^{(1+j)}\det A_{1j}$ è detto *complemento algebrico di* a_{1j}.

Si noti come la matrice venga passata alla funzione come array correntemente allocato e venga, invece, letta come un comune array di forma presunta. La stessa function, poi, crea ad ogni invocazione ricorsiva un array automatico (di dimensioni sempre diverse) che rappresenta una estratta di `A` il cui determinante è il *minore* relativo al generico elemento di `A`.

```
PROGRAM determinante
! Scopo: calcolare il determinante di una matrice quadrata.
! Gli elementi della matrice vengono letti dal file matrix.dat
  IMPLICIT NONE
  INTEGER :: n  ! dimensione della matrice
  REAL,DIMENSION(:,:),ALLOCATABLE :: A
  INTEGER :: stato
!
  INTERFACE
     RECURSIVE FUNCTION det(n,A) RESULT(detA)
        IMPLICIT NONE
        REAL :: detA
```

```
        INTEGER,INTENT(IN) :: n
        REAL,INTENT(IN),DIMENSION(n,n) :: A
     END FUNCTION det
  END INTERFACE
!
  OPEN(UNIT=11,FILE='matrix.dat',STATUS='OLD',ACTION='READ')
  READ(11,*) n
  ALLOCATE(A(n,n),IOSTAT=stato)
  IF (stato==0) THEN
     READ(11,*) A
     CLOSE(11)
! Calcolo del determinante e stampa del risultato
     WRITE(*,'(1X,A22,1X,G10.5)') "Il determinante vale: ",det(n,A)
  ELSE
     WRITE(*,*) "L'allocazione della matrice e' fallita"
  END IF
END PROGRAM determinante
!
!-------------------------------------------------------
!
RECURSIVE FUNCTION det(n,A) RESULT(detA)
! N.B.: Questa subroutine valuta il determinante della
! matrice quadrata A(n,n) in maniera ricorsiva secondo
! la regola di Laplace applicata alla prima riga
!
! Sezione dichiarativa
   REAL :: detA ! determinante di A (valore restituito dalla funzione)
! Parametri formali
   IMPLICIT NONE
   INTEGER,INTENT(IN) :: n
   REAL,INTENT(IN),DIMENSION(n,n) :: A  ! N.B.: array fittizio
! Variabili locali
   INTEGER :: j
   REAL,DIMENSION(n-1,n-1) :: m  ! N.B.:array automatico
   REAL :: detm ! determinante del generico minore
! A e' la matrice di cui valutare il determinante
! m e' il minore associato al generico elemento A(1,j)
! Sezione esecutiva
      IF (n==1) THEN
         detA = A(1,1)
      ELSE
         detA = 0.0
         DO j=1,n
```

```
! Costruzione del generico minori
            m(1:n-1,1:j-1) =a (2:n,1:j-1)
            m(1:n-1,j:n-1) =a (2:n,j+1:n)
! Calcolo del complemento algebrico (o del determinante)
            detm = det(n-1,m)  ! invocazione ricorsiva della funzione
            detA = detA+A(1,j)*(-1.0)**(1+j)*detm
         END DO
      END IF
      RETURN
END FUNCTION det
```

6.6 Procedure intrinseche per gli array

Oltre alle procedure cosiddette *di elemento*, il Fortran 90/95 prevede molte procedure intrinseche specificamente progettate per operare con gli array. L'uso di tali procedure intrinseche è fortemente raccomandato, quando si scrivano programmi che prevedano una fase di *array processing*, per molte ragioni:

- Il programma risultante dall'uso di procedure intrinseche appare, generalmente, più semplice e compatto.
- Le procedure intrinseche vengo sovente fornite, dal fornitore del compilatore, in linguaggio *assembly* per cui risultano sempre molto efficienti.
- E' richiesto al programmatore uno sforzo molto minore per la gestione degli array.

A seconda degli scopi per cui sono state progettate, le procedure intrinseche operanti su array si dividono in diverse categorie. Di ciascuna di esse si parlerà nelle prossime pagine.

6.6.1 Procedure di riduzione

`ALL(MASK[,DIM])`

Restituisce il valore `.TRUE.` se *tutti* gli elementi dell'array logico `MASK` hanno valore `.TRUE.`, in caso contrario fornisce il valore `.FALSE.` Il risultato è `.FALSE.` anche nel caso in cui `MASK` abbia dimensione nulla. Se viene specificato l'argomento opzionale `DIM`, la suddetta verifica viene effettuata esclusivamente nella dimensione specificata.

Se l'argomento `DIM` viene specificato, esso dovrà essere uno scalare di tipo `INTEGER` con valore compreso tra `1` ed n, essendo n il rango di `MASK`.

Il risultato di `ALL` è uno scalare (se l'argomento `DIM` è omesso) oppure un array (se `DIM` è presente) di tipo `LOGICAL`. In particolare, un risultato array ha sempre lo *stesso tipo* e gli *stessi parametri di kind* di `MASK`, e *rango* pari a quello di `MASK` diminuito di un'unità.

Esempi:

`ALL((/.TRUE.,.FALSE.,.TRUE./))` restituisce il valore `.FALSE.` poiché il secondo elemento di `MASK` è `.FALSE.`

`ALL((/.TRUE.,.TRUE.,.TRUE./))` restituisce il valore `.TRUE.` poiché *tutti* gli elementi di `MASK` sono `.TRUE.`

Siano `a` la matrice:

```
1  5  7
3  6  8
```

e `b` la matrice:

```
0  5  7
2  6  9
```

`ALL(a==b,DIM=1)` verifica che *tutti gli elementi* in ciascuna colonna di `a` risultino uguali agli elementi nelle corrispondenti colonne di `b`. Pertanto in questo caso il risultato sarà:

```
(/.FALSE.,.TRUE.,.FALSE./)
```

poiché soltanto gli elementi della seconda colonna sono ordinatamente uguali nelle due matrici.

`ALL(a==b,DIM=2)` verifica se *tutti gli elementi* in ciascuna riga di `a` risultino uguali agli elementi nelle corrispondenti righe di `b`. Il risultato della funzione sarà, pertanto:

```
(/.FALSE.,.FALSE./).
```

`ANY(MASK[,DIM])`

Restituisce il valore `.TRUE.` se *almeno un elemento* dell'array logico `MASK` ha valore `.TRUE.`, in caso contrario fornisce il valore `.FALSE.` Il risultato è `.FALSE.` anche nel caso in cui `MASK` abbia dimensione nulla. Se viene specificato l'argomento opzionale `DIM`, la suddetta verifica viene effettuata esclusivamente nella dimensione specificata.

Se l'argomento `DIM` viene specificato, esso dovrà essere uno scalare di tipo `INTEGER` con valore compreso tra `1` ed n, essendo n il rango di `MASK`.

Il risultato di `ANY` è uno scalare (se l'argomento `DIM` è omesso o se `MASK` ha rango unitario) oppure un array (se `DIM` è presente) di tipo `LOGICAL`. In particolare, un risultato array ha sempre lo *stesso tipo* e gli *stessi parametri di kind* di `MASK`, e *rango* pari a quello di `MASK` diminuito di un'unità.

Esempi:

`ANY((/.FALSE.,.TRUE.,.FALSE./))` restituisce il valore `.TRUE.` poiché un elemento di `MASK` (per l'esattezza, il secondo) è `.TRUE.`

Siano `a` la matrice:

```
1  5  7
3  6  8
```

e `b` la matrice:

```
0  5  7
2  6  9
```

`ANY(a==b,DIM=1)` verifica se *qualche elemento* di una qualsiasi colonna di `a` sia uguale al corrispondente elemento di `b`. Il risultato della funzione sarà, pertanto:

```
(/.FALSE.,.TRUE.,.TRUE./)
```

poiché i due array hanno elementi comuni soltanto nella seconda e nella terza colonna.

`ANY(a==b,DIM=2)` verifica se *qualche elemento* di una qualsiasi riga di `a` sia uguale al elemento di `b`. Il risultato della funzione sarà, pertanto:

```
(/.TRUE.,.TRUE./).
```

in quanto le prime righe hanno in comune gli elementi `5` e `7`, le seconde l'elemento `6`.

`COUNT(MASK[,DIM])`

Questa funzione conta il numero di ricorrenze del valore `.TRUE.` nell'array logico `MASK`. Se viene specificato l'argomento opzionale `DIM`, la suddetta operazione viene effettuata esclusivamente nella dimensione specificata. Quando specificato, il parametro `DIM` deve essere uno scalare di tipo `INTEGER` con valore compreso tra `1` ed n, essendo n il rango di `MASK`.

Il risultato di `COUNT` è uno scalare (se l'argomento `DIM` è omesso o se `MASK` ha rango unitario) oppure un array (se `DIM` è presente) di tipo `LOGICAL`. In particolare, un risultato array ha sempre lo *stesso tipo* e gli *stessi parametri di kind* di `MASK`, e *rango* pari a quello di `MASK` diminuito di un'unità.

Esempi:

`COUNT((/.TRUE.,.FALSE.,.TRUE./))` restituisce il valore `2` perché tanti sono gli elementi `.TRUE.` dell'array argomento.

`COUNT((/.TRUE.,.TRUE.,.TRUE./))` restituisce il valore `3` perché tutti e tre gli

elementi dell'array argomento hanno valore `.TRUE.`

Siano `a` la matrice:

```
1  5  7
3  6  8
```

e `b` la matrice:

```
0  5  7
2  6  9
```

`COUNT(a/=b,DIM=1)` conta quanti elementi in *ciascuna colonna* di `a` abbiano valore diverso dai corrispondenti elementi nelle colonne di `b`. Il risultato sarà, pertanto, `(/2,0,1/)`, poiché la prima colonna di `a` e la prima colonna di `b` hanno entrambi gli elementi di valore differente, la seconda colonna di `a` e la seconda colonna di `b` hanno tutti e due gli elementi in comune (e, quindi, nessun valore differente), la terza colonna di `a` e la terza colonna di `b` differiscono per un solo elemento.

`COUNT(a==b,DIM=2)` conta quanti elementi in ciascuna *riga* di `a` abbiano lo stesso valore dei corrispondenti elementi nelle righe di `b`. In tal caso il risultato è `(/2, 1/)`, poiché la prima riga di `a` ha due elementi in comune con la prima riga di `b`, mentre la seconda riga di `a` ha un solo valore in comune con la corrispondente riga di `b`.

`MAXVAL(ARRAY[,DIM][,MASK])`

Restituisce il valore massimo di un array, di una sezione di array o di in una specificata dimensione di un array.

L'argomento `ARRAY` è un array di tipo intero o reale.

L'argomento opzionale `DIM` è uno scalare di tipo intero con valore compreso nell'intervallo $1 \div n$, essendo n il rango di `ARRAY`. Si noti che questo parametro *non* è previsto dal Fortran 90 ma soltanto dal Fortran 95.

L'argomento opzionale `MASK` è un array logico compatibile con `ARRAY`.

Il risultato della funzione è un array o uno scalare avente *stesso tipo* di `ARRAY`. Il risultato è uno scalare se `DIM` è omesso oppure se `ARRAY` ha rango unitario.

Nel caso in cui il parametro `DIM` sia omesso valgono le seguenti regole:

- Se `MASK` è assente, il risultato rappresenta il valore massimo fra quelli di tutti gli elementi di `ARRAY`.
- Se `MASK` è presente, il risultato rappresenta il valore massimo fra tutti gli elementi di `ARRAY` che siano compatibili con la condizione specificata da `MASK`.

Nel caso in cui il parametro `DIM` sia presente valgono le seguenti regole:

- L'array risultante ha rango pari quello di `ARRAY` diminuito di un'unità.
- Se `ARRAY` ha rango unitario, allora il valore di `MAXVAL(ARRAY,DIM[,MASK])` sarà pari a quello di `MAXVAL(ARRAY[,MASK])`.
- Se `ARRAY` ha ampiezza nulla, oppure tutti gli elementi di `MASK` hanno valore `.FALSE.`, il valore del risultato (se `DIM` è assente) oppure il valore degli elementi dell'array risultante (se `DIM` è presente) è pari al più piccolo numero negativo supportato dal processore per il *tipo* ed i *parametri di kind* di `ARRAY`.

Esempi:

Il valore di `MAXVAL((/2,4,-1,7/))` è `7`, essendo questo il valore massimo dell'array.

Sia `mat` l'array:

```
2  6  4
5  3  7
```

`MAXVAL(mat,MASK=mat<5)` fornisce il valore `4` poiché questo è il valore massimo di `mat` fra tutti quelli minori di `5`.

`MAXVAL(mat,DIM=1)` fornisce il valore `(/5,6,7/)`, essendo `5` il massimo valore di `mat` nella prima colonna, `6` il massimo valore di `mat` nella seconda colonna, e `7` il massimo valore di `mat` nella terza colonna.

`MAXVAL(mat,DIM=2)` fornisce il valore `(/6,7/)`, essendo `4` il massimo valore di `mat` nella prima riga e `7` il massimo valore di `mat` nella seconda riga.

`MINVAL(ARRAY[,DIM][,MASK])`

Restituisce il valore minimo di un array, di una sezione di array o di in una specificata dimensione di un array.

L'argomento `ARRAY` è un array di tipo intero o reale.

L'argomento opzionale `DIM` è uno scalare di tipo intero con valore compreso nell'intervallo $1 \div n$, essendo n il rango di `ARRAY`. Si noti che questo parametro *non* è previsto dal Fortran 90 ma soltanto dal Fortran 95.

L'argomento opzionale `MASK` è un array logico compatibile con `ARRAY`.

Il risultato della funzione è un array o uno scalare avente *stesso tipo* di `ARRAY`. Il risultato è uno scalare se `DIM` è omesso oppure se `ARRAY` ha rango unitario.

Nel caso in cui il parametro `DIM` sia omesso valgono le seguenti regole:

- Se `MASK` è assente, il risultato rappresenta il valore minimo fra quelli di tutti gli elementi di `ARRAY`.
- Se `MASK` è presente, il risultato rappresenta il valore minimo fra tutti gli elementi di `ARRAY` che siano compatibili con la condizione specificata da `MASK`.

Nel caso in cui il parametro DIM sia presente valgono le seguenti regole:

- L'array risultante ha rango pari quello di ARRAY diminuito di un'unità.
- Se ARRAY ha rango unitario, allora il valore di MINVAL(ARRAY,DIM[,MASK]) sarà pari a quello di MINVAL(ARRAY[,MASK]).
- Se ARRAY ha ampiezza nulla, oppure se tutti gli elementi di MASK hanno valore .FALSE., il valore del risultato (se DIM è assente) oppure il valore degli elementi dell'array risultante (se DIM è presente) è pari al più grande numero positivo supportato dal processore per il *tipo* ed i *parametri di kind* di ARRAY.

Esempi:

Il valore di MINVAL((/2,4,-1,7/)) è -1, essendo questo il valore minimo dell'array.

Sia mat l'array:

```
2  6  4
5  3  7
```

MINVAL(mat,MASK=mat>3) fornisce il valore 4 poiché questo è il valore minimo di mat fra quelli maggiori di 3.

MINVAL(mat,DIM=1) fornisce il valore [2,3,4], essendo 2 il minimo valore di mat nella prima colonna, 3 il minimo valore di mat nella seconda colonna, e 4 il minimo valore di mat nella terza colonna.

MINVAL(mat,DIM=2) fornisce il valore [2,3], essendo 2 il minimo valore di mat nella prima riga e 3 il minimo valore di mat nella seconda riga.

PRODUCT(ARRAY[,DIM][,MASK])

Restituisce il valore del prodotto di tutti gli elementi di un intero array oppure degli elementi di un array in una data dimensione.

L'argomento ARRAY deve essere un array di tipo INTEGER o REAL.

L'argomento opzionale DIM deve essere uno scalare di tipo INTEGER con valore compreso tra 1 ed n, essendo n il rango di ARRAY.

L'altro argomento opzionale MASK deve essere un array di tipo LOGICAL compatibile con ARRAY.

Il risultato della funzione è uno scalare (se l'argomento DIM è assente oppure se ARRAY ha rango unitario) oppure un array (se DIM è presente) dello stesso tipo di ARRAY.

Nel caso in cui DIM sia assente, valgono le seguenti regole:

- Se l'argomento MASK è assente, il risultato rappresenta il prodotto di tutti gli elementi di ARRAY. Se ARRAY ha dimensione nulla, il risultato vale 1.

- Se l'argomento MASK è presente, il risultato rappresenta il prodotto di tutti gli elementi di ARRAY che corrispondano ad elementi .TRUE. di MASK. Se ARRAY ha dimensione nulla, oppure tutti gli elementi di MASK hanno valore .FALSE., il risultato vale 1.

Nel caso, invece, in cui DIM venga specificato, si ha:

- Se ARRAY ha rango unitario, il valore della funzione è pari a PRODUCT(ARRAY[,MASK]).
- L'array risultante ha rango pari a quello di ARRAY diminuito di un'unità.

Esempi:

PRODUCT((/2,3,4/)) restituisce valore 24 (tale essendo il risultato del prodotto 2×3×4).

PRODUCT((/2,3,4/),DIM=1) restituisce, chiaramente, lo stesso valore.

Sia a l'array:

```
1  4  7  2
2  3  5  1
```

PRODUCT(a,DIM=1) restituisce il valore (/2,12,35,2/), che è il prodotto degli elementi di ciascuna colonna (ossia 1×2, nella prima colonna, 4×3 nella seconda colonna, e così via).

PRODUCT(a,DIM=2) restituisce il valore (/56,30/), che è il prodotto degli elementi di ciascuna riga (ossia 1×2×7×2, nella prima riga, 2×3×5×1, nella seconda riga).

SUM(ARRAY[,DIM][,MASK])

Restituisce il valore della somma di tutti gli elementi di un intero array oppure degli elementi din un array in una data dimensione.

L'argomento ARRAY deve essere un array di tipo INTEGER, REAL o COMPLEX.

L'argomento opzionale DIM deve essere uno scalare di tipo INTEGER con valore compreso tra 1 ed n, essendo n il rango di ARRAY.

L'argomento opzionale MASK deve essere un array di tipo LOGICAL compatibile con ARRAY.

Il risultato della funzione è uno scalare (se l'argomento DIM è assente oppure se ARRAY ha rango unitario) oppure un array (se DIM è presente) dello stesso tipo di ARRAY.

Nel caso in cui DIM *non* sia specificato, valgono le seguenti regole:

- Se l'argomento MASK è assente, il risultato rappresenta la somma di tutti gli elementi di ARRAY. Se ARRAY ha dimensione nulla, il risultato vale zero.
- Se l'argomento MASK è presente, il risultato rappresenta la somma di tutti gli elementi di ARRAY che corrispondano ad elementi .TRUE. di MASK. Se ARRAY ha dimensione nulla, oppure se tutti gli elementi di MASK hanno valore .FALSE., il risultato vale zero.

Nel caso, invece, in cui DIM venga specificato, si ha:

- Se ARRAY ha rango uno, il valore della funzione è pari a SUM(ARRAY[,MASK]).
- L'array risultante ha rango pari a quello di ARRAY diminuito di un'unità.

Esempi:

SUM((/2,3,4/)) restituisce valore 9 (tale essendo il risultato della somma: 2+3+4).

SUM((/2,3,4/),DIM=1) restituisce, chiaramente, lo stesso valore.

Sia a l'array:

```
1  4  7  2
2  3  5  1
```

SUM(a,DIM=1) restituisce il valore (/3,7,12,3/), che è la somma degli elementi di ciascuna colonna (ossia 1+2, nella prima colonna, 4+3 nella seconda colonna, e così via).

SUM(a,DIM=2) fornisce come risultato l'array (/14,11/), che è la coppia formata dalla somma degli elementi di ciascuna riga (ossia 1+2+7+2, nella prima riga, 2+3+5+1, nella seconda riga).

6.6.2 Procedure di interrogazione

ALLOCATED(ARRAY)

Indica se l'*allocatable array* ARRAY risulti o meno allocato. Il risultato è uno scalare di tipo LOGICAL avente valore .TRUE. se lo stato di allocazione di ARRAY è *correntemente allocato*, .FALSE. se esso è *non allocato* oppure è *indefinito*.

Un esempio di utilizzo della funzione intrinseca ALLOCATE è fornito dal seguente frammento di programma:

```
REAL, ALLOCATABLE, DIMENSION (:,:,:) :: arr
...
PRINT*, ALLOCATED(arr)    ! stampa il valore .FALSE.
ALLOCATE(arr(12,15,20))
PRINT*, ALLOCATED(arr)    ! stampa il valore .TRUE.
```

LBOUND(ARRAY[,DIM])

Restituisce un array monodimensionale formato dai *limiti inferiori* di tutte le dimensioni di ARRAY, oppure, se DIM è presente, uno scalare avente come valore il *limite inferiore* relativo alla dimensione specificata. In entrambi i casi il risultato è di tipo INTEGER.

L'argomento `ARRAY` è un array di tipo e rango qualsiasi con l'unica restrizione che esso non può essere né un *allocatable array non correntemente allocato* né un *puntatore deassociato*.

L'argomento opzionale `DIM` deve essere uno scalare di tipo `INTEGER` con valore compreso tra `1` ed n, essendo n il rango di `ARRAY`.

Se `DIM` è presente, il risultato è uno scalare. In caso contrario, esso sarà un array di rango unitario con ogni elemento riferito alla corrispondente dimensione di `ARRAY`.

Se `ARRAY` è una sezione di array o una espressione array (diversa, però, dall'intero array), ciascun elemento del risultato avrà valore pari ad uno. Allo stesso modo, se `ARRAY(DIM)` ha dimensione nulla il corrispondente elemento del risultato avrà valore unitario.

Le seguenti istruzioni serviranno a chiarire l'utilizzo della funzione `LBOUND`:

```
REAL, DIMENSION(1:3,5:8) :: a
INTEGER, DIMENSION(2:8,-3:20) :: b
...
LBOUND(a)        ! ris.: (/1,5/)
LBOUND(a,DIM=2)  ! ris.: (/5/)
LBOUND(b)        ! ris.: (/2,-3/)
LBOUND(b(5:8,:)) ! ris.: (/1,1/) N.B.: l'argomento è una sezione di array
```

`SHAPE(SOURCE)`

Restituisce la *forma* di un array o di un argomento scalare.

L'argomento `SOURCE` può essere uno scalare o un array di tipo qualsiasi con l'unica restrizione che *non* può essere un *assumed-size array*, un *allocatable array non correntemente allocato* o un *puntatore deassociato*.

Il risultato è un array di tipo `INTEGER` di rango unitario e di ampiezza pari al rango di `SOURCE`. Il valore del risultato coincide con la forma di `SOURCE`.

Ad esempio, `SHAPE(2)` fornisce come risultato un array monodimensionale di ampiezza nulla. Dato, invece, l'array `b` così definito:

```
REAL, DIMENSION(2:4,-3:1) ::  b
```

l'istruzione:

```
SHAPE(SOURCE=b)
```

fornirà il valore `(/3,5/)`.

`SIZE(ARRAY[,DIM])`

Restituisce il *numero totale* degli elementi in un array, oppure l'*estensione* di un array lungo una dimensione specificata.

L'argomento `ARRAY` può essere un array di tipo qualsiasi, con l'unica restrizione che non può essere un *puntatore deassociato* o un *allocatable array non correntemente associato*. Può essere,

invece, un *assumed-size array* se il parametro DIM è presente con un valore minore del rango di ARRAY.

L'argomento opzionale DIM deve essere uno scalare di tipo INTEGER con valore compreso tra 1 ed n, essendo n il rango di ARRAY.

Il risultato è sempre uno scalare di tipo INTEGER. In particolare, esso è pari al numero totale di elementi di ARRAY nel caso in cui il parametro DIM non sia specificato; se, invece, DIM è presente, il risultato rappresenta l'*estensione* di ARRAY nella dimensione DIM.

A titolo di esempio, se l'array b è dichiarato come:

```
REAL, DIMENSION(2:4,-3:1) :: b
```

le istruzioni:

```
SIZE(b,DIM=2)
SIZE(b)
```

forniscono i valori 5 e 15, rispettivamente.

`UBOUND(ARRAY[,DIM])`

Restituisce un array monodimensionale formato dai *limiti superiori* di tutte le dimensioni di ARRAY, oppure, se DIM è presente, uno scalare avente come valore il *limite superiore* relativo alla dimensione specificata. Il risultato è, in ogni caso, di tipo INTEGER.

L'argomento ARRAY è un array di tipo e rango qualsiasi con l'unica restrizione che non può essere né un *allocatable array non correntemente allocato* né un *puntatore deassociato*. Può, invece, essere un *assumed-size array* se il parametro DIM è presente con un valore *minore* del rango di ARRAY.

L'argomento opzionale DIM deve essere uno scalare di tipo INTEGER con valore compreso tra 1 ed n, essendo n il rango di ARRAY.

Se DIM è presente, il risultato è uno scalare. In caso contrario, esso sarà un array di rango unitario con ogni elemento riferito alla corrispondente dimensione di ARRAY.

Se ARRAY è una sezione di array o una espressione array che non coincida con l'intero array, allora UBOUND(ARRAY,DIM) avrà un valore pari al numero di elementi nella dimensione specificata. Allo stesso modo, se ARRAY(DIM) ha dimensione nulla, il corrispondente elemento del risultato avrà valore nullo.

Le seguenti istruzioni serviranno a chiarire l'utilizzo della funzione UBOUND:

```
REAL, DIMENSION(1:3,5:8) :: a
INTEGER, DIMENSION(2:8,-3:20) :: b
...
UBOUND(a)         ! ris.: (/3,8/)
UBOUND(a,DIM=2)   ! ris.: (/8/)
UBOUND(b)         ! ris.: (/8,20/)
UBOUND(b(5:8,:)) ! ris.: (/7,24/)
```

dove il risultato dell'ultima istruzione dipende dal fatto che l'argomento di `UBOUND` è una sezione di array per cui il risultato è dato dal numero di elementi in ciascuna dimensione.

Il programma che segue, utile per "fondere" insieme due array monodimensionali e ordinarne gli elementi sia in senso crescente che in senso decrescente, mostra un possibile impiego delle procedure intrinseche `UBOUND` e `LBOUND` (si noti che la funzione `merge` invocata nel programma è user-defined e non rappresenta, invece, l'omonima funzione intrinseca che si avrà modo di incontrare nel prossimo paragrafo):

```
MODULE sort
 CONTAINS
   RECURSIVE SUBROUTINE sub_merge_sort(a,ascend)
! *** Sezione dichiarativa ***
      IMPLICIT NONE
! Parametri formali
      INTEGER, DIMENSION(:), INTENT(INOUT) :: a
      LOGICAL, INTENT(IN), OPTIONAL :: ascend
! Variabili locali
      LOGICAL :: up
      INTEGER :: low, high, mid
! *** Sezione esecutiva ***
! Se il parametro "ascend" non viene specificato, allora il criterio
!  di ordinamento adottato per default e' quello nel verso "crescente"
         IF (PRESENT(ascend)) THEN
            up = ascend
         ELSE
            up = .TRUE.
         ENDIF
         low = LBOUND(a,1)
         high= UBOUND(a,1)
         IF (low<high) THEN
            mid=(low+high)/2
            CALL sub_merge_sort(a(low:mid),up)
            CALL sub_merge_sort(a(mid+1:high),up)
            a(low:high) = merge(a(low:mid),a(mid+1:high),up)
         END IF
   END SUBROUTINE sub_merge_sort
!
   FUNCTION merge(a,b,up)
! *** Sezione dichiarativa ***
      IMPLICIT NONE
! Parametri formali
      INTEGER, DIMENSION(:), INTENT(INOUT) :: a, b
      LOGICAL, INTENT(IN) :: up
! Variabili locali
```

```
      INTEGER, DIMENSION(SIZE(a)+SIZE(b)) :: Merge
      INTEGER :: a_ptr, a_high, a_low
      INTEGER :: b_ptr, b_high, b_low
      INTEGER :: c_ptr
      LOGICAL condition
! *** Sezione esecutiva ***
         a_low  = LBOUND(a,1)
         a_high = UBOUND(a,1)
         b_low  = LBOUND(b,1)
         b_high = UBOUND(b,1)
         a_ptr  = a_low
         b_ptr  = b_low
         c_ptr  = 1
         DO WHILE (a_ptr<=a_high .AND. b_ptr<=b_high)
            IF (up) THEN
               condition= (a(a_ptr) <= b(b_ptr))
            ELSE
               condition= (a(a_ptr) >= b(b_ptr))
            END IF
            IF (condition) THEN
               merge(c_ptr)=a(a_ptr)
               a_ptr=a_ptr+1
            ELSE
               merge(c_ptr)=b(b_ptr)
               b_ptr=b_ptr+1
            END IF
            c_ptr = c_ptr + 1
         END DO
         IF (a_ptr>a_high) THEN
            merge(c_ptr:) = b(b_ptr:b_high)
         ELSE
            merge(c_ptr:) = a(a_ptr:a_high)
         END IF
   END FUNCTION merge
END MODULE sort

PROGRAM merge_sort
! *** Sezione dichiarativa ***
  USE Sort
  IMPLICIT NONE
  INTEGER, DIMENSION(:), ALLOCATABLE :: array
  INTEGER :: i, n
  REAL :: r, time
```

```
! *** Sezione esecutiva ***
  PRINT*, "Dimensione dell'array:"
  READ(*,*) n
  ALLOCATE(array(n))
  DO i=1,n
     CALL RANDOM_NUMBER(r)
     array(i) = INT(r*100.)
  END DO
  PRINT '(20I3)', array(1:20)
  time = second()
  CALL sub_merge_sort(array)
  PRINT*, "Ordinamento crescente"
  PRINT '("(Tempo impiegato = ",F10.3," secondi )")', second()-time
  PRINT '(20I3)', array(1:20)
  time = second()
  CALL sub_merge_sort(array, .FALSE.)
  PRINT*, "Ordinamento decrescente"
  PRINT '("(Tempo impiegato = ",F10.3," secondi )")', second()-time
  PRINT '(20I3)', array(1:20)
CONTAINS
   REAL FUNCTION second()
      IMPLICIT NONE
      INTEGER :: i, timer_count_rate, timer_count_max
         CALL SYSTEM_CLOCK( i, timer_count_rate, timer_count_max )
         second = REAL(i)/timer_count_rate
   END FUNCTION second
END PROGRAM merge_sort
```

6.6.3 Procedure di costruzione

`MERGE(TSOURCE,FSOURCE,MASK)`

Seleziona tra due valori, o tra gli elementi corrispondenti di due array, in accordo con la condizione specificata da una *maschera*.

`TSOURCE` può essere uno scalare o un array di tipo qualsiasi.

`FSOURCE` è uno scalare o un array avente *stesso tipo* e *parametri di kind* di `TSOURCE`.

`MASK` è un array logico.

Il risultato (che ha lo stesso tipo di `TSOURCE`) viene determinato prendendo, *elemento*×*elemento*, il valore corrispondente di `TSOURCE` (se `MASK` è `.TRUE.`) o di `FSOURCE` (se `MASK` è `.FALSE.`).

Ad esempio, se la variabile intera `r` ha valore `-3`, allora il risultato di

```
MERGE(1.0,0.0,r<0)
```

ha valore `1.0`, mentre per `r` pari a `7` ha the valore `0.0`.

Un esempio appena più complesso è il seguente. Se TSOURCE è l'array:

```
1  3  5
2  4  6
```

FSOURCE è l'array:

```
8  9  0
1  2  3
```

e MASK è l'array:

```
.FALSE.   .TRUE.   .TRUE.
 .TRUE.   .TRUE.   .FALSE.
```

allora l'istruzione:

```
MERGE(TSOURCE,FSOURCE,MASK)
```

produce il risultato:

```
8  3  5
2  4  3
```

`PACK(ARRAY,MASK[,VECTOR])`

Legge gli elementi di un array e li "compatta" in un array di rango unitario sotto il controllo di una *maschera*.

L'argomento ARRAY è un array di tipo qualsiasi.

MASK è un array logico compatibile con ARRAY. Esso determina quali elementi vengono acquisiti da ARRAY.

L'argomento opzionale VECTOR è un array monodimensionale avente lo *stesso tipo* e *parametri di kind* di ARRAY e la cui dimensione deve essere almeno t, essendo t il numero di elementi .TRUE. di MASK.

Se MASK è uno *scalare* con valore .TRUE., VECTOR deve avere un numero di elementi almeno pari ad ARRAY. Gli elementi di VECTOR sono inseriti nell'array risultante se non esistono sufficienti elementi selezionati da MASK.

Il risultato è un array di rango unitario avente *stesso tipo* e *stessi parametri di kind* di ARRAY.

Se VECTOR è presente, l'ampiezza dell'array risultante coincide con quella di VECTOR. In caso contrario, l'ampiezza dell'array risultante è pari al numero di elementi .TRUE. di MASK, o al numero degli elementi di ARRAY (se MASK è uno scalare con valore .TRUE.).

Gli elementi di ARRAY vengono processati secondo l'*array element order*, per formare l'array risultante, in modo tale che l'elemento i del risultato coincida con l'elemento di ARRAY corrispondente all'i_mo elemento .TRUE. di MASK. Se VECTOR è presente ed ha un numero di elementi maggiore del numero di elementi .TRUE. di MASK, tutti gli elementi mancanti vengono acquisiti dalle corrispondenti posizioni in VECTOR.

Ad esempio, se mat è il seguente array:

```
0  8  0
0  0  0
7  0  0
```

allora l'istruzione:

```
PACK(mat,MASK=mat/=0,VECTOR=(/1,3,5,9,11,13/))
```

produrrà il risultato: `(/7,8,5,9,11,13/)`, mentre l'istruzione:

```
PACK(mat,MASK=mat/=0)
```

fornirà come risultato il vettore `(/7,8/)`.

Un più interessante impiego della funzione in oggetto è offerto dalla seguente subroutine il cui compito è quello di estrarre gli eleemnti della diagonale principale da una matrice quadrata di input:

```
SUBROUTINE Matrix_Diagonal(A,diag,n)
! Scopo: estrarre la diagonale principale
! da una matrice quadrata A di n elementi
   IMPLICIT NONE
   REAL, INTENT(IN), DIMENSION(:,:) :: A
   REAL, INTENT(OUT), DIMENSION(:) :: diag
   INTEGER, INTENT(IN) :: n
   REAL, DIMENSION (1:SIZE(A,1)*SIZE(A,1)) :: temp
!
      IF(SIZE(A,1)==N .AND. SIZE(A,2)==N) THEN
         temp = PACK(A,.TRUE.)
         diag = temp(1:n*n:n+1)
      ELSE
         PRINT*, "La matrice A non e' quadrata"
         PRINT*, "Impossibile estrarre la diagonale"
      END IF
      RETURN
END SUBROUTINE Matrix_Diagonal
```

`SPREAD(SOURCE,DIM,NCOPIES)`

Crea un array "replicato" con una dimensione aggiuntiva eseguendo copie degli elementi esistenti lungo una dimensione specificata.

`SOURCE` è uno scalare o un array di tipo qualsiasi (nel caso di array, il rango deve essere minore di 7).

`DIM` è uno scalare di tipo `INTEGER` con valore compreso tra `1` ed $n+1$, essendo n il rango di `SOURCE`.

NCOPIES è uno scalare di tipo INTEGER. Esso rappresenta l'estensione della dimensione addizionale.

Il risultato è un array dello stesso tipo di SOURCE e di rango pari a quello di SOURCE aumentato di un'unità. In particolare, valgono le seguenti regole:

- Se SOURCE è un array, ciascun elemento dell'array risultante nella dimensione DIM è pari al corrispondente elemento di SOURCE.
- Se SOURCE è uno scalare, il risultato è un array di rango unitario formato da NCOPIES elementi, ciascuno con il valore di SOURCE.
- Se NCOPIES è minore o uguale a zero, il risultato è un array di ampiezza nulla.

Un esempio di utilizzo della funzione SPREAD è fornito dalla seguente istruzione:

```
SPREAD ("c",1,4)
```

la quale produce l'array di caratteri: (/c,c,c,c/). Ancora, se vet è l'array (/3,4,5/) ed nc ha valore 4, allora l'istruzione:

```
SPREAD(vet,DIM=1,NCOPIES=nc)
```

produrrà l'array:

```
3  4  5
3  4  5
3  4  5
3  4  5
```

mentre l'istruzione:

```
SPREAD(vet,DIM=2,NCOPIES=nc)
```

produce l'array:

```
3  3  3  3
4  4  4  4
5  5  5  5
```

UNPACK(VECTOR,MASK,FIELD)

Legge gli elementi di un array monodimensionale e li "scompatta" in un altro array (possibilmente maggiore) sotto il controllo di una *maschera*.

L'argomento VECTOR è un array monodimensionale di tipo qualsiasi e la cui dimensione deve essere almeno t, essendo t il numero di elementi .TRUE. di MASK.

MASK è un array logico. Esso determina dove saranno disposti gli elementi di VECTOR una volta scompattati.

`FIELD` deve essere un array dello *stesso tipo* e *parametri di kind* di `VECTOR` e *compatibile* con `MASK`. Gli elementi di `FIELD` vengono inseriti nell'array risultante in corrispondenza dei valori `.FALSE.` di `MASK`.

Il risultato è un array avente *stessa forma* di `MASK` e *stesso tipo* e *parametri di kind* di `VECTOR`. I suoi elementi vengono inseriti, nell'*array element order*, al modo seguente:

- Se l'elemento *i* di `MASK` ha valore `.TRUE.`, il corrispondente elemento del risultato è acquisito dal successivo elemento di `VECTOR`.
- Se l'elemento *i* di `MASK` ha valore `.FALSE.`, il corrispondente elemento del risultato è pari a `FIELD` (se `FIELD` è uno scalare) oppure è pari all'`i_`*mo* elemento di `FIELD` (se `FIELD` è un array).

Esempi:

Siano `n` l'array:

```
0  0  1
1  0  1
1  0  0
```

`p` l'array: `(/2,3,4,5/)`, e `q` l'array:

```
.TRUE.   .FALSE.  .FALSE.
.FALSE.  .TRUE.   .FALSE.
.TRUE.   .TRUE.   .FALSE.
```

L'istruzione `UNPACK(p,MASK=q,FIELD=n)` produrrà il risultato:

```
2  0  1
1  4  1
3  5  0
```

mentre l'istruzione `UNPACK(p,MASK=q,FIELD=1)` produrrà il risultato:

```
2  1  1
1  4  1
3  5  1
```

Un interessante esempio di utilizzo della funzione `PACK` è offerto dal seguente programma che valuta tutti i numeri primi compresi tra `1` `100` secondo il criterio noto come *crivello di Eratostene*:

```
PROGRAM Crivello_di_Eratostene
! Scopo: valutare i numeri primi secondo
!  il metodo di Eratostene
  IMPLICIT NONE
```

```
  INTEGER,PARAMETER :: last_number = 100
  INTEGER,DIMENSION(last_number) :: numbers
  INTEGER :: i, number_of_primes, ac
! Start
! Inizializza l'array con i valori (/0, 2, 3, ..., last_number/)
! NB: c'e' 0 al posto di 1 perché 1 rappresenta un caso particolare.
  numbers = (/0,(ac,ac=2,last_number)/)
  DO i=2,last_number
     IF (numbers(i) /= 0) THEN             ! if questo numero e' primo...
        numbers(2*i : last_number : i) = 0 ! ...se ne eliminano i multipli
     END IF
  END DO
! Conta i primi
  number_of_primes = COUNT(numbers/=0)
! Raccoglie i numeri primi nelle prime posizioni dell'array
  numbers(1:number_of_primes) = PACK(numbers, numbers /= 0)
! Li stampa
  PRINT "(A,I3,A)", "Numeri primi fra 1 e ",last_number,": "
  PRINT "(5I7)", numbers(1:number_of_primes)
END PROGRAM Crivello_di_Eratostene
```

Si riporta, per completezza, anche l'output del programma:

```
Numeri primi fra 1 e 100:
      2      3      5      7     11
     13     17     19     23     29
     31     37     41     43     47
     53     59     61     67     71
     73     79     83     89     97
```

6.6.4 Procedure di trasformazione

`RESHAPE(SOURCE,SHAPE[,PAD][,ORDER])`

Costruisce un array di forma differente a partire da un array di input.

L'argomento `SOURCE` è un array di tipo qualsiasi. Esso fornisce gli *elementi* per l'array risultante. La sua ampiezza deve essere maggiore o uguale a `PRODUCT(SHAPE)` se `PAD` è omesso oppure se ha ampiezza nulla.

L'argomento `SHAPE` è un array di sette elementi al massimo, rango unitario e dimensione costante. Esso definisce la *forma* dell'array risultante. Non può avere ampiezza nulla ed i suoi elementi non possono avere valori negativi.

L'argomento opzionale `PAD` è un array avente *stesso tipo* e *stessi parametri di kind* di `SOURCE`. Il suo compito è quello di fornire valori "di riserva" nel caso in cui l'array risultante avesse ampiezza maggiore di `SOURCE`.

L'argomento opzionale ORDER è un array della *stessa forma* di SHAPE. I suoi elementi rappresentano una permutazione della *n_pla* (/1,2,...,*n*/), essendo *n* l'ampiezza di SHAPE. Se ORDER è omesso, esso è assunto per default pari a (/1,2,...,*n*/).

Il risultato è un array di forma SHAPE avente *stesso tipo* e *parametri di kind* di SOURCE e ampiezza pari al prodotto dei valori degli elementi di SHAPE. All'interno dell'array risultante gli elementi di SOURCE sono disposti nell'ordine per dimensione così come specificato da ORDER. Se ORDER è omesso, gli elementi sono disposti secondo l'ordinamento standard.

Gli elementi di SOURCE sono seguiti (laddove necessario) dagli elementi di PAD (eventualmente in forma ripetuta) nell'ordine degli elementi di array.

Un semplice esempio di utilizzo della funzione RESHAPE è fornito dalla seguente istruzione:

```
RESHAPE((/3,4,5,6,7,8/),(/2,3/))
```

la quale fornisce come risultato la matrice:

```
3  5  7
4  6  8
```

mentre l'istruzione:

```
RESHAPE((/3,4,5,6,7,8/),(/2,4/),(/1,1/),(/2,1/))
```

fornisce il seguente risultato:

```
3  4  5  6
7  8  1  1
```

Il prossimo frammento di programma costituisce un ulteriore esempio di utilizzo della funzione RESHAPE:

```
INTEGER, DIMENSION(2,5) :: arr1
REAL, DIMENSION(5,3,8) :: f
REAL, DIMENSION(8,3,5) :: c
...
arr1 = RESHAPE((/1,2,3,4,5,6/),(/2,5/),(/0,0/),(/2,1/))
! questa istruzione fornisce:
!     [1 2 3 4 5]
!     [6 0 0 0 0]
c = RESHAPE(f,(/8,3,5/),ORDER=(/3,2,1/))
! questa istruzione assegna a c l'array f mutandone l'ordine degli elementi
END
```

6.6.5 Procedure topologiche

MAXLOC(ARRAY[,DIM][,MASK])

Restituisce la *posizione dell'elemento di valore massimo* di un array, di una sezione di array o lungo una specificata dimensione dell'array.

L'argomento `ARRAY` è un array di tipo intero o reale.

L'argomento opzionale `DIM` è uno scalare di tipo intero con valore compreso nell'intervallo $1 \div n$, essendo n il rango di `ARRAY`. Si noti che questo parametro *non* è previsto dal Fortran 90 ma soltanto dal Fortran 95.

L'argomento opzionale `MASK` è un array logico compatibile con `ARRAY`.

Il risultato della funzione è un array di tipo intero.

Nel caso in cui il parametro `DIM` sia omesso (come è sempre nel Fortran 90) valgono le seguenti regole:

- Se `MASK` è assente, gli elementi dell'array risultante rappresentano gli indici della locazione dell'elemento di `ARRAY` avente valore massimo.
- Se `MASK` è presente, gli elementi dell'array risultante rappresentano gli indici della locazione dell'elemento di `ARRAY` avente valore massimo fra quelli compatibili con la condizione specificata da `MASK`.

Nel caso in cui il parametro `DIM` sia presente (e, quindi, solo nel Fortran 95) valgono le seguenti regole:

- L'array risultante ha rango pari a quello di `ARRAY` diminuito di un'unità.
- Se `ARRAY` ha rango unitario, allora il valore di `MAXLOC(ARRAY,DIM[,MASK])` sarà pari a quello di `MAXLOC(ARRAY[,MASK])`.
- Se più elementi hanno valore pari al valore massimo, è l'indice del *primo* di tali elementi (secondo l'*ordine degli elementi di array*) quello che viene restituito.
- Se `ARRAY` ha ampiezza nulla, oppure se tutti gli elementi di `MASK` hanno valore `.FALSE.`, il valore del risultato è indefinito.

Esempi:

Il valore di `MAXLOC((/3,7,4,7/))` è `2`, essendo questo l'indice della posizione della prima occorrenza del valore massimo (`7`) nell'array monodimensionale specificato come argomento.

Sia `mat` l'array:

```
 4   0  -3   2
 3   1  -2   6
-1  -4   5  -5
```

`MAXLOC(mat,MASK=mat<5)` fornisce il valore `(/1,1/)` poiché questi sono gli indici che puntano all'elemento di valore massimo (`4`) fra tutti quelli minori di `5`.

`MAXLOC(mat,DIM=1)` fornisce il valore `(/1,2,3,2/)`, essendo `1` l'indice di riga del massimo valore di `mat` nella prima colonna (`4`), `2` l'indice di riga del massimo valore di `mat` nella seconda colonna (`1`), e così via.

`MAXLOC(mat,DIM=2)` fornisce il valore `(/1,4,3/)`, essendo `1` l'indice di colonna del massimo valore di `mat` nella prima riga (`4`), `4` l'indice di colonna del massimo valore di `mat` nella seconda riga, e così via.

`MINLOC(ARRAY[,DIM][,MASK])`

Restituisce la *posizione dell'elemento di valore minimo* di un array, di una sezione di array o lungo una specificata dimensione dell'array.

L'argomento `ARRAY` è un array di tipo intero o reale.

L'argomento opzionale `DIM` è uno scalare di tipo intero con valore compreso nell'intervallo $1 \div n$, essendo n il rango di `ARRAY`. Si noti che questo parametro non è previsto dal Fortran 90 ma soltanto dal Fortran 95.

L'argomento opzionale `MASK` è un array logico compatibile con `ARRAY`.

Il risultato è un array di tipo intero.

Nel caso in cui il parametro `DIM` sia omesso (come è sempre nel Fortran 90) valgono le seguenti regole:

- L'array risultante ha rango unitario e ampiezza pari al rango di `ARRAY`.
- Se `MASK` è assente, gli elementi dell'array risultante rappresentano gli indici della locazione dell'elemento di `ARRAY` avente valore minimo.
- Se `MASK` è presente, gli elementi dell'array risultante rappresentano gli indici della locazione dell'elemento di `ARRAY` avente valore minimo fra quelli compatibili con la condizione specificata da `MASK`.

Nel caso in cui il parametro `DIM` sia presente (e, quindi, solo nel Fortran 95) valgono le seguenti regole:

- L'array risultante ha rango pari quello di `ARRAY` diminuito di un'unità.
- Se `ARRAY` ha rango uno allora il valore di `MINLOC(ARRAY,DIM[,MASK])` sarà pari a quello di `MINLOC(ARRAY[,MASK])`.
- Se più elementi hanno valore pari al valore minimo, è l'indice del *primo* di tali elementi (secondo l'*ordine degli elementi di array*) quello che viene restituito.
- Se `ARRAY` ha ampiezza nulla, oppure se tutti gli elementi di `MASK` hanno valore `.FALSE.`, il valore del risultato è indefinito.

Esempi:

Il valore di `MINLOC((/3,1,4,2/))` è `2`, essendo questo l'indice della posizione della prima occorrenza del valore minimo nell'array monodimensionale specificato come argomento.

Sia `mat` l'array:

```
 4   0  -3   2
 3   1  -2   6
-1  -4   5  -5
```

`MINLOC(mat,MASK=mat>-5)` fornisce il valore `(/3,2/)` poiché questi sono gli indici che puntano all'elemento di valore minimo (`-4`) fra tutti quelli maggiori di `-5`.

`MINLOC(mat,DIM=1)` fornisce il valore `(/3,3,1,3/)`, essendo `3` l'indice di riga del minimo valore di `mat` nella prima colonna (`-1`), `3` l'indice di riga del minimo valore di `mat` nella seconda colonna (`-4`), e così via.

`MINLOC(mat,DIM=2)` fornisce il valore `(/3,3,4/)`, essendo `3` l'indice di colonna del minimo valore di `mat` nella prima riga (`-3`), `3` l'indice di colonna del minimo valore di `mat` nella seconda riga (`-2`), e così via.

6.6.6 Procedure di manipolazione

`CSHIFT(ARRAY,SHIFT[,DIM])`

Esegue uno *shift circolare* su un array di rango unitario o, se applicato ad un array di rango superiore, uno *shift circolare* ad intere sezioni di rango uno (*vettore*) lungo una data dimensione dell'array.

Gli elementi shiftati al di fuori delle dimensioni dell'array vengono spostati all'altra estremità dell'array.

Sezioni differenti possono essere shiftate in misura diversa oltre che in differenti direzioni.

Per quanto concerne gli argomenti, `ARRAY` può essere un array di qualsiasi rango e tipo; `SHIFT` può essere uno scalare o anche un array (nel qual caso deve avere rango minore di un'unità rispetto ad `ARRAY`) di tipo `INTEGER`; il parametro opzionale `DIM` è, invece, uno scalare intero con valore compreso tra `1` ed n, con n il rango di `ARRAY` (`DIM` è assunto, per default, pari ad uno).

Il risultato della funzione è un array avente *stesso tipo*, *parametri di kind* e *forma* di `ARRAY`. In particolare:

- Se `ARRAY` ha rango unitario, allora l'`i_`*mo* elemento dell'array risultante sarà pari a `ARRAY(1+MODULO(i+SHIFT-1,SIZE(ARRAY)))`. (Il medesimo shift è applicato a ciascun elemento).
- Se `ARRAY` ha rango maggiore di uno, ciascuna sezione dell'array risultante è shiftato del valore di `SHIFT` se quest'ultimo è uno *scalare*, oppure del corrispondente valore in `SHIFT`, se `SHIFT` è un array.
- Il valore di `SHIFT` determina l'entità e la direzione dello shift. Un valore positivo di `SHIFT` provoca uno spostamento a sinistra (nelle righe) o verso l'alto (nelle colonne). Un valore negativo di `SHIFT`, al contrario, provoca uno spostamento a destra (nelle righe) o verso il basso (nelle colonne). Un valore nullo di `SHIFT`, chiaramente, non determina alcuna variazione.

Esempi:

Sia `v` l'array monodimensionale `(/1,2,3,4,5,6/)`. Si ha quanto segue:

`CSHIFT(v,SHIFT=2)` trasla gli elementi di `v` in maniera circolare verso sinistra di due posizioni, producendo il vettore: `(/3,4,5,6,1,2/)`. Come si può notare, gli elementi `1` e `2` sono strappati dalle loro posizioni di testa e inseriti in coda all'array.

`CSHIFT(v,SHIFT=-2)` trasla gli elementi di `v` in maniera circolare verso destra di due posizioni, producendo il vettore: `(/5,6,1,2,3,4/)`. Ancora, si noti come gli elementi `5` e `6` siano stati strappati dalla coda dell'array e inseriti alla sua testa.
Sia, ora, `m` il seguente array bidimensionale:

```
1  2  3
4  5  6
7  8  9
```

L'istruzione:

```
CSHIFT(m,SHIFT=1,DIM=2)
```

shifta a sinistra, di due posizioni, ciascun elemento nelle tre righe, producendo il seguente risultato:

```
2  3  1
5  6  4
8  9  7
```

Invece, l'istruzione:

```
CSHIFT(m,SHIFT=-1,DIM=1)
```

shifta verso il basso, di una sola posizione, ciascun elemento nelle tre colonne, producendo il seguente risultato:

```
7  8  9
1  2  3
4  5  6
```

Infine, l'istruzione:

```
CSHIFT(m,SHIFT=(/1,-1,0/),DIM=2)
```

shifta ogni elemento appartenente alla prima riga di una posizione verso sinistra, ogni elemento appartenente alla seconda riga di due posizioni verso destra, *non* sposta alcun elemento della terza riga. Il risultato prodotto sarà, pertanto:

```
2  3  1
6  4  5
7  8  9
```

`EOSHIFT(ARRAY,SHIFT[,BOUNDARY][,DIM])`

Esegue uno *shift* di tipo *end-off* su un array di rango uno o, se applicata ad un array di rango superiore, uno *shift* di tipo *end-off* ad intere sezioni di rango uno (*vettori*) lungo una data dimensione dell'array.

Alcuni elementi sono shiftati al di fuori delle dimensioni dell'array e copie degli elementi del contorno (*boundary*) vengono inseriti all'altra estremità dell'array.

Sezioni differenti possono essere shiftate in misura diversa oltre che in differenti direzioni.

Relativamente agli argomenti, si ha quanto segue:

- L'argomento `ARRAY` può essere un array di qualsiasi rango e tipo.

- L'argomento `SHIFT` può essere uno scalare o anche un array (nel qual caso deve avere rango minore di un'unità rispetto a `ARRAY`) di tipo `INTEGER`.

- L'argomento opzionale `BOUNDARY` deve avere *stesso tipo* e *parametri di kind* di `ARRAY`. Esso può essere uno scalare oppure un array il cui rango sia par a quello di `ARRAY` diminuito di uno. Se il parametro `BOUNDARY` è assente, sono assunti i seguenti valori di default, dipendenti dal *tipo* di `ARRAY`:

tipo di `ARRAY`	*valore di* `BOUNDARY`
`INTEGER`	`0`
`REAL`	`0.0`
`COMPLEX`	`(0.0,0.0)`
`LOGICAL`	`.FALSE.`
`CHARACTER(LEN=`n`)`	n spazi bianchi

- Il parametro opzionale `DIM` è, invece, uno scalare intero con valore compreso tra `1` ed n, con n il rango di `ARRAY` (`DIM` è assunto, per default, pari ad uno).

Il risultato è un array avente *stesso tipo*, *parametri di kind* e *forma* di `ARRAY`. In particolare:

- Se `ARRAY` ha rango unitario, il medesimo shift è applicato a ciascun elemento. Se qualche elemento è strappato via dall'array, esso sarà rimpiazzato dal valore di `BOUNDARY`.

- Se `ARRAY` ha rango maggiore di uno, ciascuna sezione dell'array risultante è shiftato del valore di `SHIFT` se quest'ultimo è uno scalare, del corrispondente valore in `SHIFT`, se `SHIFT` è un array.

- Se un elemento è spostato fuori dall'array, il valore di `BOUNDARY` è inserito all'altra estremità della sezione di array.

- Il valore di `SHIFT` determina l'entità e la direzione dello shift. Un valore positivo di `SHIFT` provoca uno spostamento a sinistra (nelle righe) o verso l'alto (nelle colonne). Un valore negativo di `SHIFT`, al contrario, provoca uno spostamento a destra (nelle righe) o verso il basso (nelle colonne). Un valore nullo di `SHIFT`, chiaramente, non determina alcuna variazione.

Esempi:

Sia `v` l'array monodimensionale `(/1,2,3,4,5,6/)`. Si ha quanto segue:

`EOSHIFT(v,SHIFT=2)` trasla gli elementi di `v` verso sinistra di due posizioni, producendo il vettore: `(/3,4,5,6,0,0/)`. Come si può notare, gli elementi `1` e `2` sono strappati dalle loro posizioni di testa e sostituiti, in coda al vettore, dal valore di default `0`.

`EOSHIFT(v,SHIFT=-3,BOUNDARY=99)` trasla gli elementi di `v` verso destra di tre posizioni, producendo il vettore: `(/99,99,99,2,3,4/)`. Ancora, si noti come gli elementi `4`, `5` e `6` siano stati strappati dalla coda dell'array e rimpiazzati, in testa all'array, dal valore di `BOUNDARY`.
Sia, ora, `m` il seguente array bidimensionale:

```
1  2  3
4  5  6
7  8  9
```

L'istruzione:

```
EOSHIFT(m,SHIFT=1,BOUNDARY='9',DIM=2))
```

shifta a sinistra, di una posizione, ciascun elemento nelle tre righe, rimpiazzando con il valore `9` gli elementi che rimangono "scoperti", producendo, così, il seguente risultato:

```
2  3  9
5  6  9
8  9  9
```

Invece, l'istruzione:

```
EOSHIFT(EOSHIFT (m,SHIFT=-1,DIM=1)
```

shifta verso il basso, di una sola posizione, ciascun elemento nelle tre colonne, e inserisce l'elemento `0` (corrispondente al valore di default di `BOUNDARY`) negli elementi della terza riga, producendo, pertanto, il seguente risultato:

```
1  2  3
4  5  6
0  0  0
```

Infine, l'istruzione:

```
EOSHIFT(M,SHIFT=(/1,-1,0/),BOUNDARY=(/'*','?','/'/),DIM=2)
```

applicata all'array:

```
'a'  'b'  'c'
'd'  'e'  'f'
'g'  'h'  'i'
```

shifta ogni elemento appartenente alla prima riga di una posizione verso sinistra (inserendo il carattere `'*'` in coda alla riga), ogni elemento appartenente alla seconda riga di una posizione verso destra (inserendo il carattere `'?'` in testa alla riga), *non* sposta alcun elemento della terza riga (non facendo, quindi, uso, dell'ultimo elemento, di `BOUNDARY`, `'/'`). Il risultato prodotto sarà, pertanto:

```
'a'  'b'  '*'
'?'  'e'  'f'
'g'  'h'  'i'
```

`TRANSPOSE(MATRIX)`

Traspone un array di rango due.

L'argomento `MATRIX` può essere un *array bidimensionale* di tipo qualsiasi.

Il risultato della funzione è un array avente *stesso tipo* e *stessi parametri di kind* dell'argomento `MATRIX` e forma `(/n,m/)` essendo `(/m,n/)` la forma dell'argomento.

L'elemento `(i,j)`_*mo* dell'array risultante coincide con il valore di `MATRIX(j,i)`.

A titolo di esempio, definita la seguente matrice `mat`:

```
1  2  3
4  5  6
7  8  9
```

`TRANSPOSE(mat)` avrà valore:

```
1  4  7
2  5  8
3  6  9
```

6.6.7 Procedure algebriche

`DOT_PRODUCT(VECTOR_A,VECTOR_B)`

Esegue il *prodotto scalare* di due *vettori* numerici (interi, reali o complessi) o logici aventi *stessa dimensione*.

Il risultato è uno scalare il cui *tipo* dipende dai vettori operandi `VECTOR_A` e `VECTOR_B`. In particolare:

- Se l'argomento `VECTOR_A` è di tipo `INTEGER` o `REAL` allora il risultato avrà valore pari a `SUM(VECTOR_A*VECTOR_B)`.
- Se l'argomento `VECTOR_A` è di tipo `COMPLEX`, allora il risultato avrà valore pari a `SUM(CONJG(VECTOR_A)*VECTOR_B)`.
- Se l'argomento `VECTOR_A` è di tipo `LOGICAL`, allora il risultato avrà valore pari a `ANY(VECTOR_A.AND.VECTOR_B)`.

Se gli array operandi hanno dimensione nulla, il risultato vale zero se essi sono di tipo numerico oppure `.FALSE.` se gli operandi sono di tipo logico.

Esempi:

L'istruzione:

```
DOT_PRODUCT((/1,2,3/),(/3,4,5/))
```

restituisce il valore 26 (infatti: (1×3)+(2×4)+(3×5)=26), l'istruzione:

```
DOT_PRODUCT((/(1.0,2.0),(2.0,3.0)/),(/(1.0,1.0),(1.0,4.0)/))
```

fornisce l'array `(/17.0,4.0/)`, mentre l'istruzione:

```
DOT_PRODUCT((/.TRUE.,.FALSE./),(/.FALSE.,.TRUE./))
```

fornisce il valore `.FALSE.`

La subroutine che segue rappresenta un interessante esempio di utilizzo della procedura intrinseca `DOT_PRODUCT`. Il suo scopo è quello di valutare i coefficienti della retta che interpola una dispersione di dati (x, y) forniti in forma di array, mediante la tecnica dei *minimi quadrati*. A tal proposito si ricorda che, detta $y = mx + c$ la retta interpolante, i coefficienti m e c che garantiscono il miglior fitting dei dati sono valutabili tramite le relazioni:

$$m = \frac{n\sum_{i=1}^{n} x_i y_i - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}$$

$$c = \frac{\sum_{i=1}^{n} y_i - m\sum_{i=1}^{n} x_i}{n}$$

In sintesi, ecco quale è l'effetto della subroutine `least_sq`:

1. Riceve il numero `n` delle coppie di dati (`x`,`y`) ed i relativi valori dall'unità chiamante.

2. Calcola le sommatorie di `x`, `y`, `xy` x^2 usando allo scopo le funzioni intrinseche `SUM` e `DOT_PRODUCT`.

3. Calcola il valore dei coefficienti `m` e `c`.

4. Restituisce in uscita i valori di `m` e di `c` all'unità chiamante.

```
SUBROUTINE least_sq(x,y,n,m,c)
! Scopo: calcolare i coefficienti della retta interpolante
!        ai minimi quadrati della dispersione di dati x-y
   IMPLICIT NONE
! Variabili dummy
   INTEGER, INTENT(IN) :: n
   REAL, DIMENSION(n),INTENT(IN) :: x, y
   REAL, INTENT(OUT) :: m, c
! Variabili locali
   REAL :: sum_x, sum_xsq, sum_xy, sum_y
! Calcolo delle sommatorie
      sum_x=SUM(x)
      sum_y=SUM(y)
      sum_xy=DOT_PRODUCT(x,y)
      sum_xsq=DOT_PRODUCT(x,x)
! Calcolo dei coefficienti
      m=(n*sum_xy-sum_x*sum_y)/(n*sum_xsq-sum_x*sum_x)
      c=(sum_y-m*sum_x)/n
END SUBROUTINE least_sq
```

`MATMUL(MATRIX_A,MATRIX_B)`

Esegue il prodotto *righe*×*colonne* di due matrici numeriche o logiche.

Gli argomenti `MATRIX_A` che `MATRIX_B` devono essere array dello *stesso tipo* e di *rango uno o due* (almeno uno dei due, però, *deve* avere rango due) ed inoltre è necessario che l'ampiezza della prima (o dell'unica) dimensione di `MATRIX_B` sia uguale all'ampiezza dell'ultima (o dell'unica) dimensione di `MATRIX_A`.

Il rango e la forma, invece, dipendono dal rango e dalla forma degli argomenti, alla maniera seguente:

- Se `MATRIX_A` ha forma `(/n,m/)` e `MATRIX_B` ha forma `(/m,k/)`, allora il risultato sarà un array di rango due di forma `(/n,k/)`.

- Se `MATRIX_A` ha forma `(/m/)` e `MATRIX_B` ha forma `(/m,k/)`, allora il risultato sarà un array di rango uno di forma `(/k/)`.

- Se `MATRIX_A` ha forma `(/n,m/)` e `MATRIX_B` ha forma `(/m/)`, allora il risultato sarà un array di rango uno di forma `(/n/)`.

- Se gli argomenti sono di *tipo numerico*, allora l'elemento *`(i,j)_mo`* dell'array risultato avrà valore pari a `SUM((`*`riga_i`* `di MATRIX_A)*(colonna_j di MATRIX_B))`.

- Se gli argomenti sono di tipo *logico*, allora l'elemento `(i,j)_`*mo* dell'array risultato avrà valore pari a `ANY((`*`riga_i di`* `MATRIX_A).AND.(`*`colonna_j`* `di MATRIX_B))`.

Esempi:

Siano `a` la matrice:

```
2  3  4
3  4  5
```

`b` la matrice:

```
2  3
3  4
4  5
```

`x` il vettore: `(/1,2/)` e `y` il vettore: `(/1,2,3/)`.
Il risultato di `MATMUL(a,b)` è la matrice:

```
29  38
38  50
```

Il risultato di `MATMUL(x,a)` è il vettore `(/8,11,14/)`.
Il risultato di `MATMUL(a,y)` è il vettore `(/20,26/)`.

Il programma che segue vuole essere un riepilogo di alcuni dei principali argomenti trattati in questo capitolo. Allo scopo viene presentata l'implementazione del classico *algoritmo di Crout* per la risoluzione di un sistema di equazioni algebriche lineari. Al fine di agevolare la comprensione del codice, si ricorda che, dato un sistema di equazioni scritto in forma matriciale come:

$$Ax = b$$

il metodo in esame procede preliminarmente ad una *fattorizzazione LU* della matrice A dei coefficienti in modo da trasformare il problema precedente nella determinazione della soluzione dei due sistemi triangolari:

$$Ly = b$$
$$Ux = y$$

per poi passare a risolvere questi ultimi ottenendo, infine, la soluzione x del sistema dato.

```
MODULE Crout_Elimination
! Scopo: Risolvere un sistema lineare con il metodo di Crout
   IMPLICIT NONE
CONTAINS
   SUBROUTINE Solve(A,b,x,n)
      REAL, INTENT (IN) :: A(:,:),b(:)  ! Assumed-shape array
      REAL, INTENT (OUT) :: x(:)
      INTEGER, INTENT (IN) :: n
      REAL :: LU(n,n), y(n)             ! Automatic-shape array
      INTEGER :: m
  ! Start subroutine Solve
      LU = A
    ! Esegue la fattorizzazione A = L*U senza pivoting.
      DO m=1,n
         LU(m:n,m) = LU(m:n,m)-MATMUL(LU(m:n,1:m-1),LU(1:m-1,m))
         LU(m,m+1:n) = (LU(m,m+1:n) &
                        - MATMUL(LU(m,1:m-1),LU(1:m-1,m+1:n)))/LU(m,m)
      END DO
    ! Esegue una "forward substitution" per risolvere Ly = b.
      DO m=1,n
         y(m) = b(m)-DOT_PRODUCT(LU(m,1:m-1),y(1:m-1))/LU(m,m)
      END DO
    ! Esegue una "backward substitution" per risolvere Ux = w.
      DO m=n,1,-1
         x(m) = y(m)-DOT_PRODUCT(LU(m,m+1:n),x(m+1:n))
      END DO
      RETURN
   END SUBROUTINE Solve
END MODULE Crout_Elimination

PROGRAM TestCrout
  USE Crout_Elimination
  IMPLICIT NONE
  INTEGER :: n,i
  REAL, ALLOCATABLE :: A(:,:),b(:),x(:)
! Start program TestCrout
  WRITE(*,'(1X,A)',ADVANCE='NO') " Inserire il numero di equazioni: "
  READ(*,*) n
  ALLOCATE(A(n,n),b(n),x(n))  ! Allocatable array
  WRITE(*,*) " Inserire la matrice dei coefficienti (per righe): "
  DO i=1,n
     READ(*,*) A(i,:)  ! Legge la matrice A per righe.
  END DO
```

```
  WRITE(*,*) " Inserire la colonna dei termini noti: "
  READ(*,*) b
  WRITE(*,*)
  WRITE(*,*) " Dati introdotti: "
  DO i=1,n
     WRITE(*,'(1X,A17,I3,A3,<n>(1X,F8.5))') " Matrice A, riga ",i,": ",A(i,:)
  END DO
  WRITE(*,'(1X,A25,3X,<n>(1X,F8.5))') " Colonna termini noti b: ", b
  CALL Solve(A,b,x,n)
  WRITE(*,*)
  WRITE(*,'(1X,A25,3X,<n>(1X,F8.5))') " Vettore soluzione x: ", x
  STOP
END PROGRAM TestCrout
```

Si riporta, per completezza, anche un esempio di impiego del programma:

```
C:\MYPROG>testcrout
 Inserire il numero di equazioni: 3
 Inserire la matrice dei coefficienti (per righe):
11 3 2
 1 9 0
 3 2 8
  Inserire la colonna dei termini noti:
-1 2 1

  Dati introdotti:
  Matrice A, riga   1 :  11.00000  3.00000  2.00000
  Matrice A, riga   2 :   1.00000  9.00000  0.00000
  Matrice A, riga   3 :   3.00000  2.00000  8.00000
  Colonna termini noti b:  -1.00000  2.00000  1.00000

  Vettore soluzione x:     -1.77676  2.13681  1.06698
```

L'esempio che segue rappresenta una ulteriore dimostrazione della utilità, semplicità e immediatezza di impiego di alcune funzioni intrinseche operanti su array. Il programma implementa un ulteriore algoritmo di risoluzione di un sistema di equazioni lineari, vale a dire il *metodo di eliminazione di Gauss*, che consiste nel trasformare il sistema di partenza:

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + ... + a_{1,n}x_n = b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + ... + a_{2,n}x_n = b_2 \\ \qquad\qquad\qquad ... \\ a_{n,1}x_1 + a_{n,2}x_2 + ... + a_{n,n}x_n = b_n \end{cases}$$

in un equivalente sistema triangolare alto:

$$\begin{cases} c_{1,1}x_1 + c_{1,2}x_2 + ... + c_{1,n}x_n = d_1 \\ c_{2,2}x_2 + ... + c_{2,n}x_n = d_2 \\ ... \\ c_{n,n}x_n = d_n \end{cases}$$

immediatamente invertibile per *back-substitution*. Affinché il criterio sia applicabile è necessario che in ciascuna riga l'elemento di modulo massimo sia portato sulla diagonale principale (*pivoting*).

```
MODULE solve_module
   PUBLIC :: solve_linear_equations
CONTAINS
   SUBROUTINE solve_linear_equations (a,x,b,error)
      REAL,DIMENSION(:,:),INTENT (IN) :: A
      REAL,DIMENSION(:),INTENT (OUT) :: x
      REAL,DIMENSION(:),INTENT (IN) :: b
      LOGICAL,INTENT(OUT) :: error
      REAL,DIMENSION(:,:),ALLOCATABLE :: m
      REAL,DIMENSION(:),ALLOCATABLE :: temp_row
      INTEGER,DIMENSION (1) :: max_loc
      INTEGER :: n, k, k_swap
         n = SIZE(b)
         error = (SIZE(A,DIM=1)/=n).OR.(SIZE(A,dim=2)/=n)
         IF (error) THEN
            x = 0.0
         ELSE
            ALLOCATE(m(n,n+1))
            ALLOCATE(temp_row(n+1))
            m(1:n,1:n) = A
            m(1:n,n+1) = b
! Fase di triangolarizzazione
            triang_loop: DO k =1,n
! Spostamento dell'elemento di valore assoluto massimo
!  sulla diagonale principale
               max_loc = MAXLOC(ABS(m(k:n,k)))
               k_swap = k-1+max_loc(1)
               temp_row(k:n+1) = m(k,k:n+1)
               m(k,k:n+1) = m(k_swap,k:n+1)
               m(k_swap,k:n+1) = temp_row(k:n+1)
               IF (m(k,k)==0) THEN
                  error = .TRUE.
                  EXIT triang_loop
```

```
                ELSE
                   m(k,k:n+1) = m(k,k:n+1)/m(k,k)
                   m(k+1:n,k+1:n+1) = m(k+1:n,k+1:n+1) -  &
                     SPREAD(m(k,k+1:n+1),1,n-k)*SPREAD(m(k+1:n,k),2,n-k+1)
                END IF
             END DO triang_loop
! Fase di sostituzione all'indietro
             IF (error) THEN
                x = 0.0
             ELSE
                DO k=n,1,-1
                   x(k) = m(k,n+1)-SUM(m(k,k+1:n)*x(k+1:n))
                END DO
             END IF
             DEALLOCATE(m,temp_row)
          END IF
   END SUBROUTINE solve_linear_equations
END MODULE solve_module

PROGRAM p_solve
  USE solve_module
  REAL, DIMENSION(3,3) :: A
  REAL, DIMENSION(3) :: b, x
  INTEGER :: i, j
  LOGICAL :: error
! costruzione di una matrice di coefficienti di prova
  DO i=1,3
     DO j=1,3
        a(i,j) = i+j
     END DO
  END DO
  A(3,3) = -A(3,3)
! costruzione della colonna dei termini noti
  b = (/20,26,-4/)
  CALL solve_linear_equations(A,x,b,error)
  IF (error==.FALSE.) THEN
     PRINT*, "La soluzione e': ", x
  ELSE
     PRINT*, "Errore nella matrice dei coefficienti"
  END IF
END PROGRAM p_solve
```

Capitolo 7

Puntatori

Oltre a consentire la dichiarazione di oggetti di tipi di dati intrinseci o di un tipo di dati definito dall'utente, il Fortran 90/95 prevede anche uno speciale tipo di oggetto che, contrariamente agli altri, non contiene un *valore* ma l'*indirizzo di memoria* di un'altra variabile il cui valore è quello effettivamente immagazzinato. Poiché questo tipo di variabile *punta* ad un'altra variabile, esso è chiamato **puntatore**, mentre la variabile a cui esso punta è detta **target**.

In realtà, contrariamente a quanto avviene, ad esempio, nel il linguaggio C, i puntatori del Fortran sono semplicemente variabili definite con l'attributo `POINTER` e non rappresentano un tipo di dati distinto, per cui, in particolare, non è possibile alcuna aritmetica dei puntatori. Dato un puntatore, esempi di *target* validi sono:

- Variabili dello *stesso tipo* del puntatore ed esplicitamente dichiarate con l'*attributo* `TARGET`.
- Altri *puntatori* dello *stesso tipo* di quello dato.
- *Memoria dinamica* allocata con il puntatore stesso.

L'uso dei puntatori spesso fornisce una alternativa più flessibile all'uso degli *allocatable array* ed un efficiente strumento per creare e manipolare *strutture dati dinamiche* quali *liste* e *alberi binari*.

I puntatori vengono usati soprattutto in quelle situazioni dove variabili ed array devono essere creati e distrutti "dinamicamente" durante l'esecuzione di un programma e quando non sono noti, se non durante l'esecuzione stessa del programma, il numero e le dimensioni delle variabili di un dato tipo necessarie durante il *run*. Per questa ragione le variabili con attributo `POINTER` sono dette variabili *dinamiche*, per distinguerle dalle variabili tradizionali, dette *statiche*, le quali sono così chiamate perché il loro numero, il tipo e le dimensioni devono essere noti in fase di compilazione e restare inalterati durante l'intera esecuzione del programma.

7.1 Dichiarazione di puntatori e target

La forma generale delle istruzioni di dichiarazione di un *puntatore* e di un *target* sono le seguenti:

```
tipo, POINTER [, attributi] ::  nome_puntatore
tipo, TARGET [, attributi] ::  nome_target
```

in cui:

- *tipo* è il tipo di dati che può essere puntato, e può essere sia un tipo intrinseco che uno derivato.
- *attributi* è una eventuale lista di attributi del puntatore o del target.

E' possibile, inoltre dichiarare variabili puntatore o target anche a mezzo delle *istruzioni* `POINTER` e `TARGET`:

```
tipo [, attributi] [::]  nome_puntatore
tipo [, attributi] [::]  nome_target
POINTER ::  nome_puntatore
TARGET ::  nome_puntatore
```

Si noti che il *tipo* di un puntatore deve essere specificato anche se il puntatore stesso non ospita alcun dato di quel tipo: questa informazione serve solo ad indicare che il puntatore contiene l'*indirizzo* di una variabile di *quel* particolare tipo. Per quanto concerne, invece, la lista degli attributi, è bene precisare che:

- L'attributo `POINTER` è incompatibile con gli attributi `ALLOCATABLE`, `EXTERNAL`, `INTENT`, `INTRINSIC`, `PARAMETER` e, ovviamente, `TARGET`.
- L'attributo `TARGET` è incompatibile con gli attributi `EXTERNAL`, `INTRINSIC`, `PARAMETER` e, ovviamente, `POINTER`.

Un puntatore deve avere lo stesso *tipo* e lo stesso *rango* del suo *target*. Un qualsiasi tentativo di puntare ad una variabile di tipo o rango diverso produce un *errore di compilazione*. Tuttavia, per *puntatori ad array* l'istruzione di dichiarazione deve specificare unicamente il *rango* ma *non* le *dimensioni*, in questo essendo del tutto simile ad un array *allocabile*: un array a puntatore è dichiarato, pertanto, con una specifica di *deferred-shape array*, ossia in fase di compilazione viene fornito solo il numero delle dimensioni, mentre l'ampiezza di ognuna di tali dimensioni viene sostituita dal simbolo di due punti. Due esempi di possibili puntatori ad array sono i seguenti:

```
INTEGER, POINTER, DIMENSION(:) :: pt1
REAL, POINTER, DIMENSION(:,:) :: pt2
```

di cui il primo puntatore può puntare unicamente ad array monodimensionali di tipo intero, il secondo unicamente ad array bidimensionali di tipo reale.

Le seguenti linee di codice rappresentano tre esempi di dichiarazione di puntatori e relativi target:

```
REAL, POINTER :: pt
REAL, TARGET :: a=3

INTEGER, POINTER, DIMENSION(:) :: pt1, pt2
INTEGER, TARGET :: x(5), y(10), z(100)

REAL, POINTER, DIMENSION(:,:,:) :: punt
REAL, TARGET :: targ(:,:,:)
```

E', invece, *illegale* una dichiarazione nella quale si forniscano, oltre al rango, anche le dimensioni del puntatore ad array:

```
REAL, POINTER, DIMENSION(5,5) :: pt     ! illegale
```

Un puntatore può anche apparire come componente di un *tipo di dati derivato*, contrariamente agli *allocatable array* ai quali questa possibilità è inibita. A titolo di esempio si può considerare il seguente frammento di programma:

```
TYPE data
    REAL, POINTER :: a(:)
END TYPE data
TYPE(data) :: event(3)
...
DO i=1,3
   READ(*,*) n
   ALLOCATE(event(i)%a(n))
   READ(*,*) event(i)%a
END DO
...
DO i=1,3
   DEALLOCATE(event(i)%a)
END DO
```

Inoltre, i puntatori che siano componenti di una variabile di tipo derivato possono puntare anche al tipo stesso di dati derivato che si sta creando: questa caratteristica è molto utile quando si vogliano creare delle strutture dati dinamiche. Un possibile esempio di applicazione di questa particolare caratteristica è il seguente:

```
TYPE :: valore_reale
    REAL :: valore
    TYPE(valore_reale) :: pnt
END TYPE
```

7.2 Istruzioni di assegnazione di puntatori

Un *puntatore* può essere associato ad un dato *target* a mezzo di una *istruzione di assegnazione di puntatore*:

```
nome_puntatore => nome_target
```

Quando la precedente istruzione è eseguita, l'*indirizzo di memoria* del target viene immagazzinato nel puntatore: da questo momento, ogni riferimento al puntatore avrà lo stesso effetto di un riferimento ai dati immagazzinati nel target.

Se un puntatore già associato ad un target viene usato in una nuova istruzione di assegnazione, allora l'associazione con il primo target viene perduta, sostituita dalla nuova associazione.

In una istruzione di assegnazione è anche possibile assegnare il valore di un puntatore ad un altro puntatore:

```
puntatore1 => puntatore2
```

Come effetto di questa istruzione, entrambi i puntatori punteranno *direttamente* ed *indipendentemente* allo stesso target. Se uno dei due puntatori viene successivamente associato ad un diverso target l'altro puntatore *non* sarà affetto da tale cambiamento. Tuttavia, se ***puntatore2*** viene *deassociato* anche ***puntatore1*** assumerà lo stesso stato.

A titolo di esempio si consideri il seguente insieme di istruzioni:

```
REAL, TARGET :: x
REAL, POINTER :: p1, p2
x = 3.14
p1 => x
p2 => p1
x = 2.72
```

Se si esegue questo frammento di programma, sia `p1` che `p2` punteranno a `x`, per cui le istruzioni seguenti:

```
WRITE(*,*) x
WRITE(*,*) p1
WRITE(*,*) p2
```

forniranno tutte lo stesso risultato: `2.72`. Se ora si apporta una piccola modifica alle righe precedenti:

```
REAL, TARGET :: x, y
REAL, POINTER :: p1, p2
x = 3.14
y = 2.72
p1 => x
p2 => y
```

il nuovo valore di `x` e di `p1` sarà `3.14` mentre il nuovo valore di `y` e `p2` sarà `2.72`. Se a questo punto si esegue l'istruzione di assegnazione:

```
p2 = p1
```

tutte e quattro le variabili avranno valore `3.14`. Ciò significa che l'istruzione di assegnazione di puntatori ha lo stesso effetto della assegnazione delle variabili a cui essi puntano. In altre parole si sarebbe avuto lo stesso risultato se si fosse posto:

```
y = x
```

Se, invece, si fosse avuto:

```
p2 => p1
```

si sarebbe fatto in modo che sia `p1` che `p2` puntassero alla stessa variabile `x` mentre alla variabile `y` non sarebbe stato riferito alcun puntatore, con il risultato che `p1`, `p2` e `x` avrebbero avuto valore `3.14` mentre `y` avrebbe mantenuto il valore `2.72`. Questo esempio fa comprendere come ogni qualvolta un puntatore sia usato in una *espressione* al posto di un dato "effettivo", nell'espressione viene usato il valore del *target* a cui il puntatore è riferito. Questo processo è noto come *dereferencing* del puntatore: un puntatore, pertanto, è da riguardarsi semplicemente come un *alias* del suo *target*. Allo stesso modo, tutte le volte che un puntatore appare come un *alias* di un'altra variabile, esso subisce automaticamente il *dereferencing* ossia è il valore del suo target che viene usato al posto del puntatore stesso. Ovviamente, affinché ciò possa avvenire è necessario che il puntatore sia associato ad un target. I puntatori sono automaticamente deriferiti quando compaiono all'interno di una espressione o nelle istruzioni di I/O. Il seguente frammento di programma aiuterà meglio a comprendere il senso di quanto detto:

```
REAL, TARGET :: a, b
REAL, POINTER :: pt
...
pt => a
b = pt            ! pt è deriferito: b è posto uguale ad a
IF (pt<0) pt = 0  ! in questa istruzione pt è deriferito due volte
...
WRITE(*,*) pt     ! viene stampato il valore del target di pt
READ(*,*) pt      ! viene letto il valore del target di pt
```

Ricapitolando, i puntatori prevedono *due* differenti forme di assegnazione che, avendo due significati *completamente diversi*, *non* devono assolutamente essere confusi:

- *assegnazione di puntatore* (`=>`): è una forma di *aliasing* (ossia un modo di riferirsi ad uno stesso oggetto con nomi differenti), a mezzo della quale il puntatore ed il suo target sono riferiti ad una stessa locazione di memoria. Questo tipo di assegnazione può aver luogo tra una variabile puntatore ed una variabile target, oppure tra due variabili puntatore.

- *comune assegnazione* (`=`): quando a sinistra dell'assegnazione c'è l'*alias* di un dato oggetto, l'operazione di assegnazione deve essere pensata applicata proprio all'oggetto puntato (ossia il valore specificato viene assegnato allo spazio puntato). Questo tipo di assegnazione può aver luogo tra una variabile puntatore ed un oggetto avente un appropriato tipo e rango; contrariamente a quanto avviene con l'assegnazione di puntatore, l'oggetto

il cui nome è sul lato *destro* dell'assegnazione *non* deve necessariamente avere l'attributo `TARGET`. Se una particolare locazione di memoria ospita una variabile che sia puntata da un certo numero di puntatori, allora il cambiamento del valore immagazzinato in questa locazione chiaramente andrà a modificare i valori deriferiti di *tutti* i puntatori associati a quella variabile.

Il seguente frammento di codice implementa, ad esempio, tre istruzioni di assegnazione di puntatore ed una sola istruzione di assegnazione:

```
REAL,TARGET,DIMENSION(3,3)  :: b
REAL,TARGET,DIMENSION(3)    :: r, s
REAL,POINTER,DIMENSION(:,:) :: a
REAL,POINTER,DIMENSION(:)    :: x, y
y => s          ! Associazione di puntatore: nessuna assegnazione
WRITE(*,*) s    ! Primo WRITE: Stampa il valore originale di s
a => b          ! Associazione di puntatore: nessuna assegnazione
x => r          ! Associazione di puntatore: nessuna assegnazione
y = MATMUL(a,x) ! Equivale all'istruzione di assegnazione s=MATMUL(a,x)
WRITE(*,*) s    ! Secondo WRITE: Adesso s è cambiato e si avra' un
                ! output differente rispetto a prima
```

7.3 Stato di associazione di un puntatore

Lo *stato di associazione* di un puntatore indica se il puntatore è *attualmente* associato o meno ad un target valido. I tre possibili stati di associazione di un puntatore sono: *indefinito*, *associato* e *deassociato*. Subito dopo la sua dichiarazione in una istruzione di dichiarazione di tipo, lo stato di associazione del puntatore è *indefinito*. Una volta che il puntatore sia stato associato ad un target a mezzo di una istruzione di assegnazione, il suo stato di associazione diventa *associato*. Se, successivamente, il puntatore viene deassociato dal suo target e *non* associato ad un nuovo target, il suo stato di associazione diventa *deassociato*.

Un puntatore può essere deassociato dal suo target o a seguito di una nuova istruzione di assegnazione (a mezzo della quale esso risulta associato ad un nuovo target) oppure attraverso l'esecuzione dell'istruzione `NULLIFY`. Quest'ultima ha la seguente forma:

`NULLIFY (`*`puntatore1`* `[, `*`puntatore2, ...`*`])`

A seguito di tale istruzione, tutti i puntatori presenti nella lista degli argomenti risulteranno deassociati dai loro rispettivi target.

Lo stato di associazione di un puntatore può essere conosciuto a mezzo della funzione intrinseca `ASSOCIATED` la cui espressione generale è la seguente:

`ASSOCIATED(`*`puntatore`* `[, `*`target`*`])`

Il valore ritornato da questa funzione può essere `.TRUE.` o `.FALSE.` Quando il riferimento opzionale a `target` è assente, `ASSOCIATED` ritorna il valore `.TRUE.` se il puntatore è associato

ad un qualsiasi target, .FALSE. se il puntatore è stato annullato con l'istruzione NULLIFY. Quando è presente anche il riferimento a target, la funzione ASSOCIATED restituisce il valore .TRUE. se il puntatore è associato al target in questione, il valore .FALSE. se il puntatore non è associato a quel target o se è stato annullato con l'istruzione NULLIFY. Si badi che testare lo stato di un puntatore *indefinito* comporta un *errore* in fase di esecuzione. E', pertanto, buona norma "annullare" tutti i puntatori che non siano immediatamente associati dopo la loro dichiarazione. A tale scopo il Fortran 95 ha introdotto la funzione intrinseca NULL che può essere usata per annullare un puntatore mentre lo si sta dichiarando, come si può vedere nel seguente esempio:

```
REAL, POINTER :: pr1=NULL(), pr2=NULL()
INTEGER, POINTER :: pi=NULL()
```

La funzione intrinseca NULL può tornare particolarmente utile in fase di definizione di un tipo di dati derivato, allo scopo di inizializzarne una componente definita con attributo POINTER:

```
TYPE :: entry
   INTEGER :: n=10
   REAL :: x=3.14
   TYPE(entry), POINTER :: next=>NULL()
END TYPE entry
TYPE(entry),DIMENSION(100) :: matrix
```

Si ricorda, a scanso di equivoci, che l'inizializzazione "contestuale" delle componenti di un tipo di dati derivato è, così come la procedura intrinseca NULL, prerogativa del Fortran 95 per cui l'esempio precedente non può essere compilato con un compilatore Fortran 90. Naturalmente l'inizializzazione può aver luogo anche direttamente in fase dichiarativa, per cui l'esempio precedente può essere riscritto al modo seguente:

```
TYPE :: entry
   INTEGER :: n
   REAL :: x
   TYPE(entry), POINTER :: next
END TYPE entry
TYPE(entry),DIMENSION(100) :: matrix=entry(10,3.14,NULL())
```

Il seguente frammento di programma mostra, infine, alcuni esempi di utilizzo della funzione intrinseca ASSOCIATED e del comando NULLIFY:

```
REAL, POINTER :: pt1, pt2          ! stato indefinito
REAL, TARGET  :: t1, t2
LOGICAL :: test
pt1 => t1                          ! pt1 associato
pt2 => t2                          ! pt2 associato
test = ASSOCIATED(pt1)             ! .T. (pt1 è associato a t1)
test = ASSOCIATED(pt2)             ! .T. (pt2 è associato a t2)
```

```
...
NULLIFY(pt1)                          ! pt1 deassociato
test = ASSOCIATED(pt1)                ! .F. (pt1 è stato deassociato)
test = ASSOCIATED(pt1,pt2)            ! .F. (pt1 non è associato a pt2)
test = ASSOCIATED(pt2,TARGET=t2)      ! .T. (pt2 è associato a t2)
test = ASSOCIATED(pt2,TARGET=t1)      ! .F. (pt2 non è associato a t1)
NULLIFY(pt1,pt2)                      ! pt1 e pt2 deassociati
```

7.4 Allocazione dinamica della memoria con i puntatori

Oltre che ad una variabile dichiarata con attributo `TARGET` o `POINTER`, un puntatore può anche essere associato a *blocchi di memoria dinamica*. Questa memoria viene allocata attraverso l'istruzione `ALLOCATE` la quale crea una variabile o un array (di dimensioni specificate) senza nome e avente il *tipo* ed il *rango* del puntatore. Dualmente, l'istruzione `DEALLOCATE` ha lo scopo di rimuovere la precedente allocazione.

La forma generale delle istruzioni `ALLOCATE` e `DEALLOCATE` è identica a quelle dell corrispondenti istruzioni valide per gli *allocatable array*:

```
ALLOCATE(puntatore(dim) [, STAT=stato])
DEALLOCATE(puntatore [, STAT=stato])
```

in cui *puntatore* è il nome di un puntatore alla variabile o all'array che si sta creando, *dim* è la specificazione delle dimensioni nel caso in cui l'oggetto che si sta creando sia un array, e *stato* è una variabile intera avente valore zero in caso di operazione terminata con successo, positivo (e dipendente dal processore) in caso contrario. La clausola `STAT=` è opzionale ma il suo utilizzo è sempre da preferirsi al fine di evitare un eventuale arresto del programma in caso di errore. Un esempio servirà a chiarire le definizioni precedenti:

```
REAL, POINTER :: pv, pa(:)
INTEGER :: stato, n=100
...
ALLOCATE(pv,pa(n),STAT=stato)
IF (stato==0) THEN
   ...
END IF
...
DEALLOCATE(pv,pa,STAT=stato)
IF (stato==0) THEN
   ...
END IF
```

Nell'esempio in esame, la variabile `pv` punta ad un'area di memoria dinamica che può contenere un *singolo* valore reale, mentre la variabile `pa` punta ad un blocco di memoria dinamica di dimensioni tali da poter contenere `100` valori reali. Quando la suddetta area di memoria non

serve più, essa viene liberata mediante l'istruzione `DEALLOCATE`. Si noti come questo uso dei puntatori sia molto simile a quello degli array *allocabili*.

L'allocazione dei puntatori costituisce uno strumento di gestione della memoria molto flessibile e potente. Tuttavia è necessario prestare la giusta attenzione per non incorrere in errori spesso abbastanza subdoli. Se ne citeranno i due più comuni:

- Deassociare il puntatore relativo ad un'area di memoria allocata senza aver prima deallocato la stessa (ad esempio annullando il puntatore o associandolo ad un altro *target*) significa rimuovere l'unico modo di riferirsi a quell'area di memoria la quale, da quel momento in poi, resterà inaccessibile e, in ogni caso, impossibile da liberare. Questo problema (detto di *memory leak*) è esemplificato brevemente dal seguente frammento di programma:

```
INTEGER, POINTER :: punt(:)
...
ALLOCATE(punt(100))
NULLIFY(punt)      ! attenzione: punt non era deallocato
```

- La rimozione di un *target* a cui è associato un puntatore rende quest'ultimo "penzolante". Questo può accadere tutte le volte in cui si dealloca la variabile target o si esce da una procedura a cui quest'ultima è locale. Un esempio è fornito dalle seguenti righe di programma:

```
REAL, POINTER :: punt1, punt2
...
ALLOCATE(pt1)
pt2 => pt1
DEALLOCATE(pt1)    ! attenzione: pt2 era ancora associato a pt1
```

  In questo caso, dopo la deallocazione di `pt1`, ogni eventuale utilizzo di `pt2` fornirà risultati assolutamente imprevedibili. E', pertanto, buona norma annullare (mediante l'istruzione `NULLIFY`) o riassegnare *tutti* i puntatori a una nuova locazione di memoria appena la vecchia memoria venga deallocata.

Di seguito si riporta un interessante esempio di impiego di puntatori atti a creare delle strutture dati di tipo matriciale per certi versi anche più versatili degli stessi array bidimensionali. Nel seguente modulo vengono definiti sia il tipo di dati `matrix` che alcune operazioni standard come la somma ed il prodotto interni nonché il prodotto di un oggetto di tipo `matrix` per uno scalare.

```
MODULE matrici
   TYPE matrix
      INTEGER :: row
      INTEGER :: col
      REAL, DIMENSION(:,:), POINTER :: array
```

```
   END TYPE matrix

   INTERFACE OPERATOR (+)
      MODULE PROCEDURE sum
   END INTERFACE

   INTERFACE OPERATOR (*)
      MODULE PROCEDURE prod, scale
   END INTERFACE

   PRIVATE sum, prod, scale

CONTAINS

   FUNCTION sum(A,B)
      TYPE(matrix), INTENT(IN) :: A, B
      TYPE(matrix) :: sum
         IF (A%row /= B%row .OR. A%col /= B%col) THEN
            WRITE(*,*) " Impossibile effettuare la somma "
         ELSE
            sum%row = A%row
            sum%col = A%col
            ALLOCATE(sum%array(A%row, A%col))
            sum%array = A%array + B%array
         END IF
   END FUNCTION sum

   FUNCTION prod(A,B)
      TYPE(matrix), INTENT(IN) :: A, B
      TYPE(matrix) :: prod
         IF (A%col /= B%row ) THEN
            WRITE(*,*) " Impossibile effettuare il prodotto "
         ELSE
            prod%row = A%row
            prod%col = B%col
            ALLOCATE(prod%array(A%row,B%col))
            prod%array = MATMUL(A%array, B%array)
         END IF
   END FUNCTION prod

   FUNCTION scale(c,A)
      REAL, INTENT(IN) :: c
      TYPE(matrix), INTENT(IN) :: A
```

```
      TYPE(matrix) :: scale
         scale%row = A%row
         scale%col = A%col
         ALLOCATE(scale%array(A%row,A%col))
         scale%array = c*A%array
   END FUNCTION scale

   SUBROUTINE setm(row,column,array,A)
      TYPE(matrix), INTENT(OUT) :: A
      INTEGER, INTENT(IN) :: row, column
      REAL, DIMENSION(:,:), INTENT(IN) :: array
      A%row = row
      A%col = column
      ALLOCATE(A%array(A%row,A%col))
      A%array = array
   END SUBROUTINE setm

   SUBROUTINE remove(A)
      TYPE(matrix), INTENT(INOUT) :: A
      DEALLOCATE(A%array)
   END SUBROUTINE remove

   SUBROUTINE printm(A)
      TYPE(matrix), INTENT(IN) :: A
      INTEGER :: i, j
      DO i = 1,A%row
         PRINT*, (A%array(i,j), j=1,A%col)
      END DO
   END SUBROUTINE printm

END MODULE matrici
```

Naturalmente è possibile estendere questo modulo al fine di prevedere altre operazioni come il calcolo della trasposta, dell'inversa oppure del determinante. A tal fine si può fare uso, con lievi modifiche, di alcune procedure già riportate nei capitoli precedenti. Un programma utile a testare il modulo precedente potrebbe essere il seguente:

```
PROGRAM test
  USE matrici
  IMPLICIT NONE
  TYPE(matrix) :: A, B, C
  REAL, DIMENSION(2,4) :: tmp
  tmp(1,:) = (/1.0, 2.0, 3.0, 4.0 /)
  tmp(2,:) = (/5.0, 6.0, 7.0, 8.0 /)
```

```
  CALL setm(2,2,tmp(1:2,1:2), A)
  CALL setm(2,4,tmp, B)
  PRINT*, 'matrice A'
  CALL printm(A)
  PRINT*, 'matrice B'
  CALL printm(B)
  C = A*B + 2.0 * B
  PRINT*, "matrice AB + 2B"
  CALL printm(C)
  CALL remove(C)
END PROGRAM test
```

Si noti che mediante l'istruzione `DEALLOCATE` si può deallocare soltanto la memoria creata con una istruzione `ALLOCATE`: un qualsiasi tentativo, infatti, di applicare l'istruzione `DEALLOCATE` ad un puntatore associato ad un target *non* creato con una istruzione `ALLOCATE` genererà un errore in fase di esecuzione che interromperà il programma, a meno che non sia stata usata la clausola `STAT=`.

Data la perfetta analogia formale tra le istruzioni di allocazione e deallocazione di un *puntatore* e quelle valide per un *allocatable array* è chiaro che, laddove risulti appropriato, puntatori ed array allocabili possono essere allocati e deallocati nella medesima istruzione.

A questo punto appare doveroso mettere a confronto puntatori ed array allocabili relativamente alla loro funzione di allocare memoria in modo dinamico, allo scopo di comprendere quando convenga utilizzare gli uni e quando gli altri. Ebbene, del tutto in generale si può asserire che la scelta dovrebbe cadere sugli array allocabili quando a tutto il resto si vogliano anteporre le caratteristiche di semplicità di impiego e di efficienza del codice generato; sui puntatori quando, al contrario, si vogliano preferire la versatilità dell'utilizzo e la ricchezza della gamma di impiego. Quanto asserito appare ben supportato dalle seguenti considerazioni:

- L'impiego di array allocabili è limitato dal fatto che essi non possono essere utilizzati come componenti di strutture. Quindi una espressione del tipo:

  ```
  TYPE stack
     INTEGER :: index
     INTEGER, ALLOCATABLE, DIMENSION(:) :: content   ! Errore!!!
  END TYPE stack
  ```

  è completamente sbagliata. Nel caso in cu fosse necessario realizzare *strutture dati dinamiche* si dovrà ricorrere a puntatori:

  ```
  TYPE stack
     INTEGER :: index
     INTEGER, POINTER, DIMENSION(:) :: content
  END TYPE stack
  ```

  Su questo argomento, comunque, si tornerà alla fine del capitolo.

- Gli array allocabili possono essere utilizzati come *parametri attuali* in chiamate di procedura soltanto nella loro *forma allocata* (inoltre i corrispondenti parametri formali dovranno essere array fittizi di forma presunta non potendo essere, essi stessi, array allocabili). Nel caso in cui fosse necessario condividere con un sottoprogramma un area di memoria non ancora allocata all'atto della chiamata di procedura, sarà necessario fare uso di puntatori. Un'altra possibilità per superare il limite di cui sopra è quella di definire gli array allocabili in un modulo, come questo:

```
MODULE miomodulo
! Scopo: Condividere array allocabili fra piu'
!        unita' di programma
   SAVE
   REAL, ALLOCATABLE, DIMENSION(:,:)   :: A
   INTEGER, ALLOCATABLE, DIMENSION(:) :: b, c
END MODULE miomodulo
```

  e condividerne l'uso fra l'unità chiamante e la procedura invocata.

- Le procedure di ottimizzazione operate da un compilatore ottimizzante sono di gran lunga più complicate su un codice che faccia largo uso di puntatori piuttosto che di array allocabili e ciò a causa del fatto che l'utilizzo di puntatori comporta tipicamente l'impiego di diversi *alias* per riferirsi ad una medesima area di memoria. Al contrario, una variabile array e l'area di memoria a cui si riferisce è sempre determinabile con estrema semplicità e in maniera assolutamente non ambigua.

7.5 Operazioni di I/O con puntatori

Quando un puntatore appare in un'istruzione di I/O esso subisce un immediato *dereferencing* sicché ciò che viene effettivamente letto o prodotto in stampa è il valore del suo target. Chiaramente, affinché un'operazione di I/O che coinvolga un puntatore sia valida è necessario che il puntatore in oggetto si trovi nello stato *associato*, poiché in caso contrario il *dereferencing* non potrebbe avere luogo. Allo scopo di chiarire questo semplice concetto si può fare riferimento al programma seguente:

```
PROGRAM IO_con_puntatori
  IMPLICIT NONE
  INTEGER :: ierr
  REAL, DIMENSION(3), TARGET :: arr=(/1.,2.,3./)
  REAL, DIMENSION(:), POINTER :: p, q
  p => arr
  PRINT*, p         ! viene stampato arr
  ALLOCATE(q(5),STAT=ierr)
  IF(ierr == 0) THEN
     READ(*,*) q    ! Vengono dapprima letti e poi...
```

```
      PRINT*, q      ! ...stampati i dati della memoria dinamica
      DEALLOCATE(q)
      PRINT*, q      ! Operazione non valida!!!
  END IF
END PROGRAM IO_con_puntatori
```

Come è evidente, la prima istruzione di scrittura impone il *dereferencing* del puntatore `p` per cui produce la stampa del vettore `arr`. Le successive operazioni di lettura e scrittura di `q` funzionano allo stesso modo per cui il programma dapprima attende la lettura di cinque valori reali da introdurre nelle locazioni della memoria dinamica precedentemente allocata e successivamente ne produce la stampa. L'ultima istruzione di output, relativa al puntatore `q`, collocata subito dopo la deallocazione della memoria a cui esso puntava, *non* è valida per cui produce un messaggio di errore in fase di esecuzione ed il conseguente arresto del programma.

7.6 Un uso efficiente dei puntatori

Il seguente esempio mostra come un uso particolarmente semplice dei puntatori possa aumentare l'efficienza di un programma. Si supponga di dover applicare una procedura di *swap* a due array reali bidimensionali di `100`×`100` elementi. La procedura tradizionale che normalmente si applicherebbe in questo caso è la seguente:

```
REAL, DIMENSION(100,100) :: array1, array2, app
...
app = array1
array1 = array2
array2 = app
```

Questo frammento di codice è molto semplice ma scarsamente efficiente visto che comporta lo *spostamento fisico* di `10000` valori reali in ognuna delle tre precedenti istruzioni di assegnazione. La stessa operazione può essere condotta operando con i puntatori ai due oggetti da invertire, scambiando in tal modo soltanto gli *indirizzi* dei due array target e *non* gli interi array:

```
REAL, DIMENSION(100,100), TARGET :: array1, array2
REAL, DIMENSION(:,:), POINTER :: p1, p2, app
...
p1 => array1
p2 => array2
...
app => p1
p1 => p2
p2 => app
```

L'unico prezzo da pagare è il fatto che ogni riferimento agli array dovrà avvenire a mezzo degli identificatori dei loro puntatori (in altre parole, ogni volta che bisognerà riferirsi, ad esempio,

ad `array1` si dovrà scrivere `p1` prima che lo scambio abbia avuto luogo, `p2` successivamente allo scambio). In realtà si poteva ovviare alla introduzione della variabile *extra* `app`, la quale è stata usata semplicemente per motivi di simmetria rispetto al procedimento standard.

Un'applicazione pratica dei concetti testé esposti è rappresentata dal problema dell'ordinamento alfabetico di una lista di nomi in una lista. Nel programma che segue e che implementa proprio un algoritmo di ordinamento, la lettura delle stringhe di caratteri avviene in maniera interattiva (ad esempio da tastiera) ma unicamente per ragioni didattiche, ciò limitando di molto la mole di dati che è possibile inserire per testare il codice. E' chiaro che in casi pratici la lista di nomi andrà letta da file e potrà essere anche decisamente lunga. Anche in questo esempio l'uso dei puntatori evita lo spostamento fisico di grossi oggetti (stringhe di 500 caratteri di lunghezza) compensando così la scarsa efficienza dell'algoritmo di ordinamento utilizzato (il metodo *bubble sort*).

```
PROGRAM sort
  IMPLICIT NONE
  INTEGER, PARAMETER :: max_length=500, max_no = 100
  TYPE :: char_pt
     CHARACTER(LEN=max_length), pointer :: ptr
  END TYPE char_pt
  TYPE(char_pt), dimension(max_no) :: strings
  TYPE(char_pt) :: swap
  INTEGER :: i, j, n
! Invece di usare un array di stringhe di caratteri, viene impiegato
! un array di puntatori ad oggetti di tipo stringa di caratteri
  WRITE(*,"(A,I3,A3)",ADVANCE="NO") "Inserisci il numero di &
                                    &stringhe, (massimo ",max_no,"): "
  READ(*,*) n
  DO i=1,n
     ALLOCATE(strings(i)%ptr)
     WRITE(*,"(A,I3,1X,A4)",ADVANCE="NO") "Stringa n. ", i,"--> "
     READ(*,*) strings(i)%ptr
  END DO
  DO i=1,n-1
     DO j=1,n-i
! Meccanismo bubble-sort
        IF(strings(j)%ptr > strings(j+1)%ptr) THEN
           swap%ptr => strings(j)%ptr
           strings(j)%ptr => strings(j+1)%ptr
           strings(j+1)%ptr => swap%ptr
        END IF
     END DO
  END DO
  PRINT "(/,A,/,15('='))", "Lista ordinata:"
  PRINT "(A)", (trim(strings(i)%ptr), i = 1,n)
```

```
END PROGRAM sort
```

Si riporta per completezza un esempio di utilizzo di questo programma:

```
Inserisci il numero di stringhe, (massimo 100): 4
Stringa n.   1 --> Michele
Stringa n.   2 --> Giovannina
Stringa n.   3 --> Carolina
Stringa n.   4 --> Antonio

Lista ordinata:
===============
Antonio
Carolina
Giovannina
Michele
```

Un utilizzo parimenti efficiente dei puntatori potrebbe essere quello di definire un array di puntatori e di assegnarne ciascun componente ad un record di un database. In tal modo, le operazioni di ordinamento del file secondo una *chiave* non richiederà più lo spostamento di (eventualmente) grossi record ma soltanto una diversa associazione di questi ultimi ai rispettivi target.

Si noti, tuttavia, che se da un lato i puntatori possono aumentare l'efficienza di alcune procedure di calcolo, è anche vero che un loro uso eccessivo può rendere meno leggibile (e gestibile) il codice sorgente aumentando, di conseguenza, la probabilità di commettere errori. Inoltre non è da sottovalutare un altro aspetto connesso all'impiego di puntatori, che è la già citata impossibilità da parte dei compilatori di produrre codice ottimizzato qualora nel programma sorgente si sia fatto uso di puntatori.

7.7 Puntatori a sezioni di array

Come visto in precedenza, un puntatore può puntare ad un *intero array* nello stesso modo con cui può puntare ad uno scalare. Le seguenti istruzioni rappresentano un valido esempio di questa caratteristica:

```
REAL, DIMENSION(100,100), TARGET :: targ
REAL, DIMENSION(:,:), POINTER :: punt
punt => targ
```

Ma un puntatore può essere riferito anche ad una *sezione di array*, come mostrato dal seguente programma:

```
PROGRAM point_sez_array
  IMPLICIT NONE
  INTEGER :: i
```

```
  INTEGER, DIMENSION(16), TARGET :: vet=(/(i, i=1,16)/)
  INTEGER, DIMENSION(16), POINTER :: ptr1, ptr2, ptr3, ptr4, ptr5
  ptr1 => vet
  ptr2 => ptr1(2::2)
  ptr3 => ptr2(2::2)
  ptr4 => ptr3(2::2)
  ptr5 => ptr4(2::2)
  WRITE(*,'(A,16I3)') " ptr1 = ", ptr1
  WRITE(*,'(A,16I3)') " ptr2 = ", ptr2
  WRITE(*,'(A,16I3)') " ptr3 = ", ptr3
  WRITE(*,'(A,16I3)') " ptr4 = ", ptr4
  WRITE(*,'(A,16I3)') " ptr5 = ", ptr5
END PROGRAM point_sez_array
```

Il risultato fornito da questo programma è, chiaramente, il seguente:

```
ptr1 =  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
ptr2 =  2   4   6   8  10  12  14  16
ptr3 =  4   8  12  16
ptr4 =  8  16
ptr5 = 16
```

Si noti, però, che sebbene i puntatori lavorino con sezioni di array definite da *triplette di indici*, essi *non* funzionano con sezioni di array definiti da *indici vettoriali*. Ad esempio, il programma seguente fornirà un errore in fase di compilazione:

```
PROGRAM bad_sez_array
  IMPLICIT NONE
  INTEGER :: i
  INTEGER, DIMENSION(3) :: indvet=(1,3,11)
  INTEGER, DIMENSION(16), TARGET :: vet=(/(i, i=1,16)/)
  INTEGER, DIMENSION(:), POINTER :: punt
  punt => vet(indvet)    ! errore: sezione di array con indice vettoriale
  WRITE(*,'(1X,A,16I3)') 'punt = ', punt
END PROGRAM bad_sez_array
```

Come ulteriore esempio di puntatori a sezioni di array si potrebbe pensare di riscrivere il programma di risoluzione dell'equazione di Laplace già incontrato al capitolo 3. Ancora una volta si perviene ad un codice eccezionalmente compatto ed efficiente, magari appena un pò meno "immediato" da interpretare:

```
PROGRAM laplace_con_punt
! *** Sezione dichiarativa ***
  IMPLICIT NONE
  REAL,DIMENSION(10,10),TARGET    :: Told
```

```
  REAL,DIMENSION(10,10)           :: T
    REAL, POINTER, DIMENSION(:,:) :: n, e, s, w, inside
  REAL    :: diff
  INTEGER :: i, j,niter
! *** Sezione esecutiva ***
! Valori iniziali di tentativo
  T = 0
! Condizioni al contorno
  T(1:10,1) = 1.0
  T(1,1:10) = (/(0.1*j,j=10,1,-1)/)
! Inizializzazione
  Told = T
! Puntatori alle diverse porzioni di Told
  inside => Told(2:9,2:9)
  n => Told(1:8,2:9)
  s => Told(3:10,2:9)
  e => Told(2:9,1:8)
  w => Told(2:9,3:10)
! Inizio iterazioni
  niter = 0
  DO
     T(2:9,2:9) = (n + e + s + w)/4.0
     diff = MAXVAL(ABS(T(2:9,2:9)-inside))
     niter = niter + 1
! Aggiornamento dei risultati
     inside = T(2:9,2:9)
     PRINT*, "Iter n. ", niter, diff
     IF(diff < 1.0E-4) THEN
        EXIT
     END IF
  END DO
! Stampa dei risultati
  PRINT*, "Campo finale: "
  DO i = 1,10
     PRINT "(10F7.3)", T(1:10,i)
  END DO
END PROGRAM laplace_con_punt
```

7.8 Puntatori come parametri formali di procedure

I puntatori, siano essi allocati o non, possono essere usati come *parametri formali* in procedure e ad esse passati come *parametri attuali* da parte dell'unità di programma chiamante, purché

siano rispettate le seguenti condizioni:

- Se una procedura ha un parametro formale con attributo `POINTER` oppure `TARGET`, l'interfaccia deve necessariamente essere *esplicita* (il che richiede, ad esempio, l'uso di un *interface block* per le chiamate a procedure *esterne*).
- Se un parametro formale è un puntatore, anche il parametro attuale corrispondente deve essere un puntatore con lo stesso tipo e rango.
- Un puntatore che sia parametro formale non può avere l'attributo `INTENT` né può comparire in una procedura `ELEMENTAL` del Fortran 95.

Quando entrambi i parametri (formale ed attuale) hanno l'attributo `POINTER`, attraverso l'istruzione di chiamata ed il ritorno alla unità chiamante vengono passati anche il *target* (se ne esiste uno) e lo *stato di associazione*. Pertanto è necessario assicurarsi che il target rimanga valido quando si ritorni all'unità chiamante (ossia è necessario che il target non sia locale alla procedura invocata); in caso contrario il puntatore resterebbe "penzolante".

Quando l'argomento attuale è un puntatore ma il corrispondente argomento formale non lo è, il puntatore subisce il *dereferencing* ed è il suo *target* che viene assegnato al parametro formale: all'uscita dalla procedura chiamata il target del puntatore passato conserva il valore del parametro formale. Questo significa che è necessario, all'atto della chiamata di procedura, che detto puntatore sia *associato* ad un target.

L'uso di puntatori come parametri di scambio fra procedure rende un programma Fortran molto potente e flessibile ma aumenta, al contempo, le probabilità di commettere errori: in situazioni in cui, ad esempio, i puntatori sono allocati in una procedura, usati in un'altra e, infine, deallocati e annullati in un'altra ancora, è errore assai comune tentare di lavorare con puntatori deassociati o allocare nuovi array con puntatori già in uso. E', quindi, buona norma, prima di lavorare con un puntatore, interrogarne lo stato di associazione a mezzo della funzione `ASSOCIATED`.

A chiarimento di quanto detto, si consideri l'esempio seguente:

```
PROGRAM esempio_sub
  INTERFACE
     SUBROUTINE sub(p)
        REAL, DIMENSION(:,:), POINTER :: p
     END SUBROUTINE sub
  END INTERFACE

  REAL, DIMENSION(:,:), POINTER :: punt
  ...
  ALLOCATE(punt(100,100))
  CALL sub(punt)
  ...
END PROGRAM esempio_sub
```

```
SUBROUTINE sub(p)
   REAL, DIMENSION(:,:), POINTER :: p
      ...
      DEALLOCATE(p)
      ...
END SUBROUTINE sub
```

In questo esempio, il puntatore `punt` è l'argomento *dummy* di una procedura esterna per cui l'introduzione di un *interface block* risulta necessaria allo scopo di fornire una interfaccia esplicita nell'unità di programma chiamante. Un'alternativa, ovviamente, poteva essere quella di inserire la procedura `sub` in un *modulo* oppure introdurla come *procedura interna* in modo da fornire una interfaccia esplicita *di default*. Si osservi, in sintesi, cosa fa il frammento di codice precedente. Il programma chiamante alloca dinamicamente una regione di memoria atta ad ospitare `100`×`100` valori reali e ne indica il puntatore `punt` come un *alias*; quindi invoca la procedura esterna `sub`. Questa chiamata di procedura associa il parametro formale `p` al parametro attuale `punt`; quando `sub` dealloca `p` anche l'argomento attuale `punt` viene deallocato (e con esso l'area di memoria a cui è associato).

Questo modo di operare con i puntatori è più flessibile rispetto all'uso degli *allocatable array*. Questi ultimi, infatti, possono essere allocati e deallocati soltanto nella stessa unità di programma, ed inoltre possono essere usati come parametri formali in chiamate di procedura soltanto nella loro forma *allocata*. Pertanto, se l'allocazione deve avvenire all'interno della procedura chiamata si dovrà necessariamente far ricorso ai puntatori, come esemplificato nel seguente programma:

```
PROGRAM pdemo
  IMPLICIT NONE
  REAL, POINTER :: parray(:)
  OPEN(UNIT=9, FILE='mydata', STATUS='old')
  CALL readin(9,parray)
  WRITE(*,*)'array of ', SIZE(array), ' points:'
  WRITE(*,*) parray
  DEALLOCATE(parray)
  STOP
CONTAINS
   SUBROUTINE readin(iounit,z)
      INTEGER, INTENT(IN) :: iounit
       REAL, POINTER      :: z(:)   ! NB: e' vietato l'uso di INTENT
       INTEGER :: npoints
          READ(iounit) npoints      ! legge le dimensioni
          ALLOCATE(z(1:npoints))    ! alloca lo spazio
          READ(iounit) z            ! legge l'intero array
   END SUBROUTINE readin
 END PROGRAM pdemo
```

Il seguente programma mostra un ulteriore esempio di passaggio di argomenti con attributo POINTER, anche questa volta ad una procedura interna. Lo stesso programma fornisce, nel contempo, un nuovo esempio di uso efficiente di puntatori nelle operazioni di *swap*, molto frequenti negli algoritmi iterativi. Il programma calcola la radice quadrata degli elementi di un vettore y di 1000 elementi utilizzando la nota formula iterativa:

$$x_{n+1} = \frac{x_n + y/x_n}{2}$$

ed utilizza efficacemente i puntatori allo scopo di evitare onerose operazioni di copia. La convergenza del metodo viene testata esaminando la differenza fra due approssimazioni consecutive con riferimento al massimo discostamento fra i due passi.

```
PROGRAM prova_func_punt
  IMPLICIT NONE
  REAL, PARAMETER :: eps=0.001
  INTEGER :: i, j
  INTEGER, PARAMETER :: dim=1000
  REAL, DIMENSION(dim) :: y=(/(i, i=1,dim)/)
  REAL, DIMENSION(dim), target :: guess=1., xold=1., xnew
  REAL, DIMENSION(:), POINTER :: ptnew, ptold, swap
  LOGICAL :: conv=.FALSE.
  ptnew => xnew
  ptold => xold
  ptold => guess
  j = 0
  DO WHILE(.NOT.conv)
     j = j+1
     ptnew = calc(ptold,y)
     swap => ptold
     ptold => ptnew
     ptnew => swap
     conv = MAXVAL(ABS(ptnew-ptold))<=eps
  END DO
  PRINT*, "Convergenza raggiunta dopo ",j," iterazioni"
  ...      ! Fase di output
  CONTAINS
     FUNCTION calc(ptold,y)
        REAL, DIMENSION(:), POINTER :: ptold
        REAL, DIMENSION(size(ptold)) :: calc
        REAL, DIMENSION(:) :: y
           calc = (ptold+y/ptold)/2.
     END FUNCTION calc
END PROGRAM prova_func_punt
```

7.9 Funzioni Puntatore

Anche il risultato di una funzione può avere l'attributo `POINTER`, la qual cosa può tornare molto utile, ad esempio, in quei casi in cui le dimensioni del risultato non sono note a priori ma dipendono anch'stesse dai calcoli eseguiti all'interno della funzione stessa. In questi casi risulta necessario l'utilizzo della parola chiave `RESULT` nella definizione della funzione ed è inoltre la variabile risultato a dover essere definita con l'attributo `POINTER`. Si deve, infine, rispettare il consueto vincolo dell'*interfaccia esplicita* nell'unità chiamante. Il risultato della funzione puntatore può essere normalmente usato in una istruzione di assegnazione di puntatore purché associato ad un target valido. Di seguito si riporta un esempio di funzione puntatore ed un frammento di programma che la richiama.

```
MODULE mio_mod
   IMPLICIT NONE
CONTAINS
   FUNCTION even_pointer(a) RESULT(p)
      REAL, DIMENSION(:), POINTER :: a
      REAL, DIMENSION(:), POINTER :: p
         p => a(2::2)
   END FUNCTION even_pointer
END MODULE mio_mod

PROGRAM pointer_function
  USE mio_mod
  IMPLICIT NONE
  REAL, DIMENSION(15), TARGET :: a
  REAL, DIMENSION(:), POINTER :: pa, p, q, r
  ...
  pa => a
  p => even_pointer(pa)    ! p punta agli elementi pari di a
  q => even_pointer(p)     ! q punta agli elementi pari di p
  r => even_pointer(q)     ! r punta agli elementi pari di q
  ...
END PROGRAM pointer_function
```

In breve, il programma `pointer_function` usa la funzione `even_pointer` dapprima per far sì che `p` punti agli elementi pari di `pa`, ossia per associare `p` al vettore formato dagli elementi `a(2)`, `a(4)`, `a(6)`, `a(8)`, `a(10)`, `a(12)`, `a(14)`. Poi, il puntatore `q` viene associato agli elementi pari dell'array puntato da `p` (che pertanto viene *deriferito*) sicché `q` punterà all'array formato da `a(4)`, `a(8)` e `a(12)`. Infine, `r` viene puntato agli elementi pari del target di `q`, ossia viene associato al solo `a(8)`.

Come è noto, oltre alle procedure di modulo, anche le procedure interne realizzano automaticamente una interfaccia esplicita all'interno della procedura chiamante. Pertanto anche il seguente programma è perfettamente valido da un punto di vista sintattico.

```
PROGRAM zip
  IMPLICIT NONE
  INTEGER, DIMENSION(10) :: x=(/1, 0, 4, 2, 0, 3, 0, 4, 3, 0/)
  WRITE(*,*) x
  WRITE(*,*) compact(x)
CONTAINS
   FUNCTION compact(x)
! Procedura per rimuovere gli zeri dal vettore x
      IMPLICIT NONE
      INTEGER, POINTER :: compact(:)
      INTEGER x(:)
      INTEGER :: n
         n = COUNT(x/=0)          ! Conta i valori di x diversi da zero
         ALLOCATE(compact(n))     ! Alloca un vettore di n elementi
         WHERE(x/=0) compact = x  ! Copia i valori di x diversi da zero
                                  !  nel vettore compact
     END FUNCTION compact
END PROGRAM zip
```

Di seguito si riporta un ulteriore esempio, questa volta di *funzione esterna* puntatore:

```
PROGRAM main
  IMPLICIT NONE
  REAL :: x
  INTEGER, TARGET :: a, b
  INTEGER, POINTER :: largest
  INTERFACE
     FUNCTION ptr(a,b)
        IMPLICIT NONE
        INTEGER, TARGET, INTENT(IN) :: a, b
        INTEGER, POINTER :: ptr
     END FUNCTION ptr
  END INTERFACE
  CALL RANDOM_NUMBER(x)
  a = 1.0E+04*x
  CALL RANDOM_NUMBER(x)
  b = 1.0E+04*x
  largest => ptr(a,b)
  PRINT*, "L' elemento massimo fra ",a, " e ",b," e': ", largest
END PROGRAM main

FUNCTION ptr(a,b)
   IMPLICIT NONE
   INTEGER, TARGET, INTENT(IN) :: a, b
```

```
   INTEGER, POINTER :: ptr
      IF (a>b) THEN
         ptr => a
      ELSE
         ptr => b
      END IF
END FUNCTION ptr
```

7.10 Array di Puntatori

Dal momento che un puntatore è solo un *attributo* e non un *tipo di dati*, non è possibile dichiarare *direttamente* un *array di puntatori* sicché, ad esempio, la seguente dichiarazione *non* è valida e provoca un errore di compilazione:

```
REAL, DIMENSION(10), POINTER :: p
```

Si può, tuttavia, aggirare l'ostacolo definendo un *array di oggetti di tipo derivato* aventi come componenti soltanto puntatori: infatti, sebbene in Fortran non sia lecito definire un array di puntatori, è, invece, perfettamente lecito definire un tipo di dati derivato contenente solo un puntatore e creare, quindi, un array di tale tipo. Si può in tal modo coniugare l'efficienza connessa all'uso dei puntatori con le capacità di "sintesi" proprie degli array e degli oggetti di tipo derivato. Il programma seguente fornisce un esempio di quanto detto:

```
PROGRAM array_di_punt
  IMPLICIT NONE
  INTEGER :: i
  TYPE :: punt_vet
      REAL, DIMENSION(:), POINTER :: punt
  END TYPE punt_vet
  TYPE(punt_vet), DIMENSION(3) :: pt
  REAL, DIMENSION(5), TARGET :: a1=(/(i, i=1,5)/)
  REAL, DIMENSION(5), TARGET :: a2=(/(i, i=2,10,2)/)
  REAL, DIMENSION(5), TARGET :: a3=(/(i, i =-1,11,3)/)
  pt(1)%punt => a1
  pt(2)%punt => a2
  pt(3)%punt => a3
  WRITE(*,*) pt(3)%punt
END PROGRAM
```

Quando viene eseguito, il programma produce, come è ovvio, la stampa a video dell'array `a3`, sicché stamperà il record:

```
 -1   2   5   8   11
```

Un esempio leggermente diverso potrebbe essere questo:

```
TYPE :: ptr_to_array
   REAL, DIMENSION(:), POINTER :: arr
END TYPE ptr_to_array
TYPE(ptr_to_array), ALLOCATABLE(:) :: x
```

A questo punto è possibile riferirsi all'`i_`*mo* puntatore semplicemente scrivendo `x(i)%arr`. Il seguente frammento di programma mostra in che modo ciascuna riga di una matrice triangolare bassa possa essere rappresentata da un array dinamico di dimensione crescente:

```
INTEGER,PARAMETER :: n=10
TYPE(ptr_to_array), DIMENSION(n) :: a
INTEGER :: i
...
DO i = 1,n
   ALLOCATE(x(i)%arr(i))
END DO
```

Si noti che `x(i)%arr` punta ad un array di dimensione `i` allocato dinamicamente. Questa rappresentazione permette di utilizzare soltanto la metà dello spazio di memoria richiesto per un comune array bidimensionale `n`×`n`.

7.11 Strutture dati dinamiche

Come detto, un componente di un tipo di dati derivato può essere tranquillamente definito con l'attributo `POINTER`. E' stato altresì anticipato che tale componente puntatore può puntare non solo a dati di tipo predefinito o ad oggetti di tipo derivato già definiti, ma anche ad oggetti del tipo in corso di definizione. Ne è un esempio la seguente definizione di tipo:

```
TYPE node
   INTEGER :: valore            ! campo numerico
   TYPE(node), POINTER :: next  ! campo puntatore
END TYPE node
```

Questa proprietà consente di definire strutture dati dinamiche quali liste concatenate ed alberi binari, entrambe costituite da un insieme di oggetti (detti *nodi*) definiti in un modo perfettamente analogo a quanto appena visto per il tipo `node`. In altre parole una struttura dati dinamica può essere sempre vista come una collezione di oggetti legati insieme da uno o più associazioni successive di puntatore.

7.11.1 Liste concatenate

Una delle più comuni e potenti applicazioni dei puntatori è nella creazione e nella gestione di *liste concatenate*. Una lista concatenata è una collezione di oggetti di tipo derivato, ciascuno dei quali avente come componente un puntatore alla *successiva* variabile della lista. In una lista concatenata gli oggetti connessi (i *nodi*) si caratterizzano per il fatto che:

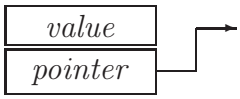


Figura 7.1: Generico nodo di lista concatenata

- Non sono immagazzinati necessariamente in maniera contigua.
- Possono essere creati dinamicamente (cioè al tempo di esecuzione).
- Possono essere inseriti in un punto qualsiasi della lista.
- Possono essere rimossi dinamicamente.

Per tale ragione una lista può crescere o ridursi in maniera arbitraria durante l'esecuzione di un programma.

Una lista concatenata consiste essenzialmente di strutture (ossia, con terminologia Fortran, di oggetti di tipo derivato) contenenti campi per i comuni dati e in più un ulteriore campo che rappresenta un puntatore al successivo elemento della lista. Per convenzione il primo e l'ultimo nodo della lista vengono chiamati rispettivamente *testa* e *coda* della lista.

Il programma che segue vuole essere un esempio introduttivo al problema della definizione di liste concatenate:

```
PROGRAM simple_linked_list
  IMPLICIT NONE
! Definizione di un nodo
  TYPE node
     INTEGER :: value
     TYPE (node), POINTER :: next
  END TYPE node
  INTEGER :: num
! Dichiarazione del nodo corrente e dell'intera lista
  TYPE (node), POINTER :: list, current
! Costruzione della lista
  NULLIFY(list) ! inizialmente la lista e' vuota
  DO
     READ(*,*) num     ! legge un valore dalla tastiera
     IF (num==0) EXIT  ! il ciclo si arresta quando viene inserito 0
     ALLOCATE(current) ! crea un nuovo nodo
     current%value = num  ! immagazzina il nuovo valore
     current%next => list ! punta al nodo precedente
     list => current      ! aggiorna la testa della lista
  END DO
! Attraversa la lista e ne stampa i valori
  current => list  ! current e' un alias della lista
```

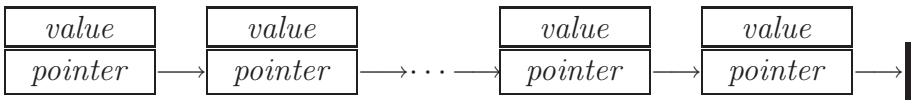


Figura 7.2: Lista concatenata

```
  DO
     IF (.NOT.ASSOCIATED(current)) EXIT ! esce quando arriva alla coda
     PRINT*, current%value   ! stampa il valore
     current => current%next ! current e' un alias del prossimo nodo
  END DO
  STOP
END PROGRAM simple_linked_list
```

Si osservi attentamente le operazioni effettuate dal programma precedente:

- Inizialmente viene definito un tipo `node` contenente un valore intero quale campo numerico ed un componente puntatore che punta ad un oggetto dello stesso tipo.
- Vengono dichiarate due variabili, `list` e `current`, di tipo `node`, la prima delle quali verrà usata per puntare alla testa della lista, la seconda al generico nodo della lista.

La procedura di costruzione della lista viene illustrata progressivamente come segue:

- All'inizio la lista è vuota sicché essa viene inizializzata "annullando" il suo alias `list`.
- Supponendo di acquisire da tastiera un valore intero la lista viene così creata e composta da un solo nodo atto ad ospitare il valore appena acquisito. Ciò ha luogo innanzitutto allocando dinamicamente della memoria per il nodo `current`, quindi immagazzinandovi il valore nel campo numerico preposto e, infine, facendo sì che la lista, attraverso il suo alias `list`, punti alla testa della lista che è ora rappresentata dal nodo appena allocato.
- Il procedimento descritto al punto precedente viene ripetuto identicamente fino a che non venga inserito da tastiera il valore `0`.

Una volta costruita la lista, il compito successivo sarà quello di scorrerne gli elementi e stamparne i valori. Ciò viene effettuato al modo seguente:

- Si rende l'oggetto `current` un alias di `list` facendo sì che esso punti al nodo di testa della lista.
- Si stampa il valore di quel nodo (ossia il campo numerico di `current%value`).
- Al termine della stampa l'oggetto `current` è fatto puntare al nodo successivo.
- Il procedimento testé descritto viene iterato fino a che non viene incontrato un nodo non associato (vale a dire il nodo di coda della lista).

Uno dei vantaggi che si ha lavorando con liste concatenate consiste nel fatto che la memoria allocata può immediatamente essere liberata quando non più necessaria. Questa operazione può essere facilmente effettuata attraversando la lista e deallocandone tutti i nodi, nello stesso modo con cui essi sono stati processati per essere sottoposti a stampa:

```
! Attraversamento della lista e deallocazione dei nodi
current => list  ! current punta alla testa della lista
DO
   IF (.NOT.ASSOCIATED(current)) EXIT ! esce quando raggiunge la coda
   list => current%next  ! list punta al successivo nodo di testa
   DEALLOCATE(current)   ! dealloca il corrente nodo di testa
   current => list       ! current punta al nuovo nodo di testa
END DO
```

Un pratico esempio di utilizzo di liste concatenate può essere concepito, ad esempio, per l'ordinamento di un elenco di valori secondo il meccanismo di *ordinamento per inserzione* (*insertion sort*) il quale, come è noto, viene utilizzato per la costruzione di una lista ordinata all'atto della generazione della lista stessa. L'algoritmo in esame lavora inserendo ciascun valore, appena questo viene letto, nella posizione che gli compete nella lista. In particolare, se tale valore è minore di tutti i valori precedentemente immagazzinati nella lista, esso verrà inserito in testa alla lista; al contrario, se è maggiore di tutti gli elementi della lista esso verrà posizionato in coda alla lista; infine, se questo valore si colloca in una posizione, per così dire, intermedia esso verrà inserito nella idonea posizione $a(j)$ facendo preliminarmente avanzare di una posizione tutti gli elementi della sottosequenza ordinata $a(j), a(j+1), \ldots a(n)$. Ebbene, si noti che se la lista in oggetto fosse implementata a mezzo di un array, l'inserimento di un qualsiasi elemento all'interno della lista comporterebbe uno spostamento "fisico" di intere sezioni di array. L'impiego, invece, di liste concatenate fornisce una valida soluzione per definire una relazione di ordinamento tra gli elementi evitando di procedere ad un riordinamento fisico degli stessi. Quanto detto è efficacemente descritto dal seguente programma:

```
PROGRAM inserzione
!
!  Scopo: Leggere una serie di valori reali da un file di dati
!         organizzarli secondo il procedimento di ordinamento
!         per inserzione. Una volta ordinati, gli elementi
!         della lista vengono prodotti in stampa.
!
! *** Sezione dichiarativa ***
  IMPLICIT NONE
  TYPE :: int_value
     REAL :: value
     TYPE(int_value),POINTER :: next_value
  END TYPE
  TYPE(int_value), POINTER :: head   ! puntatore alla testa della lista
```

```
  CHARACTER(LEN=20) :: filename      ! nome del file di dati di input
  INTEGER :: istat                   ! variabile di stato (0 = ok)
  INTEGER :: nvals = 0               ! numero di dati letti
  TYPE(int_value), POINTER :: ptr    ! puntatore al nuovo valore
  TYPE(int_value), POINTER :: ptr1   ! puntatore temporaneo per la ricerca
  TYPE(int_value), POINTER :: ptr2   ! puntatore temporaneo per la ricerca
  TYPE(int_value), POINTER :: tail   ! puntatore alla coda della lista
  REAL :: temp                       ! variabile temporanea
! *** Sezione esecutiva ***
  WRITE(*,'(1X,A)',ADVANCE='NO') "Nome del file di dati: "
  READ(*,'(A20)') filename
! Connessione del file di dati
  OPEN(UNIT=11,FILE=filename,STATUS='OLD',ACTION='READ',IOSTAT=istat )
  fileopen: IF (istat==0) THEN      ! connessione avvenuta con successo
! Dal momento che il file e' stato connesso con successo, viene letto
! il valore da ordinare, viene allocato spazio in memoria per esso,
! viene individuata la posizione che gli compete nella lista e, quindi,
! si procede al suo inserimento nella lista
     input: DO
        READ(11,*,IOSTAT=istat) temp    ! lettura del valore
        IF (istat/=0) EXIT input        ! esce per condizioni di fine-file
        nvals = nvals+1                 ! contatore
        ALLOCATE(ptr,STAT=istat)        ! alloca spazio per il nuovo nodo
        ptr%value = temp                ! immagazzina il dato letto
! Si valuta la posizione nalla lista dove inserire il nuovo valore
        new: IF (.NOT.ASSOCIATED(head)) THEN ! lista ancora vuota
           head => ptr                  ! posiziona il nuovo valore in testa
           tail => head                 ! la coda punta al nuovo valore
           NULLIFY (ptr%next_value)     ! annulla il puntatore temporaneo
        ELSE   ! Lista non vuota
           front: IF (ptr%value<head%value) THEN
! Inserimento del valore in testa alla lista
              ptr%next_value => head
              head => ptr
           ELSE IF (ptr%value>=tail%value) THEN
! Inserimento del valore in coda alla lista
              tail%next_value => ptr
              tail => ptr
              NULLIFY (tail%next_value)
           ELSE
! Ricerca della posizione dove inserire il nuovo valore
              ptr1 => head
              ptr2 => ptr1%next_value
```

```
                search: DO
                   IF ((ptr%value>=ptr1%value).AND. &
                      (ptr%value<ptr2%value)) THEN
! Inserimento del nuovo valore
                      ptr%next_value => ptr2
                      ptr1%next_value => ptr
                      EXIT search
                   END IF
                   ptr1 => ptr2
                   ptr2 => ptr2%next_value
                END DO search
             END IF front
          END IF new
       END DO input
! Stampa della lista
       ptr => head
       output: DO
          IF (.NOT.ASSOCIATED(ptr) ) EXIT    ! operazione completata
          WRITE (*,'(1X,F10.4)') ptr%value   ! stampa valore corrente
          ptr => ptr%next_value              ! aggiornamento valore corrente
       END DO output
! Operazione di connessione fallita
    ELSE fileopen
       WRITE(*,'(1X,A)') "Impossibile connettere il file specificato"
       WRITE(*,'(1X,A,I6)') "Codice errore:", istat
    END IF fileopen
    STOP
END PROGRAM inserzione
```

Così, ad esempio, supponendo che il file di testo `my_input.dat` contenga i valori:

```
2.72
3.0
-4.4
5.
2.1
9.0
10.1
-11.5
0.3
-10.1
```

si può agevolmente verificare il codice con il seguente esempio di impiego:

```
 Nome del file di dati: input.dat
```

```
   -11.5000
   -10.1000
    -4.4000
     0.3000
     2.1000
     2.7200
     3.0000
     5.0000
     9.0000
    10.1000
```

L'esempio che segue rappresenta un tipo di impiego di liste concatenate un pò più completo. In particolare, vengono previste non solo procedure per l'inserimento dei nodi e la loro produzione in output, ma anche per l'ordinamento degli oggetti della lista secondo un idoneo algoritmo di confronto nonché la deallocazione selettiva di un nodo.

```
MODULE class_object
   IMPLICIT NONE
   TYPE object
      INTEGER :: data ! unico componente della struttura
   END TYPE object

   INTERFACE OPERATOR(<)
      MODULE PROCEDURE less_than_object
   END INTERFACE
   INTERFACE OPERATOR(==)
      MODULE PROCEDURE equal_to_object
   END INTERFACE
! Di questi due operatori, il primo serve ad ordinare la lista o
! ad aggiungere un elemento, il secondo ad ordinare la lista o ad
! eliminare un elemento
CONTAINS ! definizione degli operatori sovrapposti
   FUNCTION less_than_object (obj1,obj2) RESULT(boolean)
      TYPE(object),INTENT(IN) :: obj1, obj2
      LOGICAL :: boolean
         boolean = obj1%data < obj2%data ! qui "<" e' quello standard
   END FUNCTION less_than_object

   FUNCTION equal_to_object (obj1,obj2) RESULT(boolean)
      TYPE(object),INTENT(IN) :: obj1, obj2
      LOGICAL :: boolean
         boolean = obj1%data == obj2%data ! qui "=" e' quello standard
   END FUNCTION equal_to_object
END MODULE class_object
```

```
MODULE linked_list
   USE class_object
   IMPLICIT NONE
   TYPE node ! generico nodo della lista
      PRIVATE
      TYPE(object) :: value ! componente del nodo
      TYPE(node),POINTER :: next ! puntatore al prossimo nodo
   END TYPE node
   TYPE list ! lista concatenata di nodi
      PRIVATE
      TYPE(node),POINTER :: first ! primo oggetto della lista
   END TYPE list
CONTAINS
   SUBROUTINE delete (links,obj,found)
      TYPE(list),INTENT(INOUT) :: links
      TYPE(object),INTENT(IN) :: obj
      LOGICAL,INTENT(OUT) :: found
      TYPE(node),POINTER :: previous, current
! Cerca la posizione di obj
         previous => links%first ! comincia dalla testa della lista
         current => previous%next
         found = .FALSE. ! inizializza
         DO
            IF (found.OR.(.NOT.ASSOCIATED(current))) RETURN ! fine lista
            IF (obj==current%value) THEN ! *** sovrapposto ***
               found = .TRUE.
               EXIT
            ELSE  ! si sposta sul nodo successivo nella lista
               previous => previous%next
               current => current%next
            END IF
         END DO
! Elimina il nodo se esso e' stato trovato
         IF (found) THEN
            previous%next => current%next
            DEALLOCATE(current) ! liberato lo spazio per il nodo
         END IF
   END SUBROUTINE delete

   SUBROUTINE insert(links,obj)
      TYPE(list),INTENT(INOUT) :: links
      TYPE(object),INTENT(IN) :: obj
```

```
        TYPE (node),POINTER :: previous, current
! Cerca la posizione dove inserire il nuovo oggetto
         previous => links%first ! inizializza
         current => previous%next
         DO
            IF (.NOT.ASSOCIATED(current)) EXIT ! inserisce alla fine
            IF (obj < current%value) EXIT ! *** sovrapposto ***
            previous => current
            current => current%next ! si sposta sul prossimo nodo
         END DO
! Inserisce l'oggetto prima di quello corrente (sono permessi duplicati)
         ALLOCATE(previous%next) ! alloca nuovo spazio in memoria
         previous%next%value = obj ! nuovo oggetto inserito
         previous%next%next => current
   END SUBROUTINE insert

   FUNCTION is_empty(links) RESULT(t_or_f)
      TYPE(list),INTENT(IN) :: links
      LOGICAL :: t_or_f
         t_or_f = .NOT.ASSOCIATED(links%first%next)
   END FUNCTION is_empty

   FUNCTION new() RESULT(new_list)
      TYPE(list) :: new_list
         ALLOCATE(new_list%first) ! alloca memoria per l'oggetto
         NULLIFY(new_list%first%next) ! comincia con una lista vuota
   END FUNCTION new

   SUBROUTINE print_list (links)
      TYPE(list),INTENT(IN) :: links
      TYPE(node),POINTER :: current
      INTEGER :: counter
         current => links%first%next
         counter = 0
         PRINT*,"Valore dell'oggetto:"
         DO
            IF (.NOT.ASSOCIATED(current)) EXIT ! fine lista
               counter = counter+1
               PRINT*, counter," ",current%value
               current => current%next
         END DO
   END SUBROUTINE print_list
END MODULE linked_list
```

```
PROGRAM main
! Scopo: testare il modulo linked_list
  USE linked_list
  IMPLICIT NONE
  TYPE(list) :: container
  TYPE(object) :: obj1, obj2, obj3, obj4
  LOGICAL :: delete_ok

  obj1 = object(15) ; obj2 = object(25) ! costruttore
  obj3 = object(35) ; obj4 = object(45) ! costruttore
  container = new()
  PRINT*, "Stato della lista. Lista vuota? ",is_empty(container)
  CALL insert(container,obj4) ! inserisce un oggetto
  CALL insert(container,obj2) ! inserisce un oggetto
  CALL insert(container,obj1) ! inserisce un oggetto
  CALL print_list(container)

  CALL delete(container,obj2,delete_ok)
  PRINT *, "Oggetto: ",obj2," Oggetto rimosso? ",delete_ok
  CALL print_list(container)
  PRINT*, "Stato della lista. Lista vuota? ",is_empty(container)

  CALL insert(container,obj3) ! insert object
  CALL print_list(container)
  CALL delete(container, obj1,delete_ok)
  PRINT*, "Oggetto: ",obj1," Oggetto rimosso? ",delete_ok
  CALL delete(container, obj4,delete_ok)
  PRINT*, "Oggetto: ",obj4," Oggetto rimosso? ",delete_ok
  CALL print_list(container)
  PRINT*, "Stato della lista. Lista vuota? ",is_empty(container)

  CALL delete(container,obj3,delete_ok)
  PRINT*, "Oggetto: ",obj3," Oggetto rimosso? ",delete_ok
  PRINT*, "Stato della lista. Lista vuota? ",is_empty(container)
  STOP
END PROGRAM main
```

Per completezza si riporta anche l'output del programma:

```
C:\MUPROG>main
 Stato della lista. Lista vuota?  T
 Valore dell'oggetto:
           1            15
```

```
          2            25
          3            45
Oggetto:           25  Oggetto rimosso?  T
Valore dell'oggetto:
          1            15
          2            45
Stato della lista. Lista vuota?  F
Valore dell'oggetto:
          1            15
          2            35
          3            45
Oggetto:           15  Oggetto rimosso?  T
Oggetto:           45  Oggetto rimosso?  T
Valore dell'oggetto:
          1            35
Stato della lista. Lista vuota?  F
Oggetto:           35  Oggetto rimosso?  T
Stato della lista. Lista vuota?  T
```

Si osservi, in conclusione, come le liste concatenate siano molto più flessibili dei comuni array. Infatti, come è noto, per un array statico le dimensioni devono essere fissate già in fase di compilazione il che, ad esempio, comporta che per rendere un codice di applicabilità generale è necessario dichiarare gli array con le dimensioni massime compatibili con l'assegnato problema il che rende certamente non ottimale l'impiego della memoria. Anche l'impiego di array allocabili non risolve del tutto il problema in quanto, se da un lato essi consentono di adattare la quantità di memoria allocata alle reali esigenze di calcolo, dall'altro è sempre necessario conoscere le dimensioni che devono avere gli array (ossia di quanti elementi essi debbono comporsi) prima di allocarli. Al contrario le liste concatenate consentono di allocare un solo elemento per volta per cui non è assolutamente necessario conoscere in anticipo il numero di elementi (ossia il numero di nodi) di cui si comporrà la struttura dati al termine dell'esecuzione. Questa possibilità può essere sfruttata, ad esempio, quando si vogliano risolvere equazioni differenziali alle differenze finite su domini semi-infiniti e l'infinito numerico può essere riconosciuto soltanto durante il run a valle dell'allocazione (e dell'utilizzo) di strutture dati atte a contenere le variabili del calcolo.

7.11.2 Alberi binari

Gli *alberi binari* rappresentano una delle strutture dati dinamiche più importanti dell'informatica. Un albero binario consiste di componenti ripetuti (*nodi*) arrangiati secondo una struttura ad albero invertito. L'elemento in cima all'albero è chiamato *radice* (*root*). Gli elementi situati immediatamente al di sotto di un nodo ne sono chiamati figli (*children*), di rimando, l'elemento situato direttamente al di sopra di un altro nodo ne è detto il *genitore* (*parent*). Infine, un elemento privo di figli è chiamato *foglia* (*leaf*) dell'albero.

In generale, un albero binario rappresenta una struttura dati contenente una sequenza di elementi registrati secondo l'ordine di inserimento ma riordinati secondo un assegnato criterio

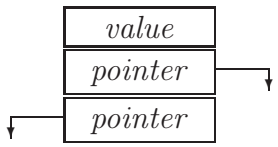


Figura 7.3: Generico nodo di albero binario

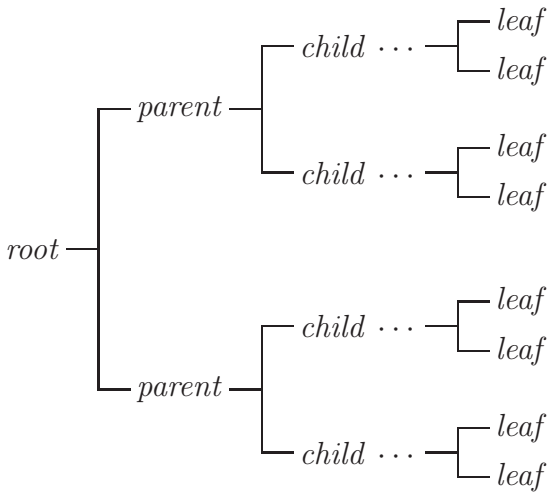


Figura 7.4: Schema di albero binario

di ordinamento. Tale ordinamento viene espresso a mezzo di una coppia di puntatori detti *puntatore sinistro* e *puntatore destro* (o anche *puntatore precedente* e *puntatore successivo*). L'elemento puntato dal puntatore sinistro precede, secondo la relazione di ordinamento applicata, l'elemento in esame, viceversa l'elemento puntato dal puntatore destro segue l'elemento in esame. Se ne deduce che un nodo debba rappresentare una *struttura* composta dal dato (o dai dati) da gestire e da una coppia di puntatori a oggetti dello stesso tipo. In un albero binario così definito, ciascun elemento è maggiore di tutti gli elementi appartenenti al sotto-albero "inferiore" (o "sinistro") e minore di quelli appartenenti al sotto-albero "superiore" (o "destro"). Da tali proprietà discendono in modo naturale gli algoritmi di ricerca, inserimento e prelievo degli elementi dell'albero. Ciò rende particolarmente efficiente le operazioni di ricerca attraverso le visite dell'albero e le operazioni di ordinamento.

Naturalmente su una struttura ad albero sarà possibile definire procedure elementari di inizializzazione dell'albero, inserimento di un nodo, estrazione di un nodo, stampa dell'elenco completo. Inoltre sarà necessario prevedere la definizione di predicati logici atti a verificare se l'albero è vuoto (ossia da se è da inizializzare) e se un dato elemento è o meno presente nell'albero.

L'esempio che segue mostra con quanta naturalezza le operazioni su un albero binario possano essere espresse mediante procedure ricorsive e l'impiego di oggetti di tipo derivato (definenti

i nodi) con componenti definiti con l'attributo `POINTER`. Il programma seguente ha lo scopo di generare una sequenza di `100` valori interi di tipo random e di organizzarli secondo un ordinamento ad albero binario. La sequenza così ordinata viene quindi riprodotta a video. Per poter comprendere appieno le operazioni svolte nel programma è opportuno tenere presente questo semplice schema il quale risulta applicabile in maniera del tutto generale a problemi di gestione di alberi binari:

- Quando viene letto il primo valore viene allocata una variabile (il *root node*) atta ad ospitarlo e i due puntatori componenti vengono annullati.
- Quando viene letto il valore successivo, viene creato un nuovo nodo per poterlo ospitare.
- Se il nuovo valore risulta minore del valore del nodo radice, il puntatore `before` di `root` è indirizzato a questo nuovo nodo.
- Se, invece, il nuovo valore risulta maggiore del valore del nodo radice, è il puntatore `after` di `root` ad essere indirizzato a questo nuovo nodo.
- Letto un nuovo valore ed allocato memoria per esso, se esso risulta maggiore del valore di `root` ma il puntatore `after` è già associato, allora esso viene confrontato con il valore puntato da `after` ed il nuovo nodo viene inserito nella posizione appropriata al di sotto di quel nodo.
- Se, invece, il nuovo valore risulta minore del valore di `root` ma il puntatore `before` è già associato, allora esso viene confrontato con il valore puntato da `before` ed il nuovo nodo viene inserito nella posizione appropriata al di sopra di quel nodo.

```
MODULE binary_tree
   IMPLICIT NONE
   SAVE
   PRIVATE greater_than, less_than, equal_to
   TYPE :: node
      INTEGER :: int
      TYPE(node), POINTER :: before
      TYPE(node), POINTER :: after
   END TYPE

   INTERFACE OPERATOR(>)
      MODULE PROCEDURE greater_than
   END INTERFACE

   INTERFACE OPERATOR(<)
      MODULE PROCEDURE less_than
   END INTERFACE
```

```
   INTERFACE OPERATOR(==)
      MODULE PROCEDURE equal_to
   END INTERFACE

CONTAINS
   LOGICAL FUNCTION greater_than(op1,op2)
      TYPE(node),intent(in) :: op1, op2
         IF (op1%int > op2%int) THEN
            greater_than = .TRUE.
         ELSE
            greater_than = .FALSE.
         END IF
   END FUNCTION greater_than

   LOGICAL FUNCTION less_than(op1,op2)
      TYPE(node), INTENT(IN) :: op1, op2
         IF (op1%int < op2%int) THEN
            less_than = .TRUE.
         ELSE
            less_than = .false.
         END IF
   END FUNCTION less_than

   LOGICAL FUNCTION equal_to(op1,op2)
      TYPE(node), INTENT(IN) :: op1, op2
         IF (op1%int == op2%int) THEN
            equal_to = .TRUE.
         ELSE
            equal_to = .FALSE.
         END IF
   END FUNCTION equal_to

END MODULE binary_tree
!
PROGRAM tree
! Crea un albero binario di 100 interi random
! --- Sezione dichiarativa ---
  USE binary_tree
  IMPLICIT NONE
  INTEGER,PARAMETER :: dim=100
  REAL,DIMENSION(dim) :: random
  INTEGER,DIMENSION(dim) :: input
  INTEGER :: i
```

```
  INTEGER :: istat
  TYPE(node), POINTER :: root ! testa dell'albero
  TYPE(node), POINTER :: temp ! contiene il valore corrente
! --- Sezione esecutiva ---
  NULLIFY(root,temp)
! Ora si valutano 100 valori reali random e li si converte
! in interi compresi tra 0 1 100
  CALL RANDOM_NUMBER(random)
  input = INT(100*random)
  WRITE(*,"('Elenco originale:',<dim>(/,2X,I10))") input
! Ora viene creato un nodo per ciascun intero e lo si inserisce nell'albero
  DO i = 1,100
     ALLOCATE(temp,STAT=istat)
     temp%int = input(i)
     CALL add_node(root,temp)
  END DO
  WRITE(*,"/,('Elenco ordinato:')")
  CALL write_node(root)
CONTAINS
   RECURSIVE SUBROUTINE add_node(ptr,new_node)
! Scopo: aggiungere un nodo ad un albero binario
      TYPE(node), POINTER :: ptr ! puntatore alla posizione corrente
      TYPE(node), POINTER :: new_node ! puntatore al nuovo nodo
      IF(.NOT.ASSOCIATED(ptr)) THEN
! Se non esiste ancora alcun albero, questo e' il primo nodo
         ptr => new_node
      ELSE
         IF (new_node < ptr) THEN
            IF(ASSOCIATED(ptr%before)) THEN
               CALL add_node(ptr%before,new_node)
            ELSE
               ptr%before => new_node
            END IF
         ELSE
            IF(ASSOCIATED(ptr%after)) THEN
               CALL add_node(ptr%after,new_node)
            ELSE
               ptr%after => new_node
            END IF
         END IF
       END IF
   END SUBROUTINE add_node
```

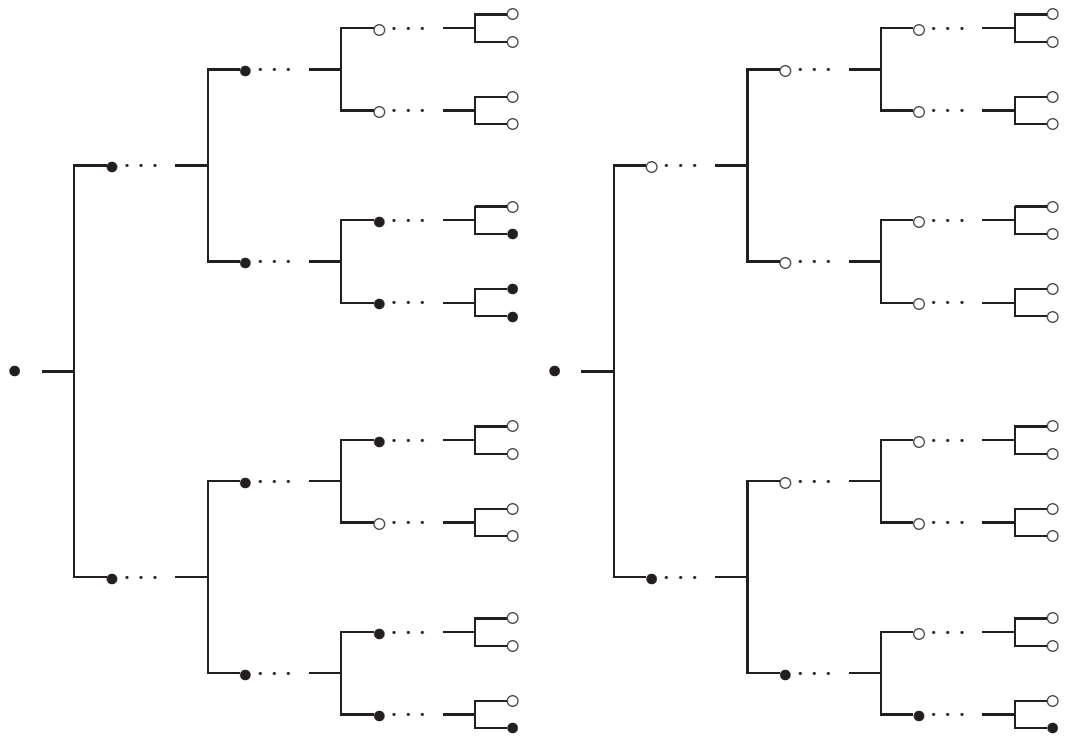
```
   RECURSIVE SUBROUTINE write_node(ptr)
      TYPE(node), POINTER :: ptr ! puntatore alla posizione corrente
! Scrive il contenuto del nodo precedente
         IF (ASSOCIATED(ptr%before)) THEN
            CALL write_node(ptr%before)
         END IF
! Stampa il contenuto del nodo corrente
         WRITE(*,"(2X,I10)") ptr%int
! Scrive il contenuto del nodo successivo
         IF(ASSOCIATED(ptr%after)) THEN
            CALL write_node(ptr%after)
         END IF
   END SUBROUTINE write_node
END PROGRAM tree
```

Al fine di verificare la bontà del codice, può essere utile osservare uno stralcio dell'output:

```
Elenco originale:
          77
          82
          14
          65
          84
         ...
          91
          54
           6
          57
          79

Elenco ordinato:
           2
           2
           3
           3
           5
         ...
          89
          90
          91
          98
          99
```

Un albero binario consente, dunque, di ordinare i dati all'interno di una struttura indipendentemente dall'ordine in cui vengono inseriti e senza operare alcuna permutazione degli elementi



(a) (b)

Figura 7.5: (a) Albero sbilanciato (b) Albero che degenera in lista lineare

inseriti. Il vantaggio di usare una struttura ad albero cresce enormemente al crescere del numero di elementi. Infatti un albero perfettamente bilanciato consente di eseguire le operazioni di ricerca con un numero di visite pari a $\log_2 n$ essendo n il numero di elementi dell'albero. Si supponga, ad esempio, di dover inserire all'interno di un database 65535 valori. Nel caso in cui si debba ricercare un valore all'interno di questo elenco, utilizzando una lista concatenata il numero medio di valori da processare sarebbe pari a $65535/2 = 32768$. Inserendo, invece, tali valori all'interno di una struttura ad albero, tutti i valori possono essere agevolmente inseriti in appena 16 "livelli" così che il numero massimo di valori da processare prima di trovare il valore desiderato è appena 16. Questo semplice esempio fa comprendere l'enorme efficienza associata alle procedure di gestione dati attraverso strutture binarie. Per tale ragione molti database sono strutturati come alberi binari.

E' da precisare, tuttavia, che non sempre una struttura ad albero è così efficiente. Poiché, infatti, l'inserimento dei nodi nell'albero dipende dall'ordine in cui i dati vengono letti può accadere che una parte di albero appaia molto più "nutrita" di un altra e l'albero sia fortemente sbilanciato, ciò aumentando il numero di livelli da processare nella visita all'albero durante le

operazioni di ricerca. In particolare la struttura può degenerare in una lista lineare qualora gli elementi vengano inseriti proprio secondo l'ordinamento prescelto. In quest'ultimo caso si perderà del tutto il vantaggio di lavorare con una struttura binaria. Entrambe queste possibilità sono schematicamente rappresentate nelle figure 7.5(a) e 7.5(b), rispettivamente. Al fine di evitare un tale problema e ridurre, così, i tempi di ricerca, molti database prevedono la possibilità di "disordinare" parzialmente i dati in ingresso attraverso particolari tecniche di *hashing* e ciò allo scopo di provvedere ad un opportuno bilanciamento dell'albero.

Si vuole concludere l'argomento presentando un altro esempio di impiego di una struttura ad albero binario. Lo scopo del seguente programma è quello di leggere (da file) un elenco del tutto casuale di nominativi e numeri di telefono e di organizzare tale elenco in forma gerarchica secondo l'ordinamento lessicografico per cognome. Ciò allo scopo di favorire la procedura di ricerca di un numero telefonico concepita per un utilizzo interattivo da parte dell'utente. In questo caso ciascun nodo rappresenterà un oggetto di tipo derivato composto da tre componenti stringa (il *nome*, il *cognome* ed il *numero di telefono*) oltre che, chiaramente, dai due puntatori ai nodi precedente e successivo. L'ordinamento alla base dell'intera struttura rispetta l'ordinamento lessicografico ASCII applicato ai cognomi e, a parità di questi, ai nomi.

```
MODULE btree
!
! Scopo: Definisce il tipo di dati del generico nodo e
!        dell'intero albero binario nonche' le operazioni
!        di inserimento di un nodo, stampa della lista
!        e ricerca di un elemento. Nel modulo vengono
!        inoltre definiti gli operatori "<", ">" e "=="
!        applicabili ad elementi di tipo nodo.
   IMPLICIT NONE

! Restrizione della visibilita' di alcuni componenti del modulo
   PRIVATE
   PUBLIC :: node, OPERATOR(>), OPERATOR(<), OPERATOR(==)
   PUBLIC :: add_node, write_node, find_node

! Dichiarazione del tipo del nodo e dell'albero
   TYPE :: node
      CHARACTER(LEN=10) :: surname
      CHARACTER(LEN=10) :: name
      CHARACTER(LEN=16) :: phone
      TYPE(node),POINTER :: before
      TYPE(node),POINTER :: after
   END TYPE

   INTERFACE OPERATOR (>)
      MODULE PROCEDURE greater_than
   END INTERFACE
```

```
   INTERFACE OPERATOR (<)
      MODULE PROCEDURE less_than
   END INTERFACE

   INTERFACE OPERATOR (==)
      MODULE PROCEDURE equal_to
   END INTERFACE

CONTAINS
   RECURSIVE SUBROUTINE add_node(ptr,new_node)
!
! Scopo: Aggiungere un nuovo nodo all'albero
!
      TYPE(node),POINTER :: ptr      ! puntatore al nodo corrente
      TYPE(node),POINTER :: new_node ! puntatore al nuovo nodo

         IF (.NOT.ASSOCIATED(ptr)) THEN
! Non esiste ancora un albero
            ptr => new_node
         ELSE IF (new_node<ptr) THEN
            IF (ASSOCIATED(ptr%before)) THEN
                CALL add_node(ptr%before,new_node)
            ELSE
               ptr%before => new_node
            END IF
         ELSE
            IF (ASSOCIATED(ptr%after)) THEN
               CALL add_node(ptr%after,new_node)
            ELSE
               ptr%after => new_node
            END IF
         END IF
   END SUBROUTINE add_node

   RECURSIVE SUBROUTINE write_node (ptr)
!
! Scopo: Stampare in maniera ordinata il contenuto dell'albero
!
      TYPE (node), POINTER :: ptr  ! puntatore al nodo corrente
! Stampa il contenuto del nodo precedente
         IF (ASSOCIATED(ptr%before)) THEN
            CALL write_node(ptr%before)
```

```
         END IF
! Stampa il contenuto del nodo corrente
         WRITE (*,"(1X,A,', ',A,1X,A)") ptr%surname,ptr%name,ptr%phone
! Stampa il contenuto del nodo successivo
         IF (ASSOCIATED(ptr%after)) THEN
            CALL write_node(ptr%after)
         END IF
   END SUBROUTINE write_node

   RECURSIVE SUBROUTINE find_node(ptr,search,error)
!
!  Scopo: Cercare un particolare nodo all'interno dell'albero.
!         La variabile "search" e' un puntatore al nome da cercare
!         e contiene il risultato della ricerca se essa ha avuto
!         successo
!
      TYPE(node),POINTER :: ptr    ! puntatore al nodo corrente
      TYPE(node),POINTER :: search ! puntatore al nodo da cercare
      INTEGER :: error             ! variabile di stato
                                   ! (0 = ok, 1 = non trovato)
         IF (search<ptr) THEN
            IF (ASSOCIATED(ptr%before)) THEN
               CALL find_node(ptr%before,search,error)
            ELSE
               error = 1
            END IF
         ELSE IF (search==ptr) THEN
            search = ptr
            error = 0
         ELSE
            IF (ASSOCIATED(ptr%after)) THEN
               CALL find_node(ptr%after,search,error)
            ELSE
               error = 1
            END IF
         END IF
   END SUBROUTINE find_node

   LOGICAL FUNCTION greater_than(op1,op2)
!
! Scopo: Verificare se operando_1 e' > operando_2
!        secondo l'ordinamento lessicografico
!
```

```
      TYPE(node),INTENT(IN) :: op1, op2

         IF (LGT(op1%surname,op2%surname)) THEN
            greater_than = .TRUE.
         ELSE IF (LLT(op1%surname,op2%surname)) THEN
            greater_than = .FALSE.
         ELSE ! i cognomi coincidono
            IF (LGT(op1%name,op2%name)) THEN
               greater_than = .TRUE.
            ELSE
               greater_than = .FALSE.
            END IF
         END IF
   END FUNCTION greater_than

   LOGICAL FUNCTION less_than(op1,op2)
!
! Scopo: Verificare se operando_1 e' < operando_2
!        secondo l'ordinamento lessicografico
!
      TYPE(node),INTENT(IN) :: op1, op2

         IF (LLT(op1%surname,op2%surname)) THEN
            less_than = .TRUE.
         ELSE IF (LGT(op1%surname,op2%surname)) THEN
            less_than = .FALSE.
         ELSE ! i cognomi coincidono
            IF (LLT(op1%name,op2%name)) THEN
               less_than = .TRUE.
            ELSE
               less_than = .FALSE.
            END IF
         END IF
   END FUNCTION less_than

   LOGICAL FUNCTION equal_to(op1,op2)
!
! Scopo: Verificare se operando_1 == operando_2
!        secondo l'ordinamento lessicografico
!
      TYPE(node),INTENT(IN) :: op1, op2

         IF ((op1%surname==op2%surname).AND.(op1%name==op2%name)) THEN
```

```
            equal_to = .TRUE.
         ELSE
            equal_to = .FALSE.
         END IF
   END FUNCTION equal_to

END MODULE btree

PROGRAM binary_tree
!
! Scopo: Legge da file una sequenza di nominativi e relativi numeri
!        di telefono e li immagazzina in un albero binario. Dopo che
!        i valori sono stati immagazzinati essi vengono stampati
!        secondo l'ordinamento lessicografico. A questo punto l'utente
!        puo' inserire un nome per verificare che lo stesso sia presente
!        nella rubrica. In caso affermativo viene stampato il record
!        a cui appartiene.
!
  USE btree
  IMPLICIT NONE
! *** Sezione dichiarativa ***
  INTEGER :: error                ! "error flag": 0 = ok
  CHARACTER(LEN=20) :: filename   ! nome del file di dati
  INTEGER :: istat                ! "variabile di stato": 0 = ok
  TYPE(node),POINTER :: root      ! puntatore al nodo root
  TYPE(node),POINTER :: temp      ! puntatore temporaneo

! *** Sezione esecutiva ***
! "Annullamento" dei nuovi puntatori
  NULLIFY(root,temp)

! Lettura del nome del file di dati
  WRITE(*,*) "Nome del file di dati: "
  READ(*,'(A20)') filename

! Connessione del file di ingresso.
  OPEN(UNIT=11,FILE=filename,STATUS='OLD',ACTION='READ',IOSTAT=istat)

  fileopen: IF (istat==0) THEN  ! connessione avvenuta con successo
! Dal momento che il file e' stato connesso con successo, viene allocato
! spazio in memoria per ciascun nodo, vengono letti i dati da inserire
! nel nodo, quindi il nodo viene inserito nell'albero
     input: DO
```

```
        ALLOCATE(temp,STAT=istat)        ! alloca il nodo
        NULLIFY(temp%before,temp%after)  ! annulla i puntatori
! Lettura dati
        READ(11,100,IOSTAT=istat) temp%surname,temp%name,temp%phone
 100    FORMAT(A10,1X,A10,1X,A16)
        IF (istat/=0) EXIT input    ! raggiunto l'"end of file"
        CALL add_node(root,temp)    ! aggiunge un nodo all'albero
     END DO input
! Stampa della lista ordinata
     WRITE (*,'(/,1X,A)') "Lista ordinata: "
     CALL write_node(root)
! Prompt per la ricerca di un elemento
     WRITE(*,'(/,1X,A)') "Nominativo da cercare nella lista: "
     WRITE(*,'(1X,A)',ADVANCE='NO') "Cognome:  "
     READ(*,'(A)') temp%surname
     WRITE(*,'(1X,A)',ADVANCE='NO') "Nome:     "
     READ (*,'(A)') temp%name
! Ricerca dell'elemento
     CALL find_node(root,temp,error)
     check: IF (error==0) THEN
        WRITE(*,'(/,1X,A)') "Il record cercato e':"
        WRITE(*,'(1X,7A)') temp%surname, ', ', temp%name, ' ', temp%phone
     ELSE
        WRITE(*,'(/,1X,A)') "L'elemento richiesto non e' disponibile"
     END IF check
  ELSE fileopen   ! connessione del file fallita
     WRITE(*,'(1X,A)') "Impossibile connettere il file specificato"
     WRITE(*,'(1X,A,I6)') "Codice errore:", istat
  END IF fileopen

END PROGRAM binary_tree
```

Può essere utile testare il precedente programma con un idoneo file di ingresso. A tale scopo, assumendo che che il file di testo `rubrica.dat` contenga il seguente elenco di nomi:

```
Allen      Barry      805-238-7999
Kent       Clark      608-555-1212
Wayne      Bruce      800-800-1111
Canary     Black      504-388-3123
El         Kara       567-234-5678
Prince     Diana      212-338-3030
Luthor     Lex        800-800-1111
Woman      Wonder     617-123-4567
Jordan     Al         713-721-0901
```

```
Reed        Oliver      618-813-1234
Gordon      Barbara     703-765-4321
Grayson     Dick        713-723-7777
```

lo si può utilizzare come file di input per la verifica della correttezza del codice:

```
C:\MYPROG>binary_tree
 Nome del file di dati: rubrica.dat

 Lista ordinata:
 Allen      , Barry      805-238-7999
 Canary     , Black      504-388-3123
 El         , Kara       567-234-5678
 Gordon     , Barbara    703-765-4321
 Grayson    , Dick       713-723-7777
 Jordan     , Al         713-721-0901
 Kent       , Clark      608-555-1212
 Luthor     , Lex        800-800-1111
 Prince     , Diana      212-338-3030
 Reed       , Oliver     618-813-1234
 Wayne      , Bruce      800-800-1111
 Woman      , Wonder     617-123-4567

 Nominativo da cercare nella lista:
 Cognome:  Kent
 Nome:     Clark

 Il record cercato e':
 Kent       , Clark      608-555-1212
```

Capitolo 8

Operazioni di I/O su file

8.1 Generalità

Le istruzioni di Input/Output forniscono un canale di comunicazione tra il programma Fortran e il mondo esterno. Del tutto in generale si può dire che un'operazione di I/O comporta la trasmissione di informazioni sottoforma di *record* da o verso *unità* (o *mezzi*) di I/O, record che possono o meno essere associati a dei *file*. Tale trasmissione, inoltre, può avvenire o meno sotto il *controllo di formato*. Ancora, la *modalità di accesso* ad un record del file può avvenire in maniera *sequenziale* o *diretta*. Tutto ciò fa comprendere come una istruzione di I/O debba contenere almeno le seguenti indicazioni:

- la *direzione* in cui avviene la trasmissione di informazioni, sia dall'esterno verso la memoria dell'elaboratore (*input*) o dalla memoria dell'elaboratore verso l'esterno (*output*);
- il *mezzo* di I/O da utilizzare;
- la lista delle *locazioni di memoria* interessate alla trasmissione delle informazioni e le modalità secondo cui tali informazioni devono essere rappresentate all'esterno.

Al fine di consentire una migliore comprensione delle istruzioni di I/O, nei prossimi paragrafi verranno esposti, dapprima, i concetti di base che riguardano record e file, per poi proseguire con un'analisi più mirata della ricca gamma di meccanismi di trasmissione dati del Fortran.

8.1.1 Record

Tutti le operazioni di I/O in Fortran hanno luogo tramite una struttura chiamata *record*. Un record è, in generale, una sequenza di uno o più *caratteri* (o di uno o più *valori*); una riga di testo è un buon esempio di record. Il Fortran distingue fra tre tipi di record:

- *record formattati*
- *record non formattati*
- *record di fine file* ("*endfile*")

Un *record formattato* è costituito da una sequenza di *caratteri* ASCII, terminata o meno con un *carriage return*, un avanzamento di linea o entrambi. La *lunghezza* di un record formattato, espressa in numero di caratteri, è variabile entro un limite predefinito dipendente dall'unità di ingresso-uscita.

Un *record non formattato* è una sequenza di *valori* e la sua interpretazione dipende dal tipo di dati ed avviene secondo modalità dipendenti dal sistema. In altri termini, tali valori vengono rappresentati nel record esterno sottoforma di "copia" delle "unità di memorizzazione" utilizzate dallo specifico elaboratore. Anche per un record non formattato la lunghezza (misurata in unità dipendenti dal sistema) può variare entro un limite predefinito, eccetto che per i *record ad accesso sequenziale* (v. oltre) *non* contenenti informazioni sulla lunghezza del record, per i quali la lunghezza può quindi essere illimitata.

Il *record di endfile* è l'*ultimo* record di un file e *non* ha lunghezza. Esso è prodotto dall'esecuzione di una speciale istruzione (`ENDFILE`). L'apposizione di un record *endfile* su un file rende impossibile ogni tentativo di leggere da (o scrivere su) quel file senza aver prima scorso all'indietro il file.

8.1.2 File

Un *file* è una sequenza di record *tutti dello stesso tipo*, esso può essere creato e reso accessibile anche con mezzi diversi dai programmi in linguaggio Fortran, ad esempio si può creare e modificare un file di testo con un normale *text editor* per poi leggerne e manipolarne le informazioni con un programma Fortran. I file contenuti in nastri e dischi magnetici sono generalmente detti *esterni*; i file ospitati, invece, nella memoria principale sono chiamati file *interni*. La *posizione* all'interno di un file si riferisce al *prossimo* record che sarà letto o scritto. Quando si accede ad un file (ossia quando lo si "apre") si è posizionati prima del primo record mentre la fine del file è subito dopo il record di *endfile*. Alcune istruzioni di I/O permettono di cambiare la posizione corrente in un file.

Il metodo utilizzato per trasferire record da (o verso) un file è detto *modalità di accesso*. L'accesso ai file avviene attraverso il loro *nome*. In generale, un file esterno può contenere sia record *formattati* che *non formattati*. In funzione, pertanto, del tipo di record che li costituiscono, i file esterni possono essere classificati come *file di testo* o come *file non di testo*. In generale, i file di tipo testo vengono utilizzati per il collegamento con l'ambiente esterno ed i supporti di I/O (tipicamente, nelle operazioni di ingresso, per la lettura di dati da tastiera o da supporto magnetico; nelle operazioni di uscita, per l'output su video, stampanti a linea o su supporti magnetici).

In funzione del metodo adoperato per accedere ai suoi record, un file può essere classificato come *ad accesso sequenziale* o *ad accesso diretto*. Più precisamente, quando un record in un file può essere letto o scritto in modo arbitrario (*casuale*), la modalità di accesso è detta *diretta*. Si accede ai singoli record attraverso un *numero di record*, che è sempre un intero positivo. Tutti i record in un file ad accesso diretto hanno la *stessa lunghezza* e contengono solo i dati attualmente scritti su di essi; in questo caso, pertanto, *non* esistono caratteri di fine record. Si noti che in un file ad accesso diretto i record possono essere riscritti ma *non* cancellati. Generalmente solo i file su disco possono usare la modalità di accesso diretto per

il trasferimento dei record. Quando, invece, i record vengono trasferiti in ordine, in maniera consecutiva, la modalità di accesso è detta *sequenziale*. In questo modo, per poter accedere ad un record del file occorre avere prima scorso tutti i record che si trovano prima di quello voluto (e quindi i record devono essere letti nel medesimo ordine in cui sono stati scritti). I record in un file ad accesso sequenziale possono, tuttavia, essere di diversa lunghezza. I file ad accesso sequenziale formattati contengono solitamente informazioni di tipo testo e ogni record ha un carattere di fine record. L'accesso sequenziale non formattato è generalmente adoperato per procedere su informazioni binarie *non codificate* come i file ".obj".

Affinché sia possibile operare su un file è necessario che questo sia "conosciuto": ciò avviene attraverso l'associazione del *nome interno del file* (valido nell'*ambiente del linguaggio*) con il *nome esterno* (valido nell'*ambiente del sistema operativo*). Questa operazione è nota come *connessione del file* ad una *unità logica*.

Per le unità di I/O "standard" quali tastiera e video, il linguaggio prevede l'uso di speciali file esterni precollegati. I file relativi alle unità di I/O standard sono da ritenersi sempre connessi per tutti i possibili programmi; essi si dicono, pertanto, *file preconnessi* e la dichiarazione di connessione, come si vedrà in seguito, è per essi implicitamente assunta.

Una particolare classe di file è quella che va sotto il nome di *file interni*. Un file interno è un particolare oggetto in grado di contenere una variabile o un array di tipo `CHARACTER`. In particolare, un file interno che contenga una sola variabile, un unico elemento di array o una sottostringa, tutti di tipo `CHARACTER`, si compone un solo record, la cui lunghezza è quella dell'entità carattere considerata. Un file interno che contenga, invece, un vettore di tipo `CHARACTER`, avrà tanti record quanti sono gli elementi dell'array, con lunghezza uguale a quella degli elementi stessi. I dati di questi file possono essere trasferiti solo con la modalità di *accesso sequenziale formattato*. Come si vedrà in seguito, solitamente questi file vengono utilizzati per la conversione del tipo di variabili da numerico a `CHARACTER`.

8.2 Le specifiche di I/O

Le istruzioni di I/O hanno tutte quante la forma:

istruzione`(`*lista_di_specificatori*`)` *lista_di_operandi*

in cui *istruzione* è la generica istruzione di I/O la cui funzione può essere una delle seguenti:

- connessione e sconnessione di un file (`OPEN`, `CLOSE`)
- interrogazione dello stato del file (`INQUIRE`)
- lettura e scrittura di record (`READ`, `WRITE`
- posizionamento del file (`REWIND`, `BACKSPACE`)
- apposizione del record di fine file (`ENDFILE`)

lista_di_operandi è tipica di ciascuna istruzione mentre *lista_di_specificatori* indica un insieme di specifiche e di informazioni ausiliarie per l'esecuzione della istruzione.

L'insieme delle istruzioni di I/O verrà dettagliatamente esaminato nei prossimi paragrafi.

8.2.1 Connessione di un file

Per poter far riferimento, in una istruzione di I/O, ad un file esterno occorre che questo sia "conosciuto" dal programma ossia *associato* o *connesso* ad una data *unità*. Con l'istruzione di connessione si realizza, così, l'associazione fra una unità logica e il nome di un file rendendone possibile l'uso nelle altre istruzioni di I/O . Al tempo stesso vengono specificate tutte quelle informazioni atte alla gestione del file, quali:

- il *tipo* del file (*formattato* o *non formattato*)
- il *metodo di accesso* (*sequenziale* o *diretto*)
- la *lunghezza dei record*
- lo *stato* del file (*nuovo*, *preesistente*, *da rimpiazzare*, *temporaneo*, *incognito*)
- l'*azione* che deve "subire" (*lettura*, *scrittura* o entrambe)
- la *gestione* di un eventuale errore nella connessione

La connessione di un file avviene tramite l'istruzione `OPEN` la quale ha la seguente sintassi:

```
OPEN([UNIT=]u[,FILE=fln][,IOSTAT=ios][,STATUS=st][,ACCESS=acc] &
     [,FORM=fm][,ACTION=act][,POSITION=pos][,DELIM=del] &
     [,PAD=pad][,RECL=rl])
```

in cui gli specificatori possono apparire in qualsiasi ordine, a meno che la parola chiave `UNIT=` non sia assente, nel qual caso l'*informazione* *u* deve essere riportata per prima.

Di seguito viene riportato il significato di ciascuno specificatore.

- La clausola `UNIT=`*u* specifica l'unità logica da associare al file; *u* è il numero identificativo dell'unità logica e rappresenta una espressione intera non negativa. Questo specificatore deve essere obbligatoriamente presente, non solo nelle istruzioni `OPEN` ma in tutte le istruzioni di I/O. Nell'espressione della specifica, la sequenza `UNIT=` può essere omessa: in tal caso lo specificatore, ridotto alla sola espressione *u*, deve essere il primo della lista.
- La clausola `FILE=`*fln* specifica il nome "esterno" del file da connettere alla unità logica *u*. E' importante osservare che, nel caso in cui il file debba essere utilizzato in forma temporanea (`STATUS='SCRATCH'`) il file *non* deve essere avere un nome e, quindi, lo specificatore `FILE=` non deve essere presente.
- La clausola `IOSTAT=`*ios* specifica il nome di una variabile intera cui è restituito lo stato dell'operazione `OPEN`. Se l'operazione di connessione ha successo la variabile *ios* sarà impostata a zero, altrimenti conterrà un valore positivo dipendente dalla specifica implementazione e relativo al tipo di problema incontrato.

- La clausola `STATUS=`*`st`* specifica lo stato del file da aprire. L'espressione carattere *`st`* può avere uno dei seguenti valori:
  - `'OLD'`: In questo caso il file deve già esistere prima che la connessione abbia luogo, in caso contrario incorrerà una condizione di errore.
  - `'NEW'`: In questo caso il file *non* deve essere già esistente ma esso viene creato all'atto della connessione, dopodiché il suo *stato* diventerà automaticamente `'OLD'`. Nel caso in cui il file esista già prima che la connessione abbia luogo, avrà luogo una condizione di errore.
  - `'REPLACE'`: Quando lo stato del file è specificato come `'REPLACE'` ed il file non esiste ancora, esso viene creato; nel caso in cui, invece, il file esista già, il vecchio file viene eliminato ed un nuovo file viene creato con lo stesso nome. In entrambi i casi lo *stato* del file, subito dopo la connessione, è automaticamente settato a `'OLD'`.
  - `'SCRATCH'`: Quando lo specificatore `STATUS` è posto uguale a `'SCRATCH'`, il file viene creato e connesso all'unità logica ma *non* può essere conservato al termine dell'esecuzione del programma o a seguito dell'esecuzione di una istruzione `CLOSE`.
  - `'UNKNOWN'`: In questo caso lo *stato* del file risulta dipendente dalla specifica implementazione Generalmente, se il file esiste già il suo stato è assunto `'OLD'`, in caso contrario è considerato `'NEW'`. Si noti che lo stato `'UNKNOWN'` è assunto come default dallo standard.

- La clausola `ACCESS=`*`acc`* specifica la modalità di accesso ai record del file per cui i possibili valori per l'espressione carattere *`acc`* possono essere `'DIRECT'` oppure `'SEQUENTIAL'`, dove quest'ultimo valore rappresenta quello di default.

- La clausola `FORM=`*`fm`* specifica se il file debba essere connesso per operazioni di I/O *formattate* oppure *non formattate*. I possibili valori per l'espressione carattere *`fm`* sono, pertanto, `'FORMATTED'` e `'UNFORMATTED'`. Se questo specificatore è omesso il valore di default assunto sarà `'FORMATTED'` se `ACCESS='SEQUENTIAL'`, `UNFORMATTED` se `ACCESS='DIRECT'`.

- La clausola `ACTION=`*`act`* specifica le operazioni che è possibile effettuare sul file connesso dall'istruzione `OPEN`. I possibili valori per l'espressione stringa *`act`* sono:
  - `'READ'`: In questo caso il file viene connesso esclusivamente per operazioni di lettura per cui è proibito l'utilizzo delle istruzioni `WRITE` e `ENDFILE`.
  - `'WRITE'`: In questo caso il file è connesso soltanto per operazioni di scrittura e l'uso dell'istruzione `READ` risulta in un errore (su alcuni sistemi, anche l'istruzione `BACKSPACE` e lo specificatore `POSITION='APPEND'` possono fallire in quanto entrambi implicano, implicitamente, una operazione di lettura).
  - `'READWRITE'`: In questo caso, che è anche quello di default, non esistono restrizioni all'uso del file che, una volta connesso, può essere utilizzato per qualsiasi operazione valida di I/O.

- La clausola `POSITION=`*`pos`* stabilisce la posizione del puntatore del file dopo che la connessione abbia avuto luogo, ossia il punto del file a partire dal quale avrà inizio la successiva operazione di I/O. I possibili valori per l'espressione stringa *`pos`* sono:

  - `'REWIND'`: Il file viene posizionato prima del primo record.
  - `'APPEND'`: Se il file è connesso con stato `'OLD'`, esso è posizionato immediatamente prima del record `endfile` se ne esiste uno, altrimenti in coda al file.
  - `'ASIS'`: Se il file viene creato insieme alla connessione, il suo puntatore viene posizionato nel punto iniziale del file. Se, invece, il file esiste e risulta già connesso, esso viene aperto senza che venga modificata la sua posizione. Infine, se il file esiste ma non risulta già connesso, l'effetto dello specificatore sulla sua posizione *non* è specificata dallo standard del linguaggio ma risulta dipendente dal particolare sistema di elaborazione.

  Si noti che lo specificatore `POSITION` è compatibile soltanto con file connessi ad accesso sequenziale e che, in sua assenza, il valore assunto di default è `ASIS`.

- La clausola `DELIM=`*`del`* specifica il tipo di delimitatore utilizzato nelle operazioni di uscita guidate da lista o con formato `NAMELIST` per le costanti di tipo `CHARACTER`. I possibili valori per questo specificatore sono:

  - `'QUOTE'`: Le costanti carattere saranno prodotte in uscita fra singoli apici mentre un eventuale apice all'interno della stringa sarà riportato raddoppiato.
  - `'APOSTROPHE'`: Le costanti carattere saranno prodotte in uscita fra doppi apici mentre un eventuale apice doppio all'interno della stringa sarà riportato raddoppiato.
  - `'NONE'`: Le costanti stringa saranno prodotte in uscita prive di delimitatori.

  Si noti che questo specificatore può comparire soltanto se il file è connesso per operazioni formattate e che il valore di default è `'NONE'`.

- La clausola `PAD=`*`pad`* specifica se un record formattato di input debba essere o meno "completato" con eventuali spazi vuoti. I possibili valori per la stringa *`pad`* sono:

  - `'YES'`: In questo caso un record in input verrà completato con caratteri *blank* qualora la specificazione di formato richieda più dati di quanti ne contenga il record.
  - `'NO'`: In questo caso un record in input *non* verrà completato con caratteri *blank* per cui il record di input deve contenere un numero di caratteri almeno pari a quello richiesto dalla lista di input e dalla relativa specificazione di formato.

  Il valore di default per lo specificatore `PAD` è `'YES'`. Si noti che esso non ha alcun effetto sulle operazioni di output.

- La clausola `RECL=`*`rl`* , dove *`rl`* rappresenta una espressione intera, restituisce la lunghezza di un record non formattato, espressa in unità che dipendono dal particolare processore,

oppure il numero di caratteri dei record formattati. Per file ad accesso diretto questo specificatore specifica la lunghezza dei record ed è obbligatorio. Per file ad accesso sequenziale esso è opzionale e specifica la lunghezza massima per un record, con un valore di default che dipende dal processore.

Un esempio di istruzione `OPEN` è il seguente:

```
OPEN(UNIT=11,IOSTAT=ios,FILE='archivio',STATUS='NEW', &
     ACCESS='DIRECT',RECL=100)
```

Un'operazione perfettamente equivalente è realizzata dalle seguenti istruzioni:

```
k = 5
dim = 100
filename = "archivio"
OPEN(UNIT=2*k+1,IOSTAT=ios,FILE=filename,STATUS='NEW', &
     ACCESS='DIRECT',RECL=dim)
```

In entrambi i casi si ha l'effetto di creare un file ad accesso diretto, non formattato, con nome esterno `archivio`, caratterizzato da record di lunghezza `100`. Il file così creato è connesso all'unità logica `11`. Un eventuale fallimento nella connessione del file può essere rilevato dal valore della variabile intera `ios` (che, nel caso, assumerebbe valore positivo dipendente dal processore) e, pertanto, non provocherà un arresto del sistema. La clausola `IOSTAT=` può tornare molto utile al fine di evitare di riscrivere su un file già esistente:

```
OPEN(UNIT=ilun,FILE=infile,FORM='FORMATTED', &
     ACCESS='SEQUENTIAL', STATUS='NEW',IOSTAT=rstat)
IF(rstat/=0) THEN
   PRINT*, "Errore durante l'apertura del file: ", infile
   PRINT*, "Assicurarsi che il file non esista gia'!"
END IF
```

Si noti che uno stesso file non può risultare connesso contemporaneamente a due distinte unità logiche. Nulla vieta, però, che un file possa essere riconnesso alla stessa o ad un'altra unità dopo essere stato sconnesso.

8.2.2 Sconnessione di un file

Al termine dell'esecuzione di un programma ogni file che sia stato connesso viene automaticamente *sconnesso* ("chiuso"). Se, tuttavia, si desidera operare tale sconnessione prima della fine del programma, ad esempio per riconnettere quel file ad altra unità, è possibile utilizzare l'istruzione `CLOSE` la cui sintassi è:

```
CLOSE([UNIT=]u[,IOSTAT=ios][,STATUS=st])
```

in cui gli specificatori possono apparire in qualsiasi ordine, a meno che la parola chiave `UNIT=` non sia assente, nel qual caso l'*unità logica* `u` deve essere specificata per prima.

Di seguito viene elencato il significato di ciascuno specificatore.

- La clausola `UNIT=`*`u`* specifica l'unità logica da disconnettere ed ha lo stesso significato visto nella frase `OPEN`.

- La clausola `IOSTAT=`*`ios`*, in cui *`ios`* rappresenta il nome di una variabile intera cui è restituito lo stato dell'operazione `CLOSE`, fornisce informazioni circa la riuscita o meno dell'operazione di sconnessione. In particolare, se l'operazione di chiusura ha successo la variabile *`ios`* sarà posta pari a zero, altrimenti le verrà assegnato un valore positivo dipendente dalla specifica implementazione e relativo al tipo di problema incontrato.

- La clausola `STATUS=`*`st`* specifica lo stato del file da chiudere. L'espressione carattere *`st`* può assumere valore `'KEEP'` o `'DELETE'`. Nel primo caso il file che era connesso all'unità logica *`u`* può essere successivamente connesso alla medesima o ad un'altra unità. Se, invece, *`st`* vale `'DELETE'` il file connesso all'unità *`u`* cessa di esistere. In entrambi i casi, sarà possibile utilizzare nuovamente l'unità *`u`* per successive operazioni di connessione.

  Si noti che applicare il valore `'KEEP'` per chiudere un file connesso con stato `'SCRATCH'` determina l'insorgere di una condizione di errore. Se lo specificatore `STATUS` è omesso, viene automaticamente assunta la dichiarazione `'KEEP'`, a meno che il file in questione non sia stato connesso con la specifica `STATUS='SCRATCH'`, nel qual caso viene assunta la dichiarazione `'DELETE'`. Si osservi, infine, che una istruzione `CLOSE` può essere eseguita anche su file inesistenti o non connessi, con il risultato di una istruzione vuota.

  A titolo di esempio si considerino le seguenti istruzioni:

```
OPEN(UNIT=11,FILE='dati.dat',STATUS='NEW',POSITION='REWIND')
...
CLOSE(11,STATUS='KEEP')
...
OPEN(UNIT=7,FILE='dati.dat',POSITION='APPEND')
```

  Esse implicano la creazione del file `dati.dat` e la sua connessione all'unità logica `11`, quindi la sconnessione da quella unità e la sua successiva riconnessione all'unità `10`.

8.2.3 Interrogazione dello stato di un file

L'istruzione `INQUIRE` consente di analizzare lo stato di un file, ossia l'insieme dei parametri che lo caratterizzano in assoluto e nei riguardi di una sua eventuale connessione. In particolare, consente di ottenere le seguenti informazioni:

- se il file (o l'unità) esiste
- se il file (o l'unità) è connessa
- il nome esterno del file connesso (se si fornisce l'unità logica)
- l'unità logica connesso (se si fornisce il nome esterno del file)

- il *tipo* del file (*di testo* o *non di testo*)
- il metodo di accesso per il quale il file è connesso
- i metodi di accesso consentiti per il file
- la lunghezza dei record del file
- la posizione del meccanismo di lettura/scrittura

Esistono tre differenti modalità di utilizzo dell'istruzione `INQUIRE`:

- Il modo *inquire-by-file*, con il quale vengono richieste informazioni relative alle caratteristiche di un file, identificandolo con il suo nome.
- Il modo *inquire-by-unit*, con il quale vengono richieste informazioni relative alle caratteristiche di un file, identificandolo con la sua unità logica.
- Il modo *inquire-by-output list*, che ha un significato totalmente diverso e sarà analizzato per ultimo.

La sintassi dei primi due meccanismi è la seguente:

inquire-by-file:

`INQUIRE(FILE=`*`fln`*`,`*`lista_di_inquire`*`)`

inquire-by-unit:

`INQUIRE([UNIT=]`*`u`*`,`*`lista_di_inquire`*`)`

in cui, come di consueto, *`u`* è una espressione intera specificante una unità logica mentre *`fln`* è una espressione costante carattere il cui valore rappresenta il nome esterno del file di lavoro. La *`lista_di_inquire`*, invece, rappresenta una lista di specificatori (opzionali), separati da virgole. L'elenco di tutti i possibili specificatori è riportato di seguito unitamente al significato di ciascuno.

- `IOSTAT=`*`ios`* ha il significato già descritto relativamente all'istruzione `OPEN`. Si ricorda che la funzione di questo specificatore è quello di consentire la gestione di eventuali condizioni di errore e che l'espressione intera *`ios`* assume valore zero in assenza di errori oppure un valore positivo (dipendente dal processore) in caso di errore.
- `EXIST=`*`ex`* rappresenta il predicato "il file (o l'unità) esiste", sicché la variabile logica *`ex`* assumerà valore `.TRUE.` oppure `.FALSE.` a seconda che il file (o l'unità) a cui l'istruzione `INQUIRE` fa riferimento esista o meno.
- `OPENED=`*`open`* rappresenta il predicato "il file (o l'unità) è connesso", sicché la variabile logica *`open`* assumerà valore `.TRUE.` oppure `.FALSE.` a seconda che il file [risp. l'unità] a cui l'istruzione `INQUIRE` fa riferimento sia o meno connesso ad un'unità [risp. ad un file].

- `NUMBER=`*num* produce il valore dell'unità logica connessa al file a cui l'istruzione `INQUIRE` fa riferimento. Tale valore, che viene assegnato alla variabile intera *num*, risulta pari a `-1` nel caso in cui il file in oggetto non sia connesso ad alcuna unità.

- `NAMED=`*nmd* rappresenta il predicato "il file ha un nome", sicché la variabile logica *nmd* assumerà valore `.TRUE.` oppure `.FALSE.` a seconda che il file a cui l'istruzione `INQUIRE` fa riferimento abbia o meno un nome.

- `NAME=`*nam* assegna alla variabile stringa *nam* il nome del file che è argomento di `INQUIRE`. Il risultato è indefinito se il file non ha un nome.

- `ACCESS=`*acc* assegna alla variabile stringa *acc* uno dei valori `'DIRECT'` o `'SEQUENTIAL'` a seconda del metodo di accesso al file argomento di `INQUIRE`, oppure `'UNDEFINED'` se non esiste una connessione.

- `SEQUENTIAL=`*seq* [risp. `DIRECT=`*dir*] assegna alla variabile stringa *seq* [risp. *dir*] il valore `'YES'` o `'NO'` a seconda che il file a cui l'istruzione `INQUIRE` fa riferimento *possa* essere o meno aperto per *accesso sequenziale* [risp. *diretto*], oppure `'UNDEFINED'` se questa informazione non è disponibile.

- `FORM=`*frm* assegna alla variabile stringa *frm* uno dei valori `'FORMATTED'` o `'UNFORMATTED'` a seconda del tipo di file interrogato, oppure `'UNDEFINED'` se non esiste una connessione.

- `FORMATTED=`*fmt* [risp. `UNFORMATTED=`*unf*] assegna alla variabile stringa *fmt* [risp. *unf*] il valore `'YES'` o `'NO'` a seconda che il file interrogato *possa* essere o meno aperto per *operazioni formattate* [risp. *non formattate*], oppure `'UNDEFINED'` se questa informazione non può essere fornita.

- `RECL=`*rl* assegna alla variabile intera *rl* il valore della lunghezza del record di un file connesso per accesso diretto, oppure la massima lunghezza del record di un file connesso per accesso sequenziale. Per record formattati questo valore è espresso come numero di caratteri, in caso contrario in unità dipendenti dal processore. La variabile *rl* resta indefinita se non esiste una connessione.

- `NEXTREC=`*nr* assegna alla variabile intera *nr* il numero, incrementato di un'unità, dell'ultimo record letto o scritto. Se nessun record è ancora stato scritto o letto, il valore assegnato a *nr* è `1`. Se, infine, il file non è connesso per accesso diretto oppure la posizione è indeterminata a causa di un precedente errore, il valore di *nr* risulta indefinito.

- `POSITION=`*pos* assegna alla variabile stringa *pos* uno dei valori `'REWIND'`, `'APPEND'` o `'ASIS'` a seconda di quanto specificato nella corrispondente istruzione `OPEN`, sempre che il file non sia stato "riposizionato" dopo la sua connessione. Se il file non è connesso oppure risulta connesso per accesso diretto, il valore di *pos* è `'UNDEFINED'`. Se, infine, il file è stato riposizionato dopo la sua connessione, il valore di *pos* dipende dal processore.

- `ACTION=`*`act`* assegna alla variabile stringa *`act`* uno dei valori `'READ'`, `'WRITE'` o `'READWRITE'` a seconda di quanto specificato nella corrispondente istruzione `OPEN`, oppure `'UNDEFINED'` se non esiste una connessione.

- `READ=`*`rd`* [risp. `WRITE=`*`wr`*] assegna alla variabile stringa *`rd`* [risp. *`wr`*] il valore `'YES'` o `'NO'` a seconda che il file interrogato *possa* essere o meno usato per operazioni di lettura [risp. di scrittura], oppure `'UNDEFINED'` se questa informazione non è disponibile.

- `READWRITE=`*`rw`* assegna alla variabile stringa *`rw`* il valore `'YES'` o `'NO'` a seconda che il file interrogato *possa* essere o meno usato per operazioni di lettura/scrittura, oppure `'UNDEFINED'` se questa informazione non è disponibile.

- `DELIM=`*`del`* assegna alla variabile stringa *`del`* uno dei valori `'QUOTE'`, `'APOSTROPHE'` o `'NONE'` a seconda di quanto specificato nella corrispondente istruzione `OPEN` (o dalla condizione di default), oppure `'UNDEFINED'` se non esiste una connessione oppure se il file in oggetto non è connesso per operazioni di I/O formattate.

- `PAD=`*`pad`* assegna alla variabile stringa *`pad`* il valore `YES` o `NO` a seconda a seconda di quanto specificato nella corrispondente istruzione `OPEN` (o dalla condizione di default).

  Si noti che una stessa variabile può essere associata ad un solo specificatore nell'ambito della medesima istruzione `INQUIRE`.

  Un esempio di utilizzo di istruzione `INQUIRE` è il seguente:

```
LOGICAL :: ex, op
CHARACTER(LEN=11) :: nam, acc, seq, frm
INTEGER :: ios, irec, nr
...
OPEN(UNIT=2,IOSTAT=ios,FILE='database',STATUS='NEW',   &
     ACCESS='DIRECT',RECL=100)
...
INQUIRE(UNIT=2,EXIST=ex,OPENED=op,NAME=nam,ACCESS=acc, &
        SEQUENTIAL=seq,FORM=frm,RECL=irec,NEXTREC=nr)
WRITE(*,100) ex,op,nam,acc,seq,frm,irec,nr
100 FORMAT(2(1X,L1,/),4(1X,A11,/),2(1X,I3,/))
```

  Supponendo, ad esempio, che fra le istruzioni `OPEN` e `INQUIRE` non abbiano avuto luogo istruzioni di lettura o scrittura, l'output del precedente frammento di programma sarebbe:

```
T
T
database
DIRECT
NO
UNFORMATTED
100
```

```
1
```

L'ultima forma dell'istruzione `INQUIRE`, ossia l'*inquire-by-output list* consta di un solo specificatore, `IOLENGTH`, e si presenta con la seguente sintassi:

```
INQUIRE(IOLENGTH=length) lista_di_output
```

in cui *length* rappresenta una variabile intera mentre *lista_di_output* è una lista di entità che si presenta nella stessa forma di una lista di output per l'istruzione `WRITE`. L'effetto di questa istruzione è quello di assegnare alla variabile *length* la lunghezza del record che risulterebbe dall'uso della *lista_di_output* in una istruzione `WRITE` non formattata. La conoscenza della suddetta lunghezza, che viene espressa in unità dipendenti dal processore, può essere utile, ad esempio, per stabilire se una data lista di output è troppo lunga per essere contenuta in un record la cui lunghezza sia stata imposta nella istruzione `OPEN` con lo specificatore `RECL=`.

A titolo di esempio, si osservi la seguente istruzione:

```
INQUIRE (IOLENGTH=len) a, b
```

Il suo scopo potrebbe essere quello di calcolare il valore `len` da assegnare allo specificatore `RECL=` in una istruzione `OPEN` che connetta un file per accesso diretto non formattato. Se, invece, è stato già specificato un valore per la clausola `RECL=`, è possibile fare un "check" su `len` per verificare che la lunghezza del record di uscita formato da `a` e `b` sia effettivamente minore o uguale della lunghezza di record specificata.

Le tre forme dell'istruzione di interrogazione sono, probabilmente, fra le istruzioni più ostiche da gestire, tuttavia esse forniscono un metodo molto potente e "portabile" per gestire le operazioni di allocazione e deallocazione dinamica di file, il tutto sotto il completo controllo del programma.

8.2.4 Operazioni di lettura su file

L'istruzione `READ` ha lo scopo di leggere una lista di valori dal file associato ad una data unità logica, convertire questi valori in accordo con un eventuale descrittore di formato e immagazzinarli nelle variabili specificate nella *lista di input*.

L'istruzione si caratterizza per la seguente sintassi:

```
READ([UNIT=]u[,[FMT=]fmt][,IOSTAT=ios][,REC=rn] &
     [,[NML=]lista][,ADVANCE=adv][,SIZE=size]) lista_di_input
```

dove, però, non tutti gli specificatori possono essere usati insieme, come si avrà modo di capire tra breve. Il significato degli specificatori validi per l'istruzione `READ` è riassunto di seguito.

- La clausola `UNIT=`*u* specifica l'unità logica associata al file i cui record devono essere sottoposti a lettura; *u* è il numero identificativo dell'unità logica e rappresenta una espressione intera non negativa. Questo specificatore deve essere obbligatoriamente presente, tuttavia

la parola chiave `UNIT=` può essere omessa nel qual caso lo specificatore, ridotto alla sola espressione *u*, deve essere il primo della lista.

- La clausola `FMT=`*fmt* ha il compito di specificare il *formato* con cui i dati letti in input devono essere interpretati. Pertanto *fmt* può assumere uno dei seguenti valori:
  - l'etichetta di una istruzione `FORMAT`;
  - una stringa di caratteri specificante informazioni sul formato di lettura (*formato contestuale*);
  - un asterisco (`*`), ad indicare una operazione di lettura *diretta da lista*.

  Si noti che la parola chiave `FMT=` può essere omessa nel qual caso lo specificatore (quando presente), ridotto alla sola espressione *fmt*, deve essere il secondo della lista.

- La clausola `IOSTAT=`*ios* ha il compito di "registrare", nella variabile intera *ios*, lo stato dell'operazione di lettura al fine di consentire la gestione di eventuali condizioni di errore. In particolare, al termine dell'operazione di lettura può presentarsi uno dei seguenti casi:
  - La variabile *ios* vale zero: ciò indica che l'operazione di lettura è avvenuta con successo.
  - La variabile *ios* ha un valore positivo (dipendente dal processore): l'operazione di lettura è fallita.
  - La variabile *ios* vale `-1`: è stata incontrata una condizione di *fine file*.
  - La variabile *ios* vale `-2`: è stata incontrata una condizione di *fine record* in una operazione di I/O *non-advancing*.

- La clausola `REC=`*rn*, in cui *rn* rappresenta una variabile (o una espressione) di tipo intero, specifica il numero rappresentativo del record che deve essere letto in un file connesso ad accesso diretto. L'utilizzo di questo specificatore in operazioni di lettura su file connessi con la clausola `ACCESS='SEQUENTIAL'` risulta, chiaramente, in un errore.

- La clausola `NML=`*lista* specifica la lista con nome che deve essere letta. Si noti che questo specificatore è incompatibile con la clausola `FMT=`*fmt*. Le operazioni di lettura con meccanismo *namelist* sono state già descritte nel capitolo 4 relativamente a dispositivi standard di ingresso per cui nel seguito ci si limiterà a descrivere le peculiarità di questo tipo di lettura relativamente ad operazioni su file. Si noti che la parola chiave `NNL=` può essere omessa nel qual caso lo specificatore (quando presente), ridotto alla sola espressione *lista*, deve essere il secondo della lista.

- La clausola `ADVANCE=`*adv* specifica se la lettura di un record debba essere o meno seguita dall'avanzamento al record successivo, ossia se il buffer di input debba essere o meno scartato al termine dell'operazione `READ`. I possibili valori per la variabile stringa *adv* sono `'YES'` (che è anche il valore di default) e `'NO'`. Nel primo caso gli eventuali dati presenti nel buffer di input che non sono stati letti vengono eliminati; nel secondo caso,

invece, il buffer di input viene conservato e i restanti dati vengono utilizzati in lettura dalla successiva istruzione `READ`. Si noti che questa clausola ha significato soltanto se la lettura coinvolge un file formattato connesso per operazioni ad accesso sequenziale.

- La clausola `SIZE=`*`size`* specifica il numero di caratteri letti dal buffer di input durante una istruzione `READ` *non advancing*. Lo specificatore in esame, dunque, assume un significato esclusivamente in presenza di una clausola `ADVANCE='NO'`.

8.2.5 Operazioni di scrittura su file

L'istruzione `WRITE` ha lo scopo di produrre sul dispositivo di output associato ad una data unità logica il valore delle espressioni indicate in una *lista di output* con un formato regolato da un eventuale descrittore.

L'istruzione si caratterizza per la seguente sintassi:

```
WRITE([UNIT=]u[,[FMT=]fmt][,IOSTAT=ios][,REC=rn] &
    [,[NML=]lista][,ADVANCE=adv]) lista_di_output
```

dove, però, non tutti gli specificatori possono essere usati insieme, come si avrà modo di capire tra breve. Il significato degli specificatori validi per l'istruzione `WRITE` è riassunto di seguito.

- La clausola `UNIT=`*`u`* specifica l'unità logica associata al file i cui record devono essere sottoposti a scrittura; *`u`* è il numero identificativo dell'unità logica e rappresenta una espressione intera non negativa. Questo specificatore deve essere obbligatoriamente presente, tuttavia la parola chiave `UNIT=` può essere omessa nel qual caso lo specificatore, ridotto alla sola espressione *`u`*, deve essere il primo della lista.

- La clausola `FMT=`*`fmt`* ha il compito di specificare il *formato* con cui i dati letti devono essere prodotti in output. Pertanto *`fmt`* può assumere uno dei seguenti valori:

  – l'etichetta di una istruzione `FORMAT`;

  – una stringa di caratteri specificante informazioni sul formato di scrittura (*formato contestuale*);

  – un asterisco (`*`), ad indicare una operazione di scrittura *diretta da lista*.

  Si noti che la parola chiave `FMT=` può essere omessa nel qual caso lo specificatore, ridotto alla sola espressione *`fmt`*, deve essere il secondo della lista.

- La clausola `IOSTAT=`*`ios`* ha il compito di "registrare", nella variabile intera *`ios`*, lo stato dell'operazione di scrittura al fine di consentire la gestione di eventuali condizioni di errore. Contrariamente a quanto avviene per l'istruzione `READ`, non potendosi presentare condizioni di errore di fine record o di fine file, a seguito dell'operazione di scrittura la variabile *`ios`* può assumere soltanto il valore zero (rappresentativo di un'operazione di scrittura avvenuta con successo) oppure un valore positivo dipendente dal processore (ad indicare una operazione di scrittura abortita).

- La clausola `REC=`*`rn`*, in cui *rn* rappresenta una variabile (o una espressione) di tipo intero, specifica il numero rappresentativo del record che deve essere scritto in un file connesso ad accesso diretto. L'utilizzo di questo specificatore è, pertanto, limitato a operazioni di scrittura su file connessi con la clausola `ACCESS='DIRECT'`.

- La clausola `NML=`*`lista`* specifica la lista con nome che deve essere prodotta in uscita. Come per l'istruzione `READ`, anche in questo caso lo specificatore è incompatibile con la clausola `FMT=`*`fmt`*. Si noti che la parola chiave `NML=` può essere omessa nel qual caso lo specificatore, ridotto alla sola espressione *`lista`*, deve essere il secondo della lista.

- La clausola `ADVANCE=`*`adv`* specifica se la scrittura di un record debba essere o meno seguita dall'avanzamento al record successivo. I possibili valori per la variabile stringa *`adv`* sono `'YES'` (che è anche il valore di default) e `'NO'`. Si noti che questa clausola ha significato soltanto in presenza di file formattati connessi per operazioni ad accesso sequenziale.

8.2.6 Istruzioni di posizionamento di un file

Durante le operazioni di I/O su un file esterno connesso per accesso sequenziale è talvolta necessario avere un maggiore controllo sulla posizione del record da leggere o da scrivere. In particolare può essere necessario alterare la posizione corrente del puntatore al file, ad esempio spostandosi all'interno del record corrente, o fra due record, all'inizio del primo record (*punto iniziale*) oppure dopo l'ultimo record (*punto terminale*). Allo scopo il linguaggio possiede tre istruzioni di posizionamento che saranno oggetto di questo paragrafo.

L'istruzione `BACKSPACE`

Può accadere talvolta che l'ultimo record introdotto debba essere sostituito da un nuovo record (ossia "riscritto"), oppure che l'ultimo record letto debba essere riletto. A tale scopo il Fortran mette a disposizione l'istruzione `BACKSPACE` che si caratterizza per la seguente sintassi:

```
BACKSPACE([UNIT=]u [,IOSTAT=ios])
```

in cui, come di consueto, *`u`* è una espressione intera il cui valore rappresenta una unità logica mentre *`ios`* è una variabile intera rappresentativa della riuscita o meno dell'operazione (i valori possibili per *`ios`* sono gli stessi visti per l'istruzione `READ`). Come sempre, l'ordine degli specificatori può essere alterato tranne nel caso in cui la parola chiave `UNIT=` sia assente, nel qual caso l'unità logica deve essere specificata per prima.

L'azione di questa istruzione è quella di posizionare il file prima del record corrente se esso è attualmente posizionato all'interno di un record, oppure prima del record precedente se il file è posizionato fra due record consecutivi. Un qualsiasi tentativo di applicare l'istruzione `BACKSPACE` quando il file è già posizionato all'inizio del file non produce alcun cambiamento alla posizione del file. Se il file è posizionato dopo il record di *endfile* l'istruzione `BACKSPACE` produce un "arretramento" del file a monte di questo record. Non è possibile applicare l'istruzione `BACKSPACE` ad un file non esistente o non connesso né è possibile applicare questa operazione ad un record scritto con formato libero oppure prodotto con meccanismo *namelist*.

Chiaramente una successione di `BACKSPACE` consecutivi produce l'arretramento del corrispondente numero di record.

Si noti, tuttavia, che questa operazione è molto onerosa in termini di risorse del processore, pertanto deve essere usata con molta parsimonia.

L'istruzione `REWIND`

```
REWIND([UNIT=]u[,IOSTAT=ios])
```

in cui, come per `BACKSPACE`, *u* è una espressione intera il cui valore rappresenta una unità logica mentre *ios* è una variabile intera rappresentativa della riuscita o meno dell'operazione (i valori possibili per *ios* sono gli stessi visti per l'istruzione `READ`). Come sempre, l'ordine degli specificatori può essere alterato tranne nel caso in cui la parola chiave `UNIT=` sia assente, nel qual caso l'unità logica deve essere specificata per prima.

L'effetto dell'istruzione `REWIND` è quello di posizionare il file all'inizio del primo record del file associato all'unità logica *ios*. Naturalmente, se il file è già posizionato nel suo punto iniziale, l'istruzione `REWIND` non avrà alcun effetto. Contrariamente a quanto avviene con `BACKSPACE`, è possibile applicare l'istruzione `REWIND` anche ad un file non esistente o non connesso, senza che ciò abbia un qualche effetto.

L'istruzione `ENDFILE`

La fine di un file connesso per acceso sequenziale è normalmente contraddistinta da uno speciale record detto *record di endfile*. Tale record è automaticamente applicato dall'istruzione `CLOSE` oppure a seguito della normale terminazione del programma. La presenza di tale record può comunque essere assicurata mediante l'esecuzione dell'istruzione `ENDFILE` la quale ha la seguente forma:

```
ENDFILE([UNIT=]u[,IOSTAT=ios])
```

in cui, come di consueto, *u* è una espressione intera il cui valore rappresenta una unità logica mentre *ios* è una variabile intera rappresentativa dell'esito dell'operazione (i valori possibili per *ios* sono gli stessi che per l'istruzione `READ`). Come sempre, l'ordine degli specificatori può essere alterato tranne nel caso in cui la parola chiave `UNIT=` sia assente, nel qual caso l'unità logica deve essere specificata per prima.

Se il file può essere connesso anche per accesso diretto, soltanto i record che precedono quello di *endfile* verranno considerati e soltanto essi, pertanto, verranno eventualmente processati in operazioni di I/O successive ad una connessione per accesso diretto.

Si noti, in conclusione, che se un file è connesso ad un'unità logica ma non esiste esso verrà creato e su di esso verrà prodotto il record di *endfile*.

8.3 Gestione dei file ad accesso sequenziale

L'insieme delle istruzioni fin qui esaminate consente di gestire un file sequenziale in maniera assolutamente completa. E' bene ricordare che il metodo di accesso di un file (sia esso sequen-

ziale oppure diretto) *non* è una proprietà intrinseca del file, ma viene determinato all'atto della connessione. Infatti un file può potenzialmente essere operato in modo sequenziale e/o diretto in dipendenza di un insieme di circostanze. In particolare, una apposita interrogazione al file può essere usata per sapere se un determinato metodo di accesso è consentito per uno specifico file; una volta connesso, il file è comunque sottoposto ad un unico metodo di acceso ma nulla vieta che in una precedente o successiva connessione il metodo sia diverso, purché consentito.

Un file connesso con accesso sequenziale si caratterizza per il fatto che i record sono processati nello stesso ordine in cui essi appaiono nel file. Inoltre, in ogni istante è definita una posizione del file, per cui in ogni istante il file o è nella sua posizione iniziale oppure nell'intervallo fra due record. L'accesso sequenziale è caratterizzato, inoltre, dal fatto che l'operazione di lettura o di scrittura avviene sempre sul record *successivo* rispetto al punto in cui il file staziona (sul primo record se il file staziona nella posizione iniziale) e che dopo ogni operazione di lettura o scrittura il file avanza di un record.

Occorre mettere in evidenza una importante caratteristica dei file connessi per accesso sequenziale: quando un record viene scritto su di un file sequenziale, esso è l'*ultimo* record del file, così ogni eventuale altro record successivo presente sul file viene distrutto. Non è, pertanto, possibile usare le istruzioni `BACKSPACE` o `REWIND` per posizionarsi su un certo record per riscriverlo lasciando i successivi record inalterati. Dunque, la scrittura del nuovo record è possibile solo perdendo tutti i record successivi, per cui le istruzioni di posizionamento vengono utilizzate, di solito, soltanto per leggere i record di un file.

Sebbene non sia ammessa la riscrittura selettiva di un record, è tuttavia possibile l'aggiunta di nuovi record in coda all'ultimo record utile del file (operazione di *append*).

La sintassi per le operazioni di connessione e di lettura/scrittura di un file ad accesso sequenziale di tipo *advancing* è le seguente:

```
OPEN([UNIT=]u,FILE=fln[,FORM=fm],[ACCESS='SEQUENTIAL'][,RECL=rl] &
     [,IOSTAT=ios][,STATUS=st][,ACTION=act][,DELIM=del][,PAD=pad] &
     [,POSITION=pos])

READ([UNIT=]u[,[FMT=]fmt][,IOSTAT=ios][,ADVANCE='YES'])

WRITE([UNIT=]u[,[FMT=]fmt][,IOSTAT=ios][,ADVANCE='YES'])
```

Naturalmente, se il file è connesso come `'UNFORMATTED'`, la clausola `FMT=` non dovrà essere presente nelle istruzioni `READ` o `WRITE`. Si noti, tuttavia, che dal momento che i file sequenziali vengono tipicamente utilizzati per il trasferimento di dati verso altri sistemi di elaborazione oppure per la lettura o la scrittura da parte di un operatore umano, essi sono quasi sempre formattati.

Le prossime istruzioni sono validi esempi di istruzioni di I/O formattate operanti su file sequenziali:

```
OPEN(UNIT=15,FILE='file1.txt',FORM='FORMATTED',STATUS='OLD',  &
     ACTION='READ',POSITION='REWIND')
OPEN(UNIT=16,FILE='file2.txt',FORM='FORMATTED',STATUS='NEW',  &
```

```
     ACTION='WRITE',POSITION='REWIND')
OPEN(UNIT=17,FILE='file3.txt',STATUS='OLD',ACTION='READWRITE', &
     POSITION='APPEND')
...
READ(UNIT=15,FMT=100,IOSTAT=stato) a, b, (c(i),i=1,40)
WRITE(UNIT=16,FMT='(2(1X,F7.3))',IOSTAT=stato,ADVANCE='YES') x, y
WRITE(17,*) i, j, k
```

Nel seguito si riportano alcuni esempi tipici di operazioni su file ad accesso sequenziale.

1. Lettura di un file sequenziale di tipo testo:

```
PROGRAM io_demo
  INTEGER   :: i, ios
  CHARACTER(LEN=50) :: line
  OPEN(UNIT=10,FILE="io_demo.f95",STATUS="OLD",           &
       ACTION="READ",FORM="FORMATTED",POSITION="REWIND")
  i=0
  PRINT*, "Testo del programma:"
  DO
     READ(UNIT=10, FMT="(A)",IOSTAT=ios) line
     IF(ios==0) THEN             ! Nessun problema
        i = i+1
        PRINT*, i,">",line,"<"
     ELSE                        ! Problemi in lettura
        EXIT
     END IF
  END DO
END PROGRAM io_demo
```

Si tratta di un esempio molto semplice di programma di gestione di file sequenziali avente come unico scopo quello di stampare a video l'intero testo del codice sorgente, inserendolo in una cornice di simboli > e <. Come si può intuire leggendo l'istruzione di connessione, si assume che il programma in esame sia stato salvato nel file `io_demo.f95`. Naturalmente, affinché l'operazione di lettura possa avere luogo si deve assumere che il file sia già esistente (`STATUS='OLD'`); la clausola `ACTION='READ'` previene la possibilità di scrivere accidentalmente sui record del programma sorgente; infine, la clausola `POSITION='REWIND'` posiziona la testina di lettura sul punto iniziale del file facendo sì che la lettura cominci dal primo record del documento.

Il prossimo esempio, anch'esso particolarmente semplice, serve a contare il numero di caratteri presenti in file di tipo testo. Si può notare un modo elegante di gestire delle condizioni di fine file o di fine record:

```
PROGRAM char_count
```

```
  IMPLICIT NONE
  INTEGER, PARAMETER :: end_of_file = -1
  INTEGER, PARAMETER :: end_of_record = -2
! Att.ne: I valori precedenti sono, in generale,
!  processor-dependent
  CHARACTER(LEN=1) :: c
  INTEGER :: count, ios
! Start
  OPEN(UNIT=11,FILE="miofile.txt",STATUS="OLD",ACTION="READ")
  count = 0
  DO
     READ(UNIT=11,FMT="(A)",ADVANCE="NO",IOSTAT=ios) c
     IF (ios==end_of_record) THEN
        CYCLE
     ELSE IF(ios==end_of_file) THEN
        EXIT
     ELSE
        count = count+1
     END IF
  END DO
  PRINT*, "Il numero di caratteri nell file e': ",count
END PROGRAM char_count
```

Un ulteriore esempio, che gestisce in forma più completa eventuali condizioni di errore, è il seguente:

```
PROGRAM prova_lettura
!
! Scopo: leggere da file gli elementi di un array monodimensionale
!        e stamparne i valori a video
!
  IMPLICIT NONE
  INTEGER , DIMENSION(10) :: a=0
  INTEGER :: IO_stat_number=0
  INTEGER :: i
  OPEN(UNIT=1,FILE='dati.dat',STATUS='OLD',ACTION='READ')
  DO i=1,10
     READ(UNIT=1,FMT=10,IOSTAT=IO_stat_number) a(i)
10   FORMAT(I3)
     IF (IO_stat_number==0) THEN
        CYCLE
     ELSE IF (IO_stat_number==-1) THEN
        PRINT*," End-of-file raggiunto alla linea ",i
        PRINT*," Controllare il file di input! "
```

```
         EXIT
      ELSE IF (IO_stat_number>0 ) THEN
         PRINT*," Dati non validi nel file alla linea ",i
         PRINT*," Controllare il file di input! "
         EXIT
      END IF
   END DO
   DO i=1,10
      PRINT* , " i = ",i," a(i) = ",a(i)
   END DO
END PROGRAM prova_lettura
```

2. Generazione di un file sequenziale di tipo testo:

```
PROGRAM tabulazione
!
! Scopo del programma: tabellare la funzione esterna func(x)
!                      nell'intervallo x1<=x<=x2 con passo dx
!
  IMPLICIT NONE
  INTEGER :: i,n,stato
  REAL :: x,x1,x2,dx

  INTERFACE
     FUNCTION func(x)
        IMPLICIT NONE
        REAL :: func
        REAL, INTENT(IN) :: x
     END FUNCTION func
  END INTERFACE

  x1=0.0   ! estremo inferiore di tabellazione
  x2=5.0   ! estremo superiore di tabellazione
  n=20     ! numeri di punti
  dx=INT(x2-x1)/REAL(n);   ! passo di tabellazione

  OPEN(UNIT=11,FILE='tabella.txt',STATUS='NEW',              &
       ACCESS='SEQUENTIAL',ACTION='WRITE',FORM='FORMATTED',  &
       IOSTAT=stato,POSITION='REWIND')
  x=x1
  DO i=1,n
     WRITE(UNIT=11,FMT=100) x,func(x)
     x=x+dx
  END DO
```

```
100 FORMAT(2(1X,'F7.3,'|'))
  CLOSE(11,STATUS='KEEP')
END PROGRAM tabulazione
!
FUNCTION func(x)
   IMPLICIT NONE
   REAL :: func
   REAL, INTENT(IN) :: x
      func=2.0*sin(x)-1.5
END FUNCTION func
```

3. Riproduzione di un file sequenziale di tipo testo:

```
PROGRAM copia
  IMPLICIT NONE
  REAL :: x,y,z
  INTEGER :: code

  OPEN(UNIT=11,FILE="source.txt",POSITION="REWIND")
  OPEN(UNIT=12,FILE="target.txt",POSITION="REWIND")
  DO
     READ(11,*,IOSTAT=code) x,y,z
     IF(code==0) THEN
        WRITE(12,*) x,y,z
     ELSE IF(code>0)
        WRITE(*,*) "Riscontrata una condizione di errore"
        WRITE(*,*) "Codice dell'errore: ", code
        STOP
     ELSE IF(code<0)
        EXIT
  END DO
  CLOSE(UNIT=11,STATUS='DELETE')
  CLOSE(UNIT=12,STATUS='KEEP')
END PROGRAM copia
```

Il seguente programma rappresenta un esempio un pò più completo di elaborazione di un file di tipo testo:

```
MODULE crittografia
! Il modulo contiene una funzione che restituisce la versione
! crittografata di un carattere c con chiave k, ottenuta secondo
! le seguenti regole:
!    - se c e' un carattere alfabetico maiuscolo, viene restituito il
!      carattere c "avanzato" di k posti, con gestione circolare
```

```
!      dell'alfabeto di 26 lettere. Ad esempio, se k = 2:
!       * c = 'A': viene restituito 'C'
!       * c = 'Z': viene restituito 'B'
!    - se c e' non un carattere alfabetico maiuscolo, viene restituito
!      c non modificato

CONTAINS
   FUNCTION traduci(c,k)
! *** Sezione Dichiarativa
       IMPLICIT NONE
!  ** Dichiarazione argomenti fittizi
       CHARACTER(1), INTENT(IN) :: c
       INTEGER, INTENT(IN) :: k
!  ** DICHIARAZIONE TIPO FUNZIONE
       CHARACTER(1) :: traduci
! *** Sezione Esecutiva
          IF (c<'A'.OR.c>'Z') THEN
             traduci = c
          ELSE IF (IACHAR(c)+k <= IACHAR('Z')) THEN
             traduci = CHAR(IACHAR(c)+k)
          ELSE
             traduci = CHAR(IACHAR(c)+k-26)
          END IF
       RETURN
   END FUNCTION traduci
END MODULE crittografia

PROGRAM cesare
! Questo programma effettua la crittografia di un file di testo secondo
! il "metodo crittografico di Giulio Cesare", in cui la chiave e' un
! numero intero k compreso fra 1 e 26.
! La codifica consiste nel fare "avanzare" le lettere del messaggio
! di k posti, con gestione circolare dell'alfabeto.
!  ESEMPIO:
!    INPUT: a. testo: "ATTACCARE DOMANI. FIRMATO: GIULIO CESARE"
!           b. k = 2
!    OUTPUT:   testo: "CVVCEECTG FQOCPK. HKTOCVQ: IKWNKQ EGUCTG"
!  ASSUNZIONE: il testo in input e' composto solamente di caratteri
!              maiuscoli, spazi e simboli di interpunzione (questi ultimi
!              non vengono tradotti).
! *** SEZIONE DICHIARATIVA
  USE crittografia
  IMPLICIT NONE
```

```
  INTEGER :: key                              ! valore della chiave
  CHARACTER(12) :: nome_input, nome_output ! nomi dei file
  INTEGER, PARAMETER :: lunghezza_linea = 240
                                ! massima lunghezza ammessa per un record
  CHARACTER(lunghezza_linea) :: linea
                                ! la linea letta di volta in volta
  INTEGER :: istatus            ! stato I/O
  INTEGER :: i                  ! indice per ciclo interno
! *** SEZIONE ESECUTIVA
  WRITE(*,*) 'Questo programma legge un file di testo, lo traduce secondo'
  WRITE(*,*) 'il metodo di Giulio Cesare, e lo stampa su un altro file'

  WRITE(*,100,ADVANCE='NO') 'Nome del file di input (da tradurre): '
  READ (*,100) nome_input
  WRITE(*,100,ADVANCE='NO') 'Nome del file di output (tradotto): '
  READ (*,100) nome_output
  WRITE(*,100,ADVANCE='NO') 'Chiave (intero positivo < 26): '
  READ (*,100) key

  OPEN (UNIT=9,FILE=nome_input,STATUS='OLD',IOSTAT=istatus,ACTION='READ')
! apertura del file sorgente in lettura
  IF (istatus==0) THEN
     OPEN (UNIT=10,FILE=nome_output,STATUS='REPLACE',IOSTAT=istatus  &
           ACTION='WRITE')
! apertura del file destinazione in scrittura
     lettura_riga: DO
        READ (9,100,IOSTAT=istatus) linea  ! Lettura di una linea da file
        IF (istatus/=0) EXIT
        elaborazione_carattere: DO i=1,LEN_TRIM(linea)
           linea(i:i) = traduci(linea(i:i),key)
        END DO elaborazione_carattere
        WRITE(10,*) linea(1:LEN_TRIM(linea))
     END DO lettura_riga
     WRITE(*,*) 'Fine traduzione'
     CLOSE(9)
     CLOSE(10)
  ELSE
     WRITE(*,*) 'Il file ', nome_input, 'non esiste'
  END IF

  STOP
100 FORMAT(A)
END PROGRAM cesare
```

8.3.1 I/O *non advancing*

Quando un'istruzione di I/O advancing viene eseguita, l'operazione di lettura o di scrittura comincia sempre dal record successivo a quello corrente e lascia il file posizionato alla fine del reord processato. Inoltre, nelle operazioni di ingresso, la lista di input non deve specificare più caratteri (o valori) di quanti stabiliti con la eventuale specificazione della lunghezza del record. Nel caso, invece, in cui, per file formattati, il numero di caratteri prodotti in input sono inferiori alla specificazione della lunghezza del record, la restante parte del record viene completata con spazi bianchi a meno che nella lista di `OPEN` non sia stato inserito lo specificatore `PAD='NO'`, nel qual caso il comportamento adottato può variare da processore a processore.

Le operazioni di I/O formattate *non advancing* danno la possibilità di leggere o scrivere anche solo una parte di un record, lasciando il file posizionato dopo l'ultimo carattere letto o scritto piuttosto che saltare alla fine del record. Una peculiarità delle istruzioni di input *non advancing* è la possibilità di leggere record di lunghezza variabile e di determinarne la lunghezza. Si noti che le istruzioni di I/O *non advancing* possono essere utilizzate soltanto con file esterni sequenziali formattati e con formato esplicitamente dichiarato (in altre parole non è possibile avere istruzioni di I/O *non advancing* con formato guidato da lista o con meccanismo *namelist*).

La sintassi per le operazioni di connessione e di lettura/scrittura di un file ad accesso sequenziale e formattato di tipo *non advancing* è le seguente:

```
OPEN([UNIT=]u,FILE=fln[,FORM='FORMATTED'],[ACCESS='SEQUENTIAL'] &
     [,RECL=rl][,IOSTAT=ios][,STATUS=st][,ACTION=act] &
     [,DELIM=del][,PAD=pad][,POSITION=pos])

READ([UNIT=]u,[FMT=]fmt[,IOSTAT=ios],ADVANCE='NO')

WRITE([UNIT=]u,[FMT=]fmt[,IOSTAT=ios][,SIZE=size],ADVANCE='NO')
```

dove alla variabile *size*, quando è presente il relativo specificatore, è assegnato come valore il numero dei caratteri letti in input mentre l'espressione *fmt* può essere l'etichetta di una istruzione `FORMAT` o una espressione stringa specificante una formato contestuale, ma *non* può essere un asterisco.

Come esempio di istruzione di I/O formattata *non advancing* si può considerare il seguente frammento di programma:

```
INTEGER, DIMENSION(9) :: vett=(/1,2,3,0,0,9,8,8,6/)
...
DO i=1,3
   WRITE(11,'(I1)',ADVANCE='NO') vet(i)
END DO
WRITE(11,'(A1)',ADVANCE='NO') "-"
DO i=4,6
   WRITE(11,'(I1)',ADVANCE='NO') vet(i)
END DO
```

```
WRITE(11,'(A1)',ADVANCE='NO') "-"
DO i=7,9
   WRITE(11,'(I1)',ADVANCE='NO') vet(i)
END DO
WRITE(11,'(A1)',ADVANCE='NO') "-"
```

il quale produce, sul file connesso all'unità logica `11`, il record `123-00-9886`.

8.3.2 I/O con meccanismo *namelist*

Le istruzioni di I/O con meccanismo *namelist* servono a trasferire, con la medesima istruzione, un intero gruppo di variabili senza l'obbligo di curare il formato e la modalità di presentazione. Affinché ciò sia possibile è necessario che le suddette variabili siano specificate in una opportuna lista dichiarate con una istruzione `NAMELIST`.

Le operazioni di I/O con meccanismo *namelist* sono utili per inizializzare lo stesso gruppo di variabili in sessioni successive di uno stesso run o anche per cambiare, in maniera selettiva, il valore di alcune variabili a cui sia stato già assegnato un valore iniziale.

I dettagli relativi all'istruzione `NAMELIST` sono stati già discussi nel capitolo 4 dove è stato illustrato anche il funzionamento delle istruzioni di lettura e scrittura con meccanismo *namelist* relativi ai dispositivi di I/O standard. Tutto quanto detto nel capitolo 4 si applica anche in relazione alla gestione dei file esterni. Può risultare utile, tuttavia, riportare la sintassi delle operazioni di connessione e di lettura/scrittura di un file ad accesso sequenziale per operazioni di I/O di tipo *namelist*:

```
OPEN([UNIT=]u,FILE=fln [,ACCESS='SEQUENTIAL'][,IOSTAT=ios ] &
     [,STATUS=st ][,ACTION=act ][,DELIM=del ][,PAD=pad ] &
     [,POSITION=pos ])

READ([UNIT=]u,[NML=] list [,IOSTAT=ios ])

WRITE([UNIT=]u,[NML=] list [,IOSTAT=ios ])
```

Si noti l'assenza, nelle istruzioni `READ` e `WRITE`, della clausola relativa al formato (`FORMAT=`) e quella relativa all'avanzamento (`ADVANCE`), entrambe incompatibili con lo specificatore `NML`.

Tanto per fissare le idee, si può considerare il seguente esempio:

```
PROGRAM prova_namelist
  IMPLICIT NONE
  INTEGER :: m, n
  TYPE persona
     INTEGER :: matr
     INTEGER :: level
     CHARACTER(20) :: name
  END TYPE persona
  TYPE(persona) :: impiegato
```

```
  NAMELIST/lista1/m,n,impiegato
! ...
  impiegato = persona(20,4,'John Smith')
! ...
  OPEN(UNIT=7,FILE='dati.dat')
  READ(UNIT=7,NML=lista1)
! ...
END PROGRAM prova_namelist
```

In questo breve programma di prova vengono dichiarate due variabili intere, `m` ed `n` ed una *struttura*, `impiegato` e tutte e tre vengono inserite nella lista con nome `lista1`. L'oggetto di tipo derivato, `impiegato`, viene inizializzato, dopodiché all'unità logica `7` viene connesso il file `dati.dat` nel quale vengono letti i valori di alcune (o tutte) le variabili della lista. Un valido esempio per il file `dati.dat` potrebbe essere:

```
&LISTA1
m = 3
n = 12
smith%level = 6
/
```

nel qual caso, della struttura `impiegato`, soltanto il valore della componente `level` verrebbe aggiornato mentre le componenti `matr` e `name` resterebbero inalterate.

Una utile innovazione apportata dal Fortran 95 riguarda la possibilità di documentare linea per linea i record di input per `NAMELIST` con opportuni commenti. Questi commenti vanno inseriti così come si farebbe nel codice sorgente, ossia come stringhe precedute da un punto esclamativo. Pertanto, ritornando all'esempio precedente, un file di input meglio documentato e più autoesplicativo potrebbe essere:

```
&LISTA1
m = 3   ! numero di figli
n = 12  ! anni di servizio
smith%level = 6
/
```

8.4 Gestione dei file ad accesso diretto

Esistono molte situazioni, come l'interrogazione di un database, in cui risulta più efficiente effettuare operazioni di lettura/scrittura di record in maniera non sequenziale ma casuale. Ciò può ottenersi connettendo il file di lavoro ad accesso diretto, ossia specificando la clausola `ACCESS='DIRECT'` nella lista degli specificatori di `OPEN`.

I record in un file connesso ad accesso diretto si caratterizzano per il fatto che possono essere letti e scritti in qualsiasi ordine. Tuttavia, un vincolo che deve essere rispettato consiste nel fatto che i record del file devono avere tutti la stessa lunghezza specificata dalla clausola

RECL=*rec* nell'istruzione OPEN. Si ricorda che la lunghezza di un record è misurata in numero di caratteri, per file formattati, o in unità dipendenti dal processore, per file non formattati. Tuttavia, dal momento che un file generato ad accesso diretto non può normalmente essere trasferito da una macchina all'altra, non c'è ragione di effettuare una conversione dei valori in esso immagazzinati dalla modalità di rappresentazione interna alla modalità testo. Ne discende che i file ad accesso diretto sono, nella stragrande maggioranza dei casi, *non formattati*. Si perviene in tal modo a forme più compatte di registrazione e ad operazioni più veloci in quanto si evitano le operazioni di conversione.

Per accedere ad un record su di un file ad accesso diretto occorre conoscerne il *numero d'ordine* che contraddistingue ciascun record. Il numero d'ordine del record da processare deve essere specificato nella lista di controllo della frase di I/O mediante la specificazione REC=*nr*.

Se un file è connesso per accesso diretto, per esso non è possibile effettuare operazioni di I/O *non advancing*, con formato guidato da lista o con meccanismo *namelist*. Ne discende che la sintassi per le operazioni di connessione e di lettura/scrittura è le seguente:

```
OPEN([UNIT=]u,FILE=fln[,FORM=fm],ACCESS='DIRECT',RECL=rl &
     [,IOSTAT=ios][,STATUS=st][,ACTION=act][,DELIM=del][,PAD=pad])

READ([UNIT=]u[,[FMT=]fmt],REC=rn[,IOSTAT=ios])

WRITE([UNIT=]u[,[FMT=]fmt],REC=rn[,IOSTAT=ios])
```

Contrariamente ai file ad accesso sequenziale, i file ad accesso diretto consentono un *aggiornamento* dei loro record nel senso che è possibile modificare un certo record lasciando inalterati tutti i precedenti ed i successivi. Tuttavia, sebbene sia possibile riscrivere in maniera selettiva un record del file, *non* è possibile eliminarlo dal file (al più è possibile sovrascriverlo con una serie di *blank*).

Per la gestione dei file ad accesso diretto è doveroso, infine, rammentare le seguenti regole:

- Un qualsiasi tentativo di leggere un record di un file connesso ad accesso diretto che non sia stato precedentemente scritto fa sì che tutte le entità specificate nella lista di input diventino indefinite.

- Se la specificazione di formato (per file formattati) impone l'avanzamento di un record (ad esempio attraverso l'utilizzo del descrittore *slash*) il numero di record è incrementato di un'unità sicché la successiva operazione di lettura/scrittura interesserà il record con numero d'ordine successivo a quello attuale.

- Nelle operazioni di uscita, la lista di output non deve specificare più valori (o caratteri, a seconda che il file sia non formattato oppure formattato) di quanti siano compatibili con la specificazione della lunghezza del record. Nel caso, invece, in cui il numero di valori (o caratteri) prodotti in input siano inferiori alla specificazione della lunghezza del record, la restante parte del record resta indefinita (per file non formattati) oppure completata con spazi bianchi (per file formattati).

- Nelle operazioni di ingresso, la lista di input non deve specificare più valori (o caratteri) di quanti stabiliti dalla specificazione della lunghezza del record. Nel caso, invece, in cui (per file formattati) il numero di caratteri prodotti in input siano inferiori alla specificazione della lunghezza del record, la restante parte del record viene completata con spazi bianchi a meno che non sia stato inserito lo specificatore `PAD='NO'` nella lista di `OPEN`, nel qual caso il comportamento adottato può variare da processore a processore.

Si riportano di seguito, in forma schematica, alcuni esempi di operazioni tipiche su file ad accesso diretto:

1. Generazione di un file ad accesso diretto di tipo non formattato:

```
PROGRAM genera_data_base
  TYPE studente
     CHARACTER(LEN=10) :: nome
     CHARACTER(LEN=10) :: cognome
     INTEGER :: media_voti
  END TYPE studente
  TYPE(studente) :: elem
  INTEGER :: matr   ! indirizzo del record da inserire
  LOGICAL :: finito=.FALSE.
  CHARACTER(LEN=1) :: risp
  OPEN(UNIT=11,FILE='pagella',STATUS='NEW',ACCESS='DIRECT', &
       FORM='UNFORMATTED',RECL=150)
  DO WHILE(.NOT.finito)
! Acquisizione dell'indirizzo del record da inserire...
     READ(*,*) matricola
! ... e del valore del record fa scrivere
     WRITE(*,100,ADVANCE='NO') "Nome: "; READ(*,*) elem%nome
     WRITE(*,100,ADVANCE='NO') "Cognome: "; READ(*,*) elem%cognome
     WRITE(*,100,ADVANCE='NO') "Media: "; READ(*,*) elem%media_voti
100  FORMAT(1X,A)
! Scrittura del record
     WRITE(UNIT=11,REC=matricola) elem%nome,elem%cognome,elem%media_voti
! Aggiornamento della condizione logica
     WRITE(*,100,ADVANCE='NO') "Finito? (S/N): "; READ(*,*) risp
     IF(risp=='S'.OR.rips=='s') finito=.TRUE.
  END DO
  CLOSE(UNIT=11,STATUS='KEEP')
END PROGRAM genera_data_base
```

2. Ricerca di un record per indirizzo in un file ad accesso diretto:

```
COMPLEX :: rad   ! elemento del file da ispezionare
INTEGER :: pos   ! indirizzo del record da trovare
```

```
OPEN(UNIT=12,FILE='archivio',STATUS='OLD',ACCESS='DIRECT', &
     FORM='UNFORMATTED',RECL=120)
! Acquisizione dell'indirizzo del record da leggere
READ(*,*) pos
! Lettura del record
READ(UNIT=12,REC=pos) rad
! Elaborazione del record letto
...
CLOSE(UNIT=12,STATUS='KEEP')
```

3. Aggiornamento di un file ad accesso diretto:

```
CHARACTER(LEN=10) :: campo1
REAL :: campo2, campo3
INTEGER :: chiave
OPEN(UNIT=13,FILE='esame',STATUS='OLD',ACCESS='DIRECT', &
     FORMAT='UNFORMATTED',RECL=150)
! Acquisizione dell'indirizzo del record da aggiornare...
READ(*,*) chiave
! ... e del valore del record da scrivere
READ(*,*) campo1
READ(*,*) campo2
! Aggiornamento del record
WRITE(UNIT=13,REC=matricola) campo1,campo2,campo3
CLOSE(UNIT=13,STATUS='KEEP')
```

E' doveroso, in conclusione, far notare che *i file ad accesso diretto non formattati aventi record la cui lunghezza sia un multiplo intero della dimensione del settore di un particolare computer rappresentano i file Fortran più efficienti per quel computer*. Quanto asserito è facilmente comprensibile se si pensa che:

- Essendo un file ad accesso diretto è possibile processarne un qualunque record direttamente.
- Essendo non formattato non saranno dedicate risorse di macchina alla conversione da o verso un formato leggibile dall'utente umano.
- Poiché ciascun record ha una lunghezza proprio pari ad un settore del disco, ogni volta che un record debba essere processato sarà necessario leggere o scrivere soltanto un settore. E' chiaro che record più grandi o più piccoli la cui lunghezza non sia multipla della dimensione del settore costringerebbero la testina di lettura/scrittura a muoversi a cavallo di due o più settori per analizzare un unico record.

Dal momento che questi file sono così efficienti, molti grossi programmi Fortran sono sviluppati per farne abbondante uso.

Al fine di confrontare l'efficienza delle operazioni di gestione di file formattati e non formattati si consideri il seguente programma il quale crea due file contenenti ciascuno 5000 record con quattro valori reali in doppia precisione per linea, dei quali però uno solo è di tipo formattato mentre l'altro è di tipo non formattato. Una volta generati in maniera random i record dei due file, essi vengono letti all'interno di due cicli separati e viene misurato il tempo speso dal programma per effettuare questa operazione.

```
PROGRAM accesso_diretto
  IMPLICIT NONE
  INTEGER,PARAMETER :: single=SELECTED_REAL_KIND(p=6)
  INTEGER,PARAMETER :: double=SELECTED_REAL_KIND(p=14)
  INTEGER,PARAMETER :: max_records = 50000      ! max numero di record
  INTEGER,PARAMETER :: number_of_reads = 100000 ! numero di record letti
  INTEGER :: i
  INTEGER :: length_fmt = 80 ! lunghezza di ciascun record formattato
  INTEGER :: length_unf      ! lunghezza di ciascun record non formattato
  INTEGER :: irec            ! puntatore al record
  REAL(KIND=single) :: value
  REAL(KIND=single) :: time_fmt  ! tempo per lettura formattata
  REAL(KIND=single) :: time_unf  ! tempo per lettura non formattata
  REAL(KIND=single) :: time_begin, time_end
  REAL(KIND=double),DIMENSION(4) :: values  ! valori del record

! Valutazione della lunghezza di ciascun record nel file non formattato
  INQUIRE(IOLENGTH=length_unf) values
  WRITE(*,"(A,I2)") "Lunghezza dei record non formattati: ", &
    length_unf
  WRITE(*,"(A,I2)") "Lunghezza dei record non formattati: ", &
    length_fmt
! Connessione dei file
  OPEN(UNIT=11,FILE="C:/MYPROG/OUT/myfile.fmt",ACCESS="DIRECT",  &
       FORM="UNFORMATTED",STATUS="REPLACE",ACTION="READWRITE",   &
       RECL=length_unf)
  OPEN(UNIT=12,FILE="C:/MYPROG/OUT/myfile.unf",ACCESS="DIRECT", &
       FORM="FORMATTED",STATUS="REPLACE",ACTION="READWRITE",    &
       RECL=length_fmt)
! Generazione dei record e loro inserimento nei rispettivi file
  DO i=1,max_records
     CALL RANDOM_NUMBER(values)
     WRITE(11,REC=i) values
     WRITE(12,"(4ES20.14)",REC=i) values
  END DO
! Processing dei record non formattati
  CALL CPU_TIME(time_begin)
```

```
  DO i=1,number_of_reads
     CALL RANDOM_NUMBER(values)
     irec = (max_records-1)*value+1
     READ(11,REC=irec) values
  END DO
  CALL CPU_TIME(time_end)
  time_unf = time_end-time_begin
! Processing dei record formattati
  CALL CPU_TIME(time_begin)
  DO i = 1, number_of_reads
     CALL RANDOM_NUMBER(value)
     irec = INT((max_records-1)*value)+1
     READ(12,"(4ES20.14)",REC=irec) values
  END DO
  CALL CPU_TIME(time_end)
  time_fmt = time_end-time_begin
! Output dei risultati
  WRITE(*,*)
  WRITE(*,"(A,F6.2)") "Tempo speso sul file non formattato: ", time_unf
  WRITE(*,"(A,F6.2)") "Tempo speso sul file formattato:     ", time_fmt
  STOP
END PROGRAM accesso_diretto
```

Compilato con l'Intel Fortran Compiler 5.0 ed eseguito su un PC Pentium III, il precedente programma produce questo output:

```
Lunghezza dei record non formattati: 32
Lunghezza dei record non formattati: 80

Tempo speso sul file non formattato:   0.94
Tempo speso sul file formattato:       1.97
```

Come si può notare le operazioni svolte sui file non formattate sono state notevolmente più rapide. Non solo; se infatti si vanno ad analizzare i file prodotti in uscita si nota subito un'altra importante caratteristica: i file non formattati occupano anche meno spazio su disco. Infatti questo è il contenuto della directory di uscita dopo l'esecuzione del file:

```
C:\MYPROG\OUT>dir
 Il volume nell'unità C non ha etichetta.
 Numero di serie del volume: E85E-5D71

 Directory di C:\MYPROG\OUT

16/12/2001  18.27       <DIR>          .
16/12/2001  18.27       <DIR>          ..
```

```
16/12/2001  18.27            1'600'000 myfile.fmt
16/12/2001  18.27            4'000'000 myfile.unf
               2 File      5'600'000 byte
               2 Directory  12'105'617'408 byte disponibili
```

Tuttavia, se da un lato i file ad accesso diretto non formattati risultano sempre più compatti e veloci dei corrispondenti file formattati, è innegabile che hanno il grosso inconveniente di essere *system-dependent* e, dunque, non portabili fra processori differenti.

8.5 File interni

I file finora trattati sono tutti "esterni", nel senso che sopravvivono all'esterno dell'ambiente Fortran. Ciò che poteva essere "interno" era il solo nome del file (l'unità logica) che doveva essere associato al nome esterno (noto al sistema operativo) attraverso l'operazione di connessione.

Esiste, tuttavia, in Fortran la possibilità di definire un file interno, anch'esso individuato da una sua unità logica, il quale può essere considerato come una zona di memoria di transito atta alla conversione di dati interni. Un file interno coincide con una variabile (o con un array di variabili) di tipo stringa di caratteri. Un file interno è *sempre* di tipo di testo (ossia *formattato*) e può essere operato unicamente in maniera *sequenziale*. Non esistendo, per questo tipo di file, la controparte "esterna", per essi non è possibile applicare l'operazione di connessione, il riferimento al file avvenendo direttamente nella frase di I/O.

Se un tale file è una variabile stringa (o una sua sottostringa) oppure è un elemento di array di tipo `CHARACTER` allora il file è costituito da un solo record; se, invece, esso un array di stringhe esso è costituito da tanti record quanti sono gli elementi della variabile. Si noti che una qualsiasi operazione di I/O su un file interno inizia sempre dal primo record.

L'impiego tipico dei file interni è quello di applicare ad una variabile di tipo stringa, trattata alla stessa stregua di un file, gli algoritmi di conversione dei dati tipici delle operazioni di I/O. In altri termini, una stringa di caratteri che rappresenti il valore di un dato (così come avviene sui supporti di I/O) può essere trasformata in un valore numerico intero o reale e come tale memorizzata in un'altra variabile secondo la forma di rappresentazione interna per essa prevista.

Utilizzando il meccanismo in esame, un qualsiasi record di input può essere letto con il formato `A` e registrato, sottoforma di stringa, in una variabile successivamente riguardata come file interno. Il valore di questa variabile stringa può essere successivamente esaminata sulla base di una lista di descrittori di formato allo scopo di esaminare i diversi campi da cui essa è costituita ed associare, a ciascun campo, la forma di rappresentazione interna desiderata.

Un esempio di utilizzo di file interno è rappresentato dal seguente frammento di programma:

```
CHARACTER(LEN=10) :: string
REAL :: pigreco
...
string = "   3.14   "
...
READ(string,*) pigreco
```

il quale opera la conversione di una stringa di caratteri in un reale. In sintesi, l'esempio in esame prevede la lettura del valore della stringa `string`, valore che viene rappresentato nella forma interna di rappresentazione dei reali ed associato alla variabile `pigreco`. Il prossimo esempio, invece, produce la conversione del valore della variabile intera `ind` in una stringa di cifre decimali e la sua successiva registrazione nella variabile `str`.

```
CHARACTER(LEN=10) :: str
INTEGER :: ind
...
ind = 31
READ(str,'(I3)') ind
```

Un esempio appena più complesso, che fa uso di un file interno multirecord, è il seguente:

```
PROGRAM prova_file_interni
   CHARACTER(5),DIMENSION(3) :: str
   INTEGER, DIMENSION(3,3) :: vet
   CHARACTER(12) :: fname
! ...
   str = (/"1 2 3","4 5 6","7 8 9"/)
! ...
! La prossima list-directed READ esegue l'assegnazione:
!       |1 4 7|
!   vet=|2 5 8|
!       |3 6 9|
   READ(str,*) vet
! Il prossimo formatted WRITE pone fname='FM003.DAT'.
   WRITE (fname,200) vet(3,1)
200 FORMAT ('FM',I3.3,'.DAT')
! Viene connesso il file FM003.DAT per successive
! operazioni di I/O
   OPEN(UNIT=11,FILE=fname,STATUS='NEW')
!   ...
END PROGRAM prova_file_interni
```

Il programma che segue, infine, rappresenta un ulteriore esempio di utilizzo dei file interni. Le operazioni eseguite sono assai semplici. Dapprima viene letta una stringa di cifre intervallate dal simbolo di dollaro. Quindi, ciascun carattere `$` viene sostituito da uno spazio bianco, affinché la successiva operazione di lettura coinvolga soltanto cifre e punti decimali di modo che sia possibile assegnare, ordinatamente, i valori letti agli elementi di un array di tipo `REAL`.

```
PROGRAM internal_file
  IMPLICIT NONE
  INTEGER :: k, error
  REAL, DIMENSION(10) :: money
```

```
  CHARACTER(LEN=80) :: cdata
  INTEGER, PARAMETER :: in=20
  OPEN(UNIT=in,FILE='accounts',STATUS='OLD',IOSTAT=error)
  IF(error /= 0) THEN
    PRINT*, "Errore durante l'apertura del file"
    STOP
  END IF
  READ(UNIT=in,FMT='(A)') cdata
! Stampa il file interno "cdata" per mostrare come appare
  PRINT*, cdata
! Sostituisce tutti i caratteri "$" con uno spazio bianco
  DO k=1,80
     IF(cdata(k:k)=='$')THEN
       cdata(k:k)=' '
     END IF
  END DO
! Ora assegna i valori letti come reali agli elementi dell'array money
  READ(UNIT=cdata,FMT='(10(1X,F4.2))')money
! Stampa l'array money per mostrare come ha lavorato
  PRINT*, money
END PROGRAM internal_file
```

Supponendo, ad esempio, che il file esterno `accounts` contenga la stringa:

```
1.52$3.50$2.11$5.25$4.02$1.25$3.50$2.04$5.55$1.30
```

l'output del programma sarebbe:

```
1.520000       3.500000       2.110000       5.250000       4.020000
1.250000       3.500000       2.040000       5.550000       1.300000
```

Appendice A

Insieme di codifica ASCII

La seguente tabella riporta l'ordinamento lessicografico noto come *codice* ASCII. I caratteri corrispondenti ai valori da 0 a 31 ed a 127 sono caratteri di controllo non riproducibili a stampa. Il carattere corrispondente al valore 32 rappresenta uno spazio vuoto.

32		56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g		
56	8	80	P	104	h		

Appendice B

Precedenza degli operatori

La *priorità* degli operatori indica l'ordine di calcolo delle operazioni in un'espressione. La seguente tabella riporta l'elenco di tutti i possibili operatori Fortran secondo l'ordine di priorità decrescente:

Operatori	**Precedenza**
Operatori user-defined unari	più alta
`**`	↓
`*`, `/`	
`+`, `-` (*unari*)	
`+`, `-` (*binari*)	
`//`	
`==`, `/=`, `<`, `<=`, `>`, `>=`	
`.NOT.`	
`.AND.`	
`.OR.`	
`.EQV.`, `.NEQV.`	↓
Operatori user-defined binari	più bassa

L'*associatività* degli operatori indica, invece, come vengono gestite le operazioni aventi la stessa priorità. Si ricorda che in Fortran l'associatività degli operatori va da sinistra verso destra, con l'unica eccezione dell'operatore `**` che, invece, agisce da destra verso sinistra. Così, per esempio:

```
A**B**C   equivale a:  A**(B**C)
 A+B-C    equivale a:   (A+B)-C
```

E' importante tene re a mente l'associatività degli operatori specialmente quando si opera con valori ai limiti della rappresentazione di macchina. Ad esempio, considerate le esepressioni:

```
A-B+C
A+C-B
```

nel caso in cui `A` fosse il numero più grande rappresentabile in macchina e `C` un valore positivo minore di `B`, mentre il calcolo della prima espressione non darebbe problemi, il compuito della seconda causerebbe il crash del programma per errore di overflow.

L'ordine in cui vengono applicati gli operatori può, comunque, essere modificato attraverso l'impiego di parentesi; il contenuto delle parentesi, infatti, viene sempre valutato prima dell'applicazione degli operatori esterni.

Si noti che due operatori non possono mai essere adiacenti. Così, ad esempio, l'espressione x^{-1} non può essere scritta come:

```
x**-1
```

ma gli operatori di elevamento a potenza ed il segno meno unario dovranno essere separati da opportune parentesi:

```
x**(-1)
```

Appendice C

Ordinamento delle istruzioni Fortran

Un qualunque programma Fortran è sempre formato da una o più unità di programma, ciascuna delle quali composta da almeno due istruzioni (ossia almeno l'istruzione iniziale e l'istruzione `END`). In ogni caso l'unità principale (*main*) deve essere obbligatoriamente presente in ogni programma e deve essere unica all'interno del programma stesso. Per quanto concerne, invece, le istruzioni di cui ciascuna unità si compone, come è noto esse possono essere distinte, a seconda del compito a cui sono destinate, in *eseguibili* e *non eseguibili*. Ciascuna di esse ha un suo proprio "campo di applicabilità" e, a suo modo, una sua "propedeuticità". A tal riguardo la tabella C.1 illustra l'ordine con cui le istruzioni possono apparire all'interno di un'unità di programma. Le linee orizzontali delimitano i campi di applicabilità delle istruzioni che non possono essere combinate insieme; le linee verticali, invece, separano istruzioni che possono essere combinate. Come si può notare da quanto riportato in tabella, le istruzioni non eseguibili tipicamente precedono quelle eseguibili all'interno di un'unità di programma, eccezion fatta per le istruzioni `FORMAT` le quali, invece, possono essere efficacemente "mescolate" a quelle eseguibili.

Tabella C.1: Ordinamento delle istruzioni Fortran

<table>
<tr><td colspan="2">Istruzione <code>PROGRAM</code>, <code>FUNCTION</code> o <code>SUBROUTINE</code></td></tr>
<tr><td colspan="2">Istruzioni <code>USE</code></td></tr>
<tr><td colspan="2">Istruzione <code>IMPLICIT NONE</code></td></tr>
<tr><td rowspan="2">Istruzioni <code>FORMAT</code></td><td>Definizione di tipi derivati
Blocchi di interfaccia
Istruzioni di dichiarazione di tipo
Istruzioni di specificazione
Dichiarazioni di funzioni</td></tr>
<tr><td>Istruzioni e costrutti eseguibili</td></tr>
<tr><td colspan="2">Istruzione <code>CONTAINS</code></td></tr>
<tr><td colspan="2">Procedure interne o di modulo</td></tr>
<tr><td colspan="2">Istruzione <code>END</code></td></tr>
</table>

Tabella C.2: Istruzioni ammesse nelle unità di programma

Unità di programma	**Programma principale**	**Modulo**	**Procedura esterna**	**Procedura di modulo**	**Procedura interna**	**Blocco di interfaccia**
Istruzione `USE`	Si	Si	Si	Si	Si	Si
Istruzione `FORMAT`	Si	No	Si	Si	Si	No
Dichiarazioni varie	Si	Si	Si	Si	Si	Si
Definizione di tipi derivati	Si	Si	Si	Si	Si	Si
Blocco di interfaccia	Si	Si	Si	Si	Si	Si
Istruzione eseguibile	Si	No	Si	Si	Si	No
Istruzione `CONTAINS`	Si	Si	Si	Si	No	No
Dichiarazioni di funzioni	Si	No	Si	Si	Si	No

Oltre alle suddette limitazioni, non tutte le istruzioni Fortran possono apparire in una qualunque unità di programma. La tabella C.2 illustra quali tipi di istruzione possono figurare nelle varie unità di programma e quali, invece, sono proibite.

Appendice D

Applicabilità degli specificatori alle istruzioni di I/O

Nella seguente tabella sono riportati, per ciascuno specificatore, l'istruzione alla quale è applicabile e se vi compare come parametro di ingresso ("I") o di uscita ("U") e per quale metodo di accesso e per quale tipo di formattazione del file lo specificatore è applicabile.

Tabella D.1: Applicabilità degli specificatori di I/O

	OPEN	CLOSE	WRITE READ	BACKSPACE ENDFILE REWIND	INQUIRE	*metodo di accesso*	*tipo del file*
UNIT	I	I	I	I	I	qualsiasi	qualsiasi
FILE	I	-	-	-	I	qualsiasi	qualsiasi
IOSTAT	U	U	U	U	U	qualsiasi	qualsiasi
STATUS	I	I	-	-	-	qualsiasi	qualsiasi
FMT	-	-	I	-	-	qualsiasi	FORMATTED
ACCESS	I	-	-	-	U	qualsiasi	qualsiasi
DELIM	I	-	-	-	U	qualsiasi	qualsiasi
PAD	I	-	-	-	U	qualsiasi	qualsiasi
ACTION	I	-	-	-	U	qualsiasi	qualsiasi
FORM	I	-	-	-	U	qualsiasi	qualsiasi
RECL	I	-	-	-	U	DIRECT	qualsiasi
POSITION	I	-	-	-	U	qualsiasi	qualsiasi
REC	-	-	I	-	-	DIRECT	qualsiasi
ADVANCE	-	-	I	-	-	SEQUENTIAL	FORMATTED
NML	-	-	I	-	-	SEQUENTIAL	FORMATTED
SIZE	-	-	U (solo READ)	-	-	SEQUENTIAL	FORMATTED
NEXTREC	-	-	-	-	U	DIRECT	qualsiasi
EXIST	-	-	-	-	U	qualsiasi	qualsiasi
OPENED	-	-	-	-	U	qualsiasi	qualsiasi
NUMBER	-	-	-	-	U	qualsiasi	qualsiasi

	OPEN	CLOSE	WRITE READ	BACKSPACE ENDFILE REWIND	INQUIRE	*metodo di accesso*	*tipo del file*
NAMED	-	-	-	-	U	qualsiasi	qualsiasi
NAME	-	-	-	-	U	qualsiasi	qualsiasi
SEQUENTIAL	-	-	-	-	U	qualsiasi	qualsiasi
DIRECT	-	-	-	-	U	qualsiasi	qualsiasi
FORMATTED	-	-	-	-	U	qualsiasi	qualsiasi
UNFORMATTED	-	-	-	-	U	qualsiasi	qualsiasi
READ	-	-	-	-	U	qualsiasi	qualsiasi
WRITE	-	-	-	-	U	qualsiasi	qualsiasi
READWRITE	-	-	-	-	U	qualsiasi	qualsiasi
IOLENGTH	-	-	-	-	U	qualsiasi	qualsiasi

Appendice E

Procedure intrinseche

La tabella E.1 riporta un elenco alfabetico delle procedure intrinseche previste dal linguaggio Fortran. I dati riportati nelle diverse colonne sono da interpretarsi secondo quanto di seguito descritto:

- La prima colonna riporta il *nome generico* di ogni procedura e la sua sequenza di chiamata (essendo quest'ultima la lista delle *parole chiave* associate a ciascun argomento). Gli argomenti rappresentati fra parentesi quadre sono da ritenersi facoltativi. Anche l'uso delle parole chiave è facoltativo, tuttavia queste devono essere obbligatoriamente specificate quando uno degli argomenti facoltativi manca nella sequenza di chiamata o se gli argomenti non vengono specificati nell'ordine standard. Ad esempio, la funzione intrinseca `SIN` ha un solo argomento e la parola chiave dell'argomento è `X` sicché questa procedura può essere invocata indifferentemente con o senza la parola chiave, vale a dire che le seguenti istruzioni sono perfettamente equivalenti:

  ```
  senalfa = SIN(X=alfa)
  senalfa = SIN(alfa)
  ```

  Analogamente, la subroutine intrinseca `RANDOM_NUMBER` ha un solo argomento la cui parola chiave è `HARVEST`. L'uso della parola chiave è pertanto facoltativo risultando perfettamente equivalenti le seguenti istruzione di invocazione di procedura:

  ```
  CALL RANDOM_NUMBER(HARVEST=myrand)
  CALL RANDOM_NUMBER(myrand)
  ```

  Se si considera, invece, la funzione `MAXVAL(ARRAY[,DIM][,MASK])`, dal momento che essa prevede tre argomenti di cui uno soltanto obbligatorio, se tutti e tre gli argomenti vengono specificati secondo l'ordine standard essi possono essere inclusi nella lista senza specificare le parole chiave. Se, invece, l'argomento `MASK` deve essere specificato in assenza di `DIM` allora bisognerà utilizzare la sua parola chiave:

  ```
  valore = MAXVAL(array,MASK=maschera)
  ```

  Il seguente prospetto elenca il tipo di dati delle parole chiave più tipiche:

`A`	Qualsiasi
`BACK`	Logico
`DIM`	Intero
`I`	Intero
`KIND`	Intero
`MASK`	Logico
`STRING`	Stringa
`X`, `Y`	Numerico (intero, reale o complesso)
`Z`	Complesso

- La seconda colonna riporta il *nome specifico* di una funzione intrinseca. Se questa colonna è vuota la procedura non ha un nome specifico e *non* può essere utilizzata come argomento in una chiamata di procedura. I tipi di dati utilizzati con le funzioni specifiche sono:

`c`, `c1`, `c2`, ...	Complesso di default
`d`, `d1`, `d2`, ...	Reale in doppia precisione
`i`, `i1`, `i2`, ...	Intero di default
`r`, `r1`, `r2`, ...	Reale di default
`l`, `l1`, `l2`, ...	Logico
`str1`, `str2`, ...	Stringa

  Come è noto, molte funzioni intrinseche possono essere utilizzate con argomenti di tipo diverso sia utilizzando sempre lo stesso nome generico sia utilizzando un nome specifico in relazione al tipo dell'argomento. Così, ad esempio, per valutare la radice quadrata di un oggetto `x` di tipo numerico ed assegnarla alla variabile `radice` è possibile scrivere:

```
radice = SQRT(x)
```

  indipendentemente dal tipo dell'argomento, oppure è possibile specificare il nome specifico a seconda del tipo dell'argomento (e del risultato):

```
radice_r = SQRT(x)  ! se x e' REAL singola precisione
radice_d = DSQRT(x) ! se x e' REAL doppia precisione
radice_c = CSQRT(x) ! se x e' COMPLEX
```

- La terza colonna contiene il tipo di valore restituito dalla procedura se questa è una funzione. Ovviamente, le subroutine intrinseche non sono associate a particolari tipi di dati.

Tabella E.1: Elenco alfabetico delle procedure intrinseche del Fortran

Nome generico e parole chiave	Nome specifico	Tipo della funzione
ABS(A)		Tipo dell'argomento
	ABS(r)	Reale di default
	CABS(c)	Reale di default
	DABS(d)	Doppia precisione
	IABS(i)	Intero di default
ACHAR(I)		Carattere
ACOS(X)		Tipo dell'argomento
	ACOS(r)	Reale di default
	DACOS(d)	Doppia precisione
ADJUSTL(STRING)		Stringa
ADJUSTR(STRING)		Stringa
AIMAG(Z)	AIMAG(c)	Reale
AINT(A[,KIND])		Tipo dell'argomento
	AINT(r)	Reale di default
	DINT(d)	Doppia precisione
ALL(MASK[,DIM])		Logico
ALLOCATED(ARRAY)		Logico
ANINT(A[,KIND])		Tipo dell'argomento
	ANINT(r)	Reale
	DNINT(d)	Doppia precisione
ANY(MASK[,DIM])		Logico
ASIN(X)	ASIN(r)	Tipo dell'argomento
	ASIN(r)	Reale
	DASIN(d)	Doppia precisione
ASSOCIATED(POINTER[,TARGET])		Logico
ATAN(X)		Tipo dell'argomento
	ATAN(r)	Reale
	DATAN(d)	Doppia precisione
ATAN2(Y,X)		Tipo dell'argomento
	ATAN2(r2,r1)	Reale
CEILING(A[,DIM])		Intero
CHAR(I[,DIM])		Carattere
CMPLX(X,Y[,DIM])		Complesso
CONGJ(X)	CONJG(c)	Complesso
COS(X)		Tipo dell'argomento
	CCOS(c)	Complesso
	COS(r)	Reale
	DCOS(d)	Doppia precisione
COSH(X)		Tipo dell'argomento
	COSH(r)	Reale
	DCOSH(d)	Doppia precisione
COUNT(MASK[,DIM])		Intero
CPU_TIME(TIME)		Subroutine
CSHIFT(ARRAY[,SHIFT][,DIM])		Tipo dell'array

Nome generico e parole chiave	Nome specifico	Tipo della funzione
DATE_AND_TIME([DATE][,TIME][,ZONE][,VALUES])		Subroutine
DBLE(A)		Doppia precisione
DIGITS(X)		Intero
DIM(X,Y)		Tipo dell'argomento
	DDIM(d1,d2)	Doppia precisione
	DIM(r1,r2)	Reale
	IDIM(i1,i2)	Intero
DOT_PRODUCT(VECTOR_A,VECTOR_B)		Tipo dell'argomento
DPROD(X,Y)	DPROD(x1,x2)	Doppia precisione
EOSHIFT(ARRAY,SHIFT[,BOUNDARY][,DIM])		Tipo dell'array
EPSILON(X)		Reale
EXP(X)		Tipo dell'argomento
	CEXP(c)	Complesso
	DEXP(d)	Doppia precisione
	EXP(r)	Reale
EXPONENT(X)		Intero
FLOOR(A[,KIND])		Intero
FRACTION(X)		Reale
HUGE(X)		Tipo dell'argomento
IACHAR(C)		Intero
IAND(I,J)		Intero
IBCLR(I,POS)		Tipo dell'argomento
IBITS(I,POS,LEN)		Tipo dell'argomento
IBSET(I,POS)		Tipo dell'argomento
ICHAR(C)		Intero
IEOR(I,J)		Tipo dell'argomento
INDEX(STRING,SUBSTRING[,BACK])	INDEX(str1,str2)	Intero
INT(A[,DIM])		Intero
	IDINT(i)	Intero
IFIX(r)		Intero
IOR(I,J)		Tipo dell'argomento
ISHFT(I,SHIFT)		Tipo dell'argomento
ISHFTC(I,SHIFT[,SIZE])		Tipo dell'argomento
KIND(X)		Intero
LBOUND(ARRAY[,DIM])		Intero
LEN(STRING)	LEN(str)	Intero
LEN_TRIM(STRING)		Intero
LGE(STRING_A,STRING_B)		Logico
LGT(STRING_A,STRING_B)		Logico
LLE(STRING_A,STRING_B)		Logico
LLT(STRING_A,STRING_B)		Logico
LOG(X)		Tipo dell'argomento
	ALOG(r)	Reale
	CLOG(c)	Complesso
	DLOG(d)	Doppia precisione

Nome generico e parole chiave	Nome specifico	Tipo della funzione
LOG10(X)		Tipo dell'argomento
	ALOG10(r)	Reale
	DLOG10(d)	Doppia precisione
LOGICAL(L[,KIND])		Logico
MATMUL(MATRIX_A,MATRIX_B)		Tipo dell'argomento
MAX(A1,A2[,A3,...])		Tipo dell'argomento
	AMAX0(i1,i2[,...])	Reale
	AMAX1(r1,r2[,...])	Reale
	DMAX1(d1,d2[,...])	Doppia precisione
	MAX0(i1,i2[,...])	Intero
	MAX1(r1,r2[,...])	Intero
MAXEXPONENT(X)		Intero
MAXLOC(ARRAY[,DIM][,MASK])		Intero
MAXVAL(ARRAY[,DIM][,MASK])		Tipo dell'argomento
MERGE(TSOURCE,FSOURCE,MASK)		Tipo dell'argomento
MIN(A1,A2[,A3,...])		Tipo dell'argomento
	AMIN0(i1,i2[,...])	Reale
	AMIN1(r1,r2[,...])	Reale
	DMIN1(d1,d2[,...])	Doppia precisione
	MIN0(i1,i2[,...])	Intero
	MIN1(r1,r2[,...])	Intero
MINEXPONENT(X)		Intero
MINLOC(ARRAY[,DIM][,MASK])		Intero
MINVAL(ARRAY[,DIM][,MASK])		Tipo dell'argomento
MOD(A,P)		Tipo dell'argomento
	AMOD(r1,r2)	Reale
	MOD(i,j)	Intero
	DMOD(d1,d2)	Doppia precisione
MODULO(A,P)		Tipo dell'argomento
MVBITS(FROM,FROMPOS,LEN,TO,TOPOS)		Subroutine
NEAREST(X,S)		Reale
NINT(A[,KIND])		Intero
	IDNINT(i)	Intero
	NINT(x)	Intero
NOT(I)		Tipo dell'argomento
NULL([MOLD])		Puntatore
PACK(ARRAY,MASK[,VECTOR])		Tipo dell'argomento
PRECISION(X)		Intero
PRESENT(A)		Logico
PRODUCT(ARRAY[,DIM][,MASK])		Tipo dell'argomento
RADIX(X)		Intero
RANDOM_NUMBER(HARVEST)		Subroutine
RANDOM_SEED([SIZE][,PUT][,GET])		Subroutine
RANGE(X)		Intero

Nome generico e parole chiave	Nome specifico	Tipo della funzione
REAL(A[,KIND])		Reale
	FLOAT(i)	Reale
	SNGL(d)	Reale
REPEAT(STRING,NCOPIES)		Stringa
RESHAPE(SOURCE,SHAPE,PAD[,ORDER])		Tipo dell'argomento
RRSPACING(X)		Tipo dell'argomento
SCALE(X,I)		Tipo dell'argomento
SCAN(STRING,SET[,BACK])		Intero
SELECTED_INT_KIND(R)		Intero
SELECTED_REAL_KIND([P][,R])		Intero
SET_EXPONENT([X][,I])		Tipo dell'argomento
SHAPE(SOURCE)		Intero
SIGN(A,B)		Tipo dell'argomento
	DSIGN(d1,d2)	Doppia precisione
	ISIGN(i1,i2)	Intero
	SIGN(r1,r2)	Reale
SIN(X)		Tipo dell'argomento
	CSIN(c)	Complesso
	DSIN(d)	Doppia precisione
	SIN(r)	Reale
SINH(X)		Tipo dell'argomento
	DSINH(d)	Doppia precisione
	SINH(r)	Reale
SIZE(ARRAY[,DIM])		Intero
SPACING(X)		Tipo dell'argomento
SPREAD(SOURCE,DIM,NCOPIES)		Tipo dell'argomento
SQRT(X)		Tipo dell'argomento
	CSQRT(c)	Complesso
	DSQRT(d)	Doppia precisione
	SQRT(r)	Reale
SUM(ARRAY[,DIM][,MASK])		Tipo dell'argomento
SYSTEM_CLOCK(COUNT,COUNT_RATE,COUNT_MAX)		Subroutine
TAN(X)		Tipo dell'argomento
	DTAN(d)	Doppia precisione
	TAN(r)	Reale
TANH(X)		Tipo dell'argomento
	DTANH(d)	Doppia precisione
	TANH(r)	Reale
TINY(X)		Reale
TRANSFER(SOURCE,MOLD[,SIZE])		Tipo dell'argomento
TRANSPOSE(MATRIX)		Tipo dell'argomento
TRIM(STRING)		Stringa
UBOUND(ARRAY[,DIM])		Intero
UNPACK(VECTOR,MASK,FIELD)		Tipo dell'argomento
VERIFY(STRING,SET[,BACK])		Intero

Bibliografia

[1] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, J. L. Wagner *Fortran 90 Handbook - Complete ANSI/ISO Reference*, McGraw-Hill, New York 1992.

[2] D. Bacon, S. Graham, O.Sharp *Compiler Transformations for High-Performance Computing*, Computing Surveys, No. 4, pp. 345-420, Dec. 1994.

[3] S. J. Chapman *Fortran 90/95 for Scientists and Engineers*, McGraw-Hill, Boston, 1998.

[4] T. M. R. Ellis, I. R. Philips, T. M. Lahey *Fortran 90 Programming*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[5] M. Metcalf, J. Reid *Fortran 90/95 Explained*, Oxford University Press, Oxford and New York, 1999.

[6] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery *Numerical Recipes in Fortran 90. Second Edition*, Cambridge University Press, Cambridge 1997.